



Transcompilation of VDM-SL to C#

Master's Thesis in Computer Science
by Steffen Pham Diswal 20117044

supervised by Aslan Askarov
co-supervised by Peter Gorm Larsen

Department of Computer Science
Department of Engineering
Aarhus University

15 June 2016

Abstract

The VDM Specification Language (VDM-SL) is a formal modelling language that focuses on high-level abstractions and design-by-contract to enable automatic software verification, which is particularly useful when developing mission-critical software systems that influence the lives of many people. Automatic code generation gives software engineers a convenient way to realise a VDM-SL model in an ordinary programming language, allowing interoperability with standard libraries and external software modules.

This thesis presents a set of rules for translating VDM-SL to C# and implements them in a proof-of-concept transcompiler as an extension to the Overture tool. The generated C# code is compared to the Java code produced by the Java code generator in Overture whose shortcomings are outlined when translating elements of the VDM-SL type system. The performances of the .NET Code Contracts library and the OpenJML tool are measured, revealing a significant computational overhead in the latter.

Unit tests have driven the development of the transcompiler, thereby exploring the use of test-driven development and automated tests in compiler construction. In particular, four different testing approaches are presented. The unit tests also serve as validation of the implementation in addition to a case study that evaluates the quality of the generated C# code.

Resumé

VDM Specification Language (VDM-SL) er et formelt modelleringssprog, der lægger vægt på højt abstraktionsniveau og kontraktbaseret programmering for at muliggøre automatisk softwareverifikation, hvilket især er brugbart i udviklingen af missionskritiske systemer, der får dagligdagen til at hænge sammen for mange mennesker. Automatisk kodegenerering er et bekvemt værktøj for softwareingeniører til at realisere en VDM-SL-model i et almindeligt programmeringssprog, således at der tages højde for interoperabilitet med standardbiblioteker og eksterne softwaremoduler.

Dette speciale præsenterer et regelsæt for oversættelse af VDM-SL til C# og implementerer det i en transcompiler-prototype som en udvidelse til Overture-værktøjet. Den genererede C#-kode sammenlignes med Java-koden produceret af Java-kodegeneratoren i Overture, hvis mangler bliver påpeget i forbindelse med oversættelsen af typesystemet i VDM-SL. Ydeevnen af .NET Code Contracts-biblioteket og OpenJML-værktøjet bliver også målt, hvilket afslører betydelige beregningsmæssige omkostninger ved brugen af sidstnævnte.

Unit tests har drevet udviklingen af transcompileren for på den måde at undersøge mulighederne i at anvende test-driven development og automatiserede tests til compilerkonstruktion. Fire forskellige tilgange bliver præsenteret i særdeleshed. Unit testene fungerer også som validering af implementeringen, foruden et casestudie der vurderer kvaliteten af den genererede C#-kode.

Acknowledgements

First of all, I would like to thank my academic supervisor, Peter Gorm Larsen, for his guidance, great patience and invaluable feedback on my work and for believing in this project. Without his efforts, I could not have accomplished this project.

I am grateful to Aslan Askarov for agreeing to be my formal supervisor and representative from the Department of Computer Science.

I owe a big thanks to Peter W. V. Tran-Jørgensen for introducing me to the Overture platform, helping me out with OpenJML and taking his time to assist me whenever I banged my head against a wall.

I would also like to thank the team behind JetBrains for giving their marvellous products to students for free. During the last four months, I have probably spent more time with IntelliJ than other human beings. This tool is truly an eye-opener.

Finally, I am eternally grateful to my friends and family for keeping me company when I needed a break and to my parents for their everlasting love and support.

Thank you, everyone!

Steffen Pham Diswal
Aarhus, 15 June 2016

Prerequisites

From a practical point of view, this thesis assumes that the reader is familiar with object-oriented programming in the Java language as well as object-oriented terms and design patterns. It also makes basic assumptions about the **reader's technical knowledge of** computer architecture.

From a theoretical point of view, this thesis assumes that the reader has good understanding of discrete mathematics, in particular propositional logic, set theory, language theory and theory of computation, including regular expressions and context-free grammars.



Table of Contents

Abstract.....	iii
Resumé	v
Acknowledgements.....	vii
Prerequisites	ix
Table of Contents	xi
1 Introduction	1
1.1 Objectives	2
1.2 Approach	3
1.3 Reading guide	4
2 Background	9
2.1 VDM-SL	10
2.2 JML	18
2.3 C# and .NET	20
2.4 Compilers	25
2.5 Automated testing	28
3 Specifications	31
3.1 Flat specifications.....	31
3.2 Modular specifications	36
4 Functionality	45
4.1 Functions and operations.....	46

4.2	Pre- and postconditions	53
4.3	States.....	56
4.4	Invariants	59
4.5	Values	63
5	Types	67
5.1	Numeric types.....	68
5.2	Collection types.....	72
5.3	Union types.....	83
5.4	Quote types	89
5.5	Composite types.....	93
5.6	Type aliases	100
5.7	Type invariants	105
6	Realisation	111
6.1	Prototype implementation	111
6.2	Automated testing	115
7	Results	125
7.1	An automated teller machine in VDM-SL	125
7.2	Benchmarking FAD codes.....	133
8	Conclusion	137
8.1	Validity of results	137
8.2	Objectives, revisited	138
8.3	Future work	140
	References	143
	Glossary	149
A	All translations	A-1
B	Runtime library	B-1

“ Engineering walks the bridge between science and technology.

– Dines Bjørner (1937-)

1 Introduction

Software systems influence the lives of people in many aspects, from observing the medical conditions of hospital patients to distributing electricity from power plants to customers. They conduct financial transactions and ensure safe land, sea and air transportation. They are even present in space. It is crucial that these systems operate correctly as lives and money depend on them. They are mission-critical software systems.

Being a formal modelling language, VDM-SL differs significantly from low-level programming languages by focusing on high-level abstractions rather than pointers and algorithms [1]. Software engineers do not need to care about memory leaks and numeric overflows because these issues never occur in VDM-SL. Combined with the design-by-contract elements of the language that assist automatic software verification, these language traits shine when modelling mission-critical systems. However, they come at a remarkable price: VDM-SL specifications are not meant to be executed as computer programs – they are merely descriptions of software that guide programmers to developing formally verified solutions. In fact, only a subset of all VDM-SL specifications are even executable!

Object-oriented programming languages like Java [2] and C# [3] bridge the gap between VDM-SL and low-level programming languages by introducing classes and automatic memory management through garbage collection. This brings along certain technical limitations and challenges that programmers need to deal with. A transcompiler is a great help here since it can translate the overall structure of a VDM-SL specification to a skeleton program that programmers may continue developing into the final, executable software solution. Nevertheless – even for a transcompiler – translating a VDM-SL specification into equivalent source code for a programming language is not straightforward due to the semantic and technical differences between the languages.

This thesis studies the differences between VDM-SL and C# in order to construct a transcompiler between the languages. It is motivated by the fact that C# targets the .NET platform, which is a large, general-purpose software platform for developing desktop, web and mobile applications as well as video games [4]. According to the TIOBE Index, C# is the fifth most popular programming language as of June 2016 [5]. With almost one million C#-tagged questions, it ranks third on Stack Overflow [6].

This chapter introduces the contents of this thesis by specifying its objectives and the scientific method used to approach them. It also guides the reader by outlining the structure of the following chapters and presenting the typographical conventions used in the text.



Chapter contents

Section 1.1 specifies the objectives of this thesis as three hypotheses.

Section 1.2 describes the scientific method used to verify or falsify the hypotheses and presents the platforms and tools applied for this purpose.

Section 1.3 outlines the chapter structure of this thesis and presents the typographical conventions of the text.

1.1 Objectives

The objectives of this thesis are expressed as three hypotheses to be verified or falsified in the following chapters. The first hypothesis compares the VDM-SL and C# languages:



Hypothesis I

VDM-SL versus C#

A large subset of the language constructs in VDM-SL have semantically equivalent constructs in C# that result in maintainable C# source code.

The second hypothesis investigates the use of test-driven development (TDD) for compiler construction:



Hypothesis II

TDD for compilers

Test-driven development is feasible for compiler construction without sacrificing developer productivity, analysability of the test code or quality in unit testing.

The third hypothesis compares the .NET Code Contracts library to the JML language and the OpenJML tool:



Hypothesis III

.NET Code Contracts versus JML

The .NET Code Contracts library supports the same set of design-by-contract elements in VDM-SL as JML. The performance of .NET Code Contracts surpasses the performance of OpenJML.

1.2 Approach

Regarding Hypothesis I, the question on semantic equivalence between the languages is inherently true as both languages are Turing complete. Therefore, all VDM-SL constructs can be implemented manually in a C# library. However, the point of Hypothesis I is to reduce the size of such a library to a minimum by utilising the built-in language constructs of C# to express VDM-SL constructs rather than programming them manually. The primary task is thus to translate as many VDM-SL constructs as possible to corresponding native C# constructs.

The VDM-SL language manual provides a good overview of language constructs of VDM-SL and is consulted as needed [7]. The C# language specification [3], the .NET guidelines [8] and the Code Contracts user guide [9] provide insight into the possibilities of the C# language and the .NET language.

The maintainability of source code is essential, see Definition 1.1. While there is a corresponding C# program for a VDM-SL specification, it is little use if the C# source code is unreadable and does not follow the guidelines of the .NET platform. The secondary task is thus to translate VDM-SL constructs into readable and flexible C# constructs, not just semantically equivalent C# constructs.



Definition 1.1

Maintainability

Maintainability is a quality that describes how easy it is to modify an existing software system in order to correct errors, improve performance and meet new requirements [10].

The translations from VDM-SL to C# are collected into a set of translation rules which are implemented in a proof-of-concept transcompiler from VDM-SL to C#. It is developed with test-driven development (TDD) [10] [11], cf. Hypothesis II.

TDD is meant as a software development technique rather than a software validation technique – it is therefore crucial that it does not pose an obstacle to the development, for example by hampering the developer productivity, see Definition 1.2, or comprising the quality of the code produced.

**Definition 1.2****Developer productivity**

Developer productivity is the ratio between the amount of code written versus the time spent writing it.

It is also important to evaluate the translations from VDM-SL to C#. Existing translation efforts for VDM-SL include the Java/JML code generator in the Overture tool [12] [13]. Java is similar to C# in many aspects and is therefore worth casting a glance at. The C# code produced by the transcompiler is compared to the Java code produced by the Java code generator in Overture to reveal differences and similarities between the C# and Java languages. The design-by-contract elements of JML [14] are compared to those of .NET Code Contracts and the computational performances of .NET and the OpenJML tool [15] are measured in a benchmark, cf. Hypothesis III.

1.2.1 Platforms and tools

The languages involved in the translation rules are the VDM-SL dialect of VDM-10 and C# 6.0 on .NET Framework 4.6. The implementation languages for the transcompiler are Java 7 and Kotlin 1.0 [16] on a 64-bit version of JDK 8.

All work regarding implementation and evaluation have been carried out using the integrated development environments Overture 2.3 (for VDM-SL) [17] [18], IntelliJ IDEA 2016 (for Java and Kotlin) [19] and Visual Studio 2015 (for C#) [20] on Windows 10.

Benchmarking has been performed on an HP desktop computer that runs the 64-bit edition of Windows 10 and is equipped with a 3.4 GHz Intel i7 Skylake processor and 16 GB RAM. The C# benchmarks have been run on .NET CLR 4.0, whereas the JML benchmarks have been run on Java HotSpot VM 1.8 after being built on OpenJDK 7 via an Ubuntu 14 virtual machine.

1.3 Reading guide

This chapter has defined the objectives of this thesis and outlined the method used to approach them. The remaining chapters should be read in order.

Chapter 2 explains the concepts and languages treated by this thesis to give the reader fundamental insight into understanding the approaches and results presented in the later chapters.

Chapters 3 to 5 discuss the translations of VDM-SL constructs to C# constructs and present sets of translation rules concerning specifications, functionality and types, respectively. They do not cover all VDM-SL constructs exhaustively, but focus on the ones that give rise to non-trivial translations or are significantly different from the equivalent Java translation. The sections in these chapters follow a common structure, which is described in Section 1.3.1. The complete set of translations realised in this thesis is presented in Appendix A. Some of the translations depend on manually implemented features which are located in the transcompiler runtime library and presented in Appendix B.

Chapter 6 presents a prototype implementation of the translations for the Overture platform. It investigates the feasibility of test-driven development for compiler construction and presents four testing approaches on a syntactical level.

Chapter 7 evaluates the quality of the translations and the prototype in a case study and measures the performance of .NET Code Contracts versus OpenJML.

Chapter 8 concludes on the achievements of this thesis, discusses the validity of the results and presents enhancements to be done in the future.

1.3.1 Section structure for translations

The sections in Chapters 3 to 5 present approaches to translating VDM-SL constructs to C# constructs. Each section follows the same structure:

To begin with, it briefly introduces the VDM-SL concept to be translated – complemented by an example with VDM-SL source code – and outlines the possible approaches in C# and .NET.



Example 1.3

Listing 1.3a shows a fragment of a VDM-SL specification which calls the `println` operation in the `IO` library to print a message in the console.

Listing 1.3a

VDM-SL

```
1 let greeting = "I am VDM-SL."  
2 in IO`println(greeting)
```

The most natural approach in C# is presented first, defining the relevant language constructs and discussing the advantages and disadvantages of that particular approach. The VDM-SL example from before is followed up in this section, showing a proof-of-concept translation from VDM-SL to C# using the approach just described.



Example 1.3, continued

Listing 1.3a showed how to print a message in the console from VDM-SL. Listing 1.3b shows the equivalent code in C# by calling `Console.WriteLine`.

Listing 1.3b

C#

```
1 var greeting = "I am C#.";
2 Console.WriteLine(greeting);
```

In some cases, there are alternative approaches to the one just presented. They are presented next, following the same structure as the first approach. After all approaches have been presented, they are weighed against each other with regard to their advantages and disadvantages as well as their resemblance to the original VDM-SL concept. The best approach is then chosen accordingly **and summarised as a ‘translation rule’, in this case by Rule 1.4**. Sometimes, the C# translation has certain limitations such as semantic differences from VDM-SL and technical issues. These potential limitations are described immediately after the summarisation of the translation rule.



Rule 1.4

The chosen translation rule is described in this box.



Limitations

Rule 1.4

The potential limitations of the translation rule are described in this box. If there are no apparent limitations, this box is omitted.

Finally, the translation performed by the Java code generator in the Over-ture tool is presented. It has been configured to output Java code with JML annotations. The corresponding Java code for the previously presented VDM-SL example is outlined, here in Listing 1.3c, and compared to the C# approach.

In most of the cases, the generated Java code will include language constructs from JML; however, only the essential JML constructs are preserved in the code example, omitting many assertions on type consistency that would otherwise take up too much space. **It is still referred to as ‘Java’ code rather than ‘JML’ code.**

```
Listing 1.3c Java  
1 string greeting = "I am Java.";  
2 System.out.println(greeting);
```

1.3.2 Typographical conventions

The text in the thesis uses the following conventions:

Bold text highlights terms or words of particular importance as a means to guide the reader.

Italic text emphasises a particular word in the given context.

TEXT IN SMALL CAPS highlights terms that are further explained in the Glossary chapter located at the end of the thesis. Only the first occurrence of a term is highlighted like this.

Monospace text denotes references to code fragments such as identifiers. Whenever the text refers strictly to the name of an identifier, it is not highlighted like this.

Bold monospace text denotes references to keywords in code. Whenever a keyword is used as an adjective in the text, it is not highlighted like this.

[X] indicates a reference to other scientific or industrial work. All citations are listed in the References chapter in order of appearance throughout the text. They follow the IEEE citation style.

“ Essentially, all models are wrong, but some are useful.

– George E. P. Box (1919-2013)

Background

The concepts treated in this thesis include software modelling, design-by-contract, compilation and automated testing. The results relate to the languages VDM-SL, C#, Java and JML. Therefore, the reader is recommended to be familiar with these concepts and languages before reading the remainder of this thesis.

This chapter explains these concepts and languages and elaborates on the technicalities of their platforms to give the reader a fundamental insight into understanding the approaches and results presented in the following chapters.



Chapter contents

Section 2.1 describes the syntax and semantics of the VDM Specification Language (VDM-SL); defines the design-by-contract paradigm and presents the Overture tool.

Section 2.2 describes of the design-by-contract elements in the Java Modelling Language (JML) and presents the OpenJML tool.

Section 2.3 describes the syntax and semantics of the C# language as well as the architecture, features and conventions of the .NET platform.

Section 2.4 describes the phases of compilation; defines the concepts of abstract syntax trees and intermediate representations; explains how they relate to each other and presents the code generation platform of Overture.

Section 2.5 explains the concepts of automated testing and test-driven development.

2.1 VDM-SL

The Vienna Development Method (VDM) is a model-oriented software development process, invented in 1973 by Dines Bjørner and Cliff Jones at IBM Laboratory Vienna [21]. It describes a stepwise approach for transforming requirements of a software system into a MODEL. The VDM Specification Language (VDM-SL), standardised by ISO in 1996, defines the syntax and semantics for specifying VDM models. It is strongly influenced by the notations of discrete mathematics. This thesis studies the variant of VDM-SL that is a part of the VDM-10 language revision [7]. It has two siblings, VDM++ and VDM Real Time Language (VDM-RT), which extend VDM-SL with class-based object-orientation, concurrency and real-time analysis for distributed systems. They are beyond the scope of this thesis, though.

A VDM-SL specification is a description of a software system, expressed as a model. It contains definitions of types (see Section 2.1.2), functions (see Section 2.1.3), operations (see Section 2.1.3), states (see Section 2.1.4) and values (see Section 2.1.4), characterising what information the system processes and what actions it performs. A definition is always declared in a definition block, which is a section in the specification that holds definitions of a certain kind. For example, a function definition must be located in a definition block denoted by the `functi ons` keyword, and a type definition must be located in a definition block denoted by the `types` keyword.



Example 2.1

Definition blocks

Listing 2.1a shows a type definition `I tem` (line 2) located in the `types` block, two value definitions `VAT` (line 5) and `Pri ces` (line 6) in the `val ues` block and a function definition `ComputePri ce` (lines 9-10) in the `functi ons` block.

Listing 2.1a

VDM-SL

```

1  types
2      I tem = <Hat> | <Sungl asses>;
3
4  val ues
5      VAT = 0.18;
6      Pri ces = { <Hat> |-> 14.99, <Sungl asses> |-> 36.99 };
7
8  functi ons
9      ComputePri ce: I tem -> rat
10     ComputePri ce(i tem) == Pri ces(i tem) * (1 + VAT);

```

2.1.1 Modules

There are two kinds of specifications in VDM-SL: flat specifications and modular specifications. A flat specification has a single global environment that contains all definitions, whereas a modular specification spreads the definitions across separate modules. A module has its own environment of definitions, which is isolated from the environments of other modules. It is defined via the `module` keyword. Definitions appear in definition blocks as they do in flat specifications, but they must be preceded by the `definitions` keyword within the enclosing module definition.

By default, definitions within a module are inaccessible from other modules due to encapsulation. However, a module may define an interface of exported definitions that will be publicly accessible from other modules.

This interface is declared with the `exports` keyword above the definition blocks and lists the signatures of the definitions to export. The list of signatures is divided into blocks, similar to definition blocks. For example, the signature of an exported function definition must appear in an interface block named `functions`. The same goes for operations, types and values. When exporting a type, only its name is exposed by default. To expose the internal structure of a type, its entry in the exports interface must be marked with the `struct` keyword. The module state is always private and can only be exposed through getter and setter operations. If all definitions in the module are to be fully exported, including the structure of types, the interface says so with the `all` keyword rather than listing all definition signatures individually.

Other modules cannot use exported definitions right away. They must import the needed definitions first through an interface of imported types. It is declared by the `imports` keyword and lists the names of the imported definitions and the modules that define them. The `all` keyword can be used to import all definitions from a module. It is possible to expose an imported definition under a different name in the module by renaming it in the interface.



Example 2.2

Modules

Listing 2.2a shows some four definitions from Listing 2.1a defined within a module named `ShoppingSystem`. The `Item` type definition and the `Pri ces` value definition are exported (lines 3-4).

Listing 2.2a

VDM-SL

```

1  module ShoppingSystem
2
3  exports types Item
4         value Prices
5
6  definitions
7  types
8      Item = <Hat> | <Sunglasses>
9
10 values
11     VAT = 0.18;
12     Prices = { <Hat>          |-> 14.99 ,
13               <Sunglasses> |-> 36.99 };
14
15 functions
16     ComputePrice: Item -> rat
17     ComputePrice(item) == Prices(item) * (1 + VAT);
18
19 end ShoppingSystem

```

2.1.2 Types

VDM-SL has a static typing discipline and distinguishes between two main groups of types: basic types and compound types.

Basic types represent atomic values and include `bool` (\mathbb{B}), `char`, `int` (\mathbb{Z}), `nat` (\mathbb{N}_0), `nat1` (\mathbb{N}_1), `rat` (\mathbb{Q}), `real` (\mathbb{R}), `token` and `quotes`.

A token is a distinct value that can only be compared to other tokens for equality. The token type corresponds to the set of tokens.

A quote type corresponds to the singleton set of a quote literal representing a constant, named identifier. Quote types are comparable to enumerated types of programming languages. The `Item` type in Listing 2.2a consists of two quote types `<Hat>` and `<Sunglasses>`.

Compound types represent structurally complex values such as sets, sequences, mappings, tuples, data records and untagged unions.

A set is a finite collection of unordered items without duplications denoted by `set of T`, where `T` is the type of the items. A sequence is a finite collection of ordered items with possible duplications denoted by `seq of T`. A mapping is a finite collection of unordered key-value pairs where all keys are unique. It is denoted by `map T to U`, where `T` is the domain type and `U` is the range type.

A tuple, formally known as a *product type*, is a fixed-size vector of elements of distinct types. It is denoted by $T_1 * T_2 * \dots * T_N$ for component types T_1, T_2, \dots, T_N .

A record, formally known as a *composite type*, is a named, fixed-size vector of data fields. The syntax for a record **named 'Id' is** $\text{Id} : : f_1 : T_1 f_2 : T_2 \dots f_n : T_N$ for fields of names f_1, f_2, \dots, f_n and types T_1, T_2, \dots, T_N .

An untagged union is a type that accepts values from multiple constituent types. It is denoted by $T_1 \mid T_2 \mid \dots \mid T_N$ for constituent types T_1, T_2, \dots, T_N . A special case of unions is the optional type: $[T] = T \mid \text{nil}$.

A type can be exposed under multiple, interchangeable names using type aliases. Such types can also be constrained to smaller sets of legal values by using invariants, which are explained in Section 2.1.5. Composite types can also be constrained by invariants. They are defined in the `types` block of the specification.

Besides supplying built-in types, VDM-SL provides built-in operations for manipulating typed expressions [7]. All types exhibit PASS-BY-VALUE SEMANTICS and STRUCTURAL EQUALITY.

2.1.3 Functions and operations

There are two kinds of units of computation in VDM-SL: functions and operations. Operations are further divided into pure operations and impure operations. Functions are defined in the `functions` block of the specification, whereas operations are defined in the `operations` block.

A function represents a REFERENTIALLY TRANSPARENT unit of computation free of SIDE EFFECTS. It takes a series of arguments as input and produces a result value; hence, the body of a function is an expression. It is allowed to call other functions and pure operations, but may not call impure operations. Additionally, VDM-SL supports PARAMETRICALLY POLYMORPHIC functions for which the types of the parameters can be parametrised by type variables.

A pure operation is free of side effects like a function, but has an extra capability: It is allowed to read the fields of the state and use their values in the computation. States are described in Section 2.1.4. The body of an operation is a statement rather than an expression. Thus, a pure operation is obliged to use the VDM-SL `return` keyword to indicate a result value. VDM-SL does not support parametrically polymorphic operations.

An impure operation is a REFERENTIALLY OPAQUE unit of computation that is allowed to have side effects. It may both read and update the fields of the state and call all kinds of functions and operations. It is allowed to produce a result value, but not required to.

Table 2.3 summarises the capabilities of functions and operations.

	May read the fields of the state	May update the fields of the state	May call functions	May call pure operations	May call impure operations	Must produce a result value	Is free of side effects	Supports parametric polymorphism
<i>Functions</i>			✓	✓		✓	✓	✓
<i>Pure operations</i>	✓		✓	✓		✓	✓	
<i>Impure operations</i>	✓	✓	✓	✓	✓			

2.1.4 States and values

A state is a collection of variables, called *fields*, which persist information and can be accessed and reassigned at any time from an operation within their scope, cf. Table 2.3. The collection of fields in the state corresponds to a composite type definition although only one state is allowed per flat specification or per module. It is always private to a module and cannot be exported.

A state is defined in the `state` block. The information in the fields can be initialised immediately via the `init` keyword. Like composite types, the state can be protected by an invariant, see Section 2.1.5.

A value defines a named constant that can be accessed from anywhere in the flat specification or module. It is defined in the `values` block.

2.1.5 Design-by-contract

Design-by-contract is a programming paradigm that originates from Hoare logic [22]. It recommends that the programmer annotates software components with contracts, which are mutual agreements between collaborating software components on what they can expect from each other.

Under the assumption that the other components fulfil their part of the contract, it is reasonable to assume that the properties prescribed by the contract are satisfied, hence eliminating the need for DEFENSIVE PROGRAMMING. Should a contract be violated, it is evident which software component has failed. While the contracts are indeed checked during software development to uncover potential bugs, they are usually disabled in production mode to save the computational overhead of contract assertions.

Preconditions and postconditions specify contracts on computational software units, see Definitions 2.4 and 2.5. They ensure that the input and output of computations always meet the expectations.

✱	<div>Definition 2.4</div> <div>Precondition</div> <p>A precondition is a logical predicate that is <i>assumed</i> to hold right <i>before</i> a computation. If it is violated, the outcome of the computation is undefined.</p>
✱	<div>Definition 2.5</div> <div>Postcondition</div> <p>A postcondition is a logical predicate that is <i>guaranteed</i> to hold right <i>after</i> a computation.</p>

They are supported natively by VDM-SL through the `pre` and `post` clauses where they specify contracts on functions and operations.

The logical predicate contained in a precondition is defined behind the scenes as a function prefixed by ‘`pre_`’ **with** the same series of parameters as the function or operation it is guarding. When it guards an operation, it takes an additional parameter that contains the state upon entry – unless the specification does not contain a state definition in which case it is left out.

The logical predicate contained in a postcondition becomes a corresponding function prefixed by ‘`post_`’. It takes the same parameters as input as the function or operation it is guarding plus the result value as an additional parameter although it leaves the latter out when it guards an impure operation that does not produce a result value. If the specification defines a state, it also takes a pair of the state values upon entry and upon exit. The state upon entry is known as the pre-state or the ‘before’ state, as opposed to the post-state or ‘after’ state upon exit. Within the `post` clause, the predicate can refer to the result value via the `RESULT` keyword as well as the pre-state by suffixing the respective field with a tilde (~) character.



Example 2.6

Pre- and postconditions

Listing 2.6a shows a function f (lines 2-5) and an operation op (lines 14-17) whose bodies have been omitted for the sake of brevity using the `is not yet specified` construct of VDM-SL. They are both guarded by pre- and postconditions, take three parameters of types X , Y and Z as input and produce a value of type R as output.

Let e_f and e_{op} be the logical predicates for the preconditions of f and op and let $e_{f'}$ and $e_{op'}$ be the logical predicates for their postconditions. VDM-SL implicitly wraps them into the Boolean function definitions `pre_f` (lines 7-8), `post_f` (lines 10-11), `pre_op` (lines 20-21) and `post_op` (lines 23-24). The state s of type St is passed as parameters to `pre_op` and `post_op`.

Listing 2.6a

VDM-SL

```

1  functions
2      f: X * Y * Z -> R
3      f(x, y, z) == is not yet specified
4      pre pre_f(x, y, z)
5      post post_f(x, y, z, RESULT);
6
7      pre_f: X * Y * Z +> bool
8      pre_f(x, y, z) == e_f(x, y, z);
9
10     post_f: X * Y * Z * R +> bool
11     post_f(x, y, z, r) == e_{f'}(x, y, z, r);
12
13  operations
14     op: X * Y * Z ==> R
15     op(x, y, z) == is not yet specified
16     pre pre_op(x, y, z, s)
17     post post_op(x, y, z, RESULT, s~, s);
18
19  functions
20     pre_op: X * Y * Z * St +> bool
21     pre_op(x, y, z, s) == e_{op}(x, y, z, s);
22
23     post_op: X * Y * Z * R * St * St +> bool
24     post_op(x, y, z, r, s~, s) == e_{op'}(x, y, z, r, s~, s);

```

In addition to defining contracts on computational units, it is possible to define contracts on the program state by specifying invariants, see Definition 2.7. They ensure that the program always finds itself in a consistent state.



Definition 2.7

Invariant

An invariant is a logical predicate that is *assumed* to hold at *any point in time* during certain periods of the program execution, in some cases during the entire lifetime of the program. If it is violated, the behaviour of the program is undefined.

VDM-SL supports type invariants through the `inv` clause, which constrain the set of valid values that a typed piece of data can hold. They are applicable to type aliases and composite types. State invariants constrain the values held by the state fields. Since states are defined implicitly as composite types in VDM-SL, state invariants are a special case of composite type invariants. Like pre- and postconditions, the logical predicate of an invariant is automatically defined as **a function prefixed by ‘inv_’** that takes the piece of information it guards as input.

Once a state invariant – or a composite type invariant – is satisfied, it is guaranteed to remain satisfied until the state is modified which can happen during assignments to the state fields. Sometimes, a state invariant specifies a condition that involves multiple fields. Updating the values of the fields one-by-one may temporarily violate the state invariant and must therefore be done using the ‘multiple assignments’ **construct of VDM-SL** which is denoted by the `atomic` keyword. It disables the state invariant checking while carrying out multiple separate assignments and re-enables it afterwards, thus disregarding any temporary violations [7].



Example 2.8

Invariants

Listing 2.8a shows a state `St` (lines 1-5) with two fields `x` and `y` of type `int`. Let `est` be the logical predicate for the invariant of `St`. VDM-SL implicitly wraps it into the Boolean function definition `inv_St` (lines 8-9).

Listing 2.8a

VDM-SL

```

1  state St of
2      x: int
3      y: int
4      inv s == inv_St(s)
5  end
6
7  functions
8      inv_St: St => bool
9      inv_St(s) == est(s);

```

2.1.6 The Overture tool

Overture is an Eclipse-based INTEGRATED DEVELOPMENT ENVIRONMENT for VDM-SL, VDM++ and VDM-RT [18]. With its built-in interpreter, it is capable of running executable VDM specifications to validate their correctness, especially when it comes to design-by-contract elements which, by default, throw exceptions when they are violated.

It is also equipped with a combinatorial testing tool for exhaustive and partial model checking, a debugger and a proof obligation tool that uses static analysis on the design-by-contract elements to assist the modeller in verifying the correctness of the specification.

Finally, Overture supports automatic code generation from VDM specifications to Java as well as experimental code generation to C++. From the perspective of this thesis, it is the most interesting feature of Overture. The code generation platform of Overture is explained in greater detail in Section 2.4.3.

2.2 JML

The Java Modelling Language (JML) is an enhancement of the Java programming language that improves the support for design-by-contract elements [14]. Invented in 1999 by Gary T. Leavens, Albert L. Baker and Clyde Ruby, it relies on a standalone compiler to transform JML specifications into Java bytecode.

JML adds support for formally specified pre- and postconditions on Java methods along with class invariants. They are defined using specialised comments – annotation comments – which begin with an at (@) character. Therefore, a JML program can be compiled by a Java compiler, which will simply disregard the comments with the contract annotations. The JML annotation syntax is explained in the following two sections.

2.2.1 Pre- and postconditions

Pre- and postconditions in JML are specified on methods with the `requires` and `ensures` annotations, respectively, as shown in Listing 2.9. `requires` takes a logical predicate as input that must be satisfied upon method entrance. Similarly, the logical predicate in `ensures` must be satisfied upon method exit.

Within a postcondition, the logical predicate may refer to the return value of the method via the `\result` annotation. It may also retrieve the pre-state via the `\old` annotation, which takes an expression as input and outputs its value at method entrance. Any method called from `requires` or `ensures` must be marked with the `pure` modifier of JML to indicate that it is referentially transparent [23].

Listing 2.9

Java

```
1  //@ requires n >= 0;
2  //@ ensures n * n == \result;
3  public static double SquareRoot(double n) { /* Omitted */ }
```

2.2.2 Invariants

Class invariants in JML are specified with the `instance invariant` annotation, see Listing 2.10. Static invariants are specified similarly with the `static invariant` annotation. By default, invariants declared in classes are class invariants and invariants declared in interfaces are static invariants. In those cases, they may leave out the `instance` or `static` modifier.

Invariants are checked upon entrance and exit of methods that are not annotated with the `helper` modifier – in some sense, they have become implicit pre- and postconditions. The `helper` modifier is applied to helper methods in order to avoid infinite cycles of invariant checking.

Listing 2.10

Java

```
1  public final class Date
2  {
3      //@ invariant day >= 1 && day <= 31;
4      private final int day;
5
6      //@ invariant month >= 1 && month <= 12;
7      private final int month;
8
9      //@ invariant year >= 1900 && year <= 2100;
10     private final int year;
11
12     // ...
13 }
```

2.2.3 Tool support

There are multiple tools available for JML; however, many of them targets ancient versions of Java and are obsolete in a modern setting. One of the tools that are still maintained is OpenJML [15], which targets OpenJDK 7. It provides a runtime checker that transforms JML annotations into native Java runtime assertions, which are normally specified with the `assert` keyword [2].

The Java code generator in Overture supports translating VDM-SL contracts to JML annotations [13] [18]. Jørgensen et al. have conducted an experiment on translating VDM-SL traces¹ to Java and benchmarking their execution time with and without JML annotations [24]. This experiment has revealed a significant computational overhead originating from the runtime checker in OpenJML. Chapter 7 revisits this experiment.

2.3 C# and .NET

.NET is a software platform, invented by Microsoft in 2001 [4]. The core of .NET is the ISO- and Ecma-standardised Common Language Infrastructure (CLI) specification [25]. It defines a platform-independent assembly language, Common Intermediate Language (CIL), as well as the architecture of the corresponding runtime environment, Virtual Execution System (VES), see Figure 2.11. Additionally, it defines a set of standard libraries that can be used by all CLI-compliant programs.

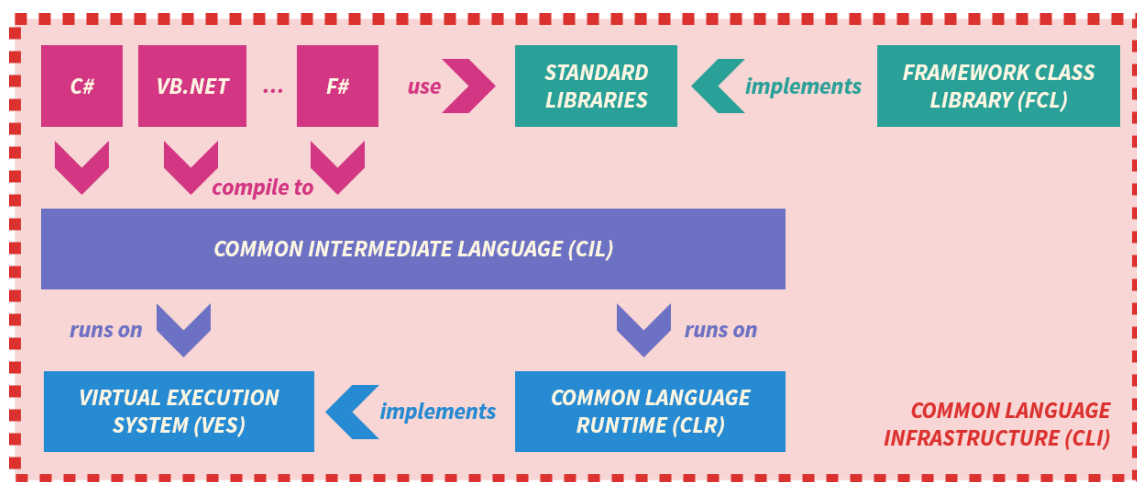


Figure 2.11

The architecture of the Common Language Infrastructure.

¹ A *trace* is a definition within a VDM-SL specification that directs the combinatorial testing tool in Overture to expose potential contract violations in functions and operations.

The primary implementation of the CLI targets the Windows operating systems under the brand *.NET Framework*. In this context, the Common Language Runtime (CLR) is the implementation of the runtime environment. It is a virtual machine that executes programs by JUST-IN-TIME COMPILATION and provides services for GARBAGE COLLECTION and EXCEPTION HANDLING. Framework Class Library (FCL) is the implementation of the standard libraries. Other implementations of the CLI include the cross-platform Mono and .NET Core projects as well as .NET Micro Framework for ARM-based embedded systems.

Alongside the development of .NET, a development team at Microsoft lead by Anders Hejlsberg conceived the C# programming language [3]. Primarily influenced by C++ and Java, it is a class-based object-oriented language that targets the .NET platform by compiling to CIL. Its syntax resembles that of Java to a large extent.

2.3.1 Types and namespaces

C# has a static typing discipline; hence, all values are typed. Moreover, C# is a pure object-oriented language, meaning that all values are represented as objects. This is slightly different from Java, which has a concept of primitive types whose values are not represented as objects.

A new type can be defined by declaring a class, an interface, an enum or a struct or by instantiating an anonymous type. Classes and interfaces in C# are equivalent to their Java counterparts. C# enums are enumerations of integer-valued constants. Unlike enum types in Java, they may not define methods. C# structs support the same functionality as classes; however, they also exhibit pass-by-value semantics and structural equality – all structs implicitly extend the abstract `ValueType` class which provides these features.

By default, classes exhibit PASS-BY-SHARING SEMANTICS and REFERENTIAL EQUALITY. This behaviour can be overridden by implementing certain methods in the classes. Classes that only contain data and possibly exhibit pass-by-value semantics and structural equality like structs are called *data classes* or Plain Old CLR Objects (POCOs). An anonymous type is an example of a data class. It defines a list of data fields and exhibits structural equality.

C# supports parametric polymorphism through generics. Unlike Java, the .NET platform does not have a concept of type erasure, which means the generic type information is available both at compile-time and at runtime.

Types are organised in namespaces, which correspond to packages in Java. A type in one namespace can refer to types in other namespaces by qualifying their type names with the name of their namespace. Alternatively, they can be imported via a `using` directive to avoid the namespace qualification. Most types of the standard library reside in the `System` namespace or one of its subordinate namespaces.

Classes that extend the abstract `System.Attribute` class are attributes. They can augment other types with extra information or behaviour. They are applied as tags on the other types using a special square bracket notation, see Listing 2.12. **By convention, the names of all attributes end with ‘Attribute’.** This suffix is left out in the tag. Attributes in C# correspond to annotations in Java.

Listing 2.12

C#

```
1 public class SomeAttribute : Attribute { }
2
3 [Some] // Attribute tag for SomeAttribute.
4 public class SomeTaggedClass { }
```

2.3.2 Expression-bodies

A method defines a computation performed by a type or an instance of a type. The former is a *static method*, the latter is an *instance method*. A method takes a series of arguments as input and is allowed to produce a result, called a *return value*. The body of a method is a statement; hence, in order to return a value, the `return` keyword must be used. There is an exception to this rule, though. If the method only contains a return statement, it may use the expression-body syntax instead in which case the `return` keyword is omitted, as shown in Listing 2.13.

Listing 2.13

C#

```
1 public int Identity1(int n)
2 {
3     return n; // Ordinary return statement.
4 }
5
6 public int Identity2(int n) => n; // Expression-body.
```

Expression-bodies are also allowed for methods that only contain a single call to another method or only perform a single assignment.

2.3.3 .NET Code Contracts library

C# supports the design-by-contract paradigm through the .NET Code Contracts library [9], which is a spin-off from the Spec# research project [26]. It resides in the `System.Diagnostics.Contracts` namespace and supports preconditions, postconditions and invariants like VDM-SL as well as explicit assertions. It throws exceptions when contracts are violated.

All contractual elements are conditionally compiled, which means that they are only emitted in the CIL bytecode when the Code Contracts library is enabled. This is achieved by defining a pre-processor symbol named `CONTRACTS_FULL` at the top of the source file, as shown in Listing 2.14. By leaving out this symbol definition, the program runs without the overhead of contract enforcement.

Listing 2.14

C#

```
1 // Include this definition to enable Code Contracts.  
2 #define CONTRACTS_FULL
```

The Code Contracts library comes with an extension to Visual Studio that enables fine-grained control on how often contracts are checked [27].

2.3.4 Naming conventions

This thesis distinguishes between identifiers in UpperCamelCase form and in lowerCamelCase form. The former always begins with an uppercase letter and the latter always begins with a lowercase letter. In both cases, words are separated by uppercase letters – underscores (`_`) are avoided.

The C# language is case sensitive like Java. The .NET documentation proposes naming conventions for the various language constructs [8]. Table 2.15 lists the most important ones. In general, consecutive uppercase letters are **avoided; for instance, the names ‘Id’ and ‘Xml’ are preferred over ‘ID’ and ‘XML’**. Nevertheless, note that interfaces are prefixed by ‘I’ and type parameters are prefixed by ‘T’.



Table 2.15

Naming conventions in .NET

	Convention	Example
<i>Classes</i>	UpperCamel Case	Examp l e l d
<i>Interfaces</i>	I UpperCamel Case	I Examp l e l d
<i>Enums</i>	UpperCamel Case	Examp l e l d
<i>Structs</i>	UpperCamel Case	Examp l e l d
<i>Methods</i>	UpperCamel Case	Examp l e l d
<i>Properties</i>	UpperCamel Case	Examp l e l d
<i>Constant fields</i>	UpperCamel Case	Examp l e l d
<i>Instance fields</i>	I owerCamel Case	examp l e l d
<i>Local variables</i>	I owerCamel Case	examp l e l d
<i>Type parameters</i>	TUpperCamel Case	TExamp l e l d

Translating identifiers from one language to another may result in conflicts when an identifier from the source language clashes with a reserved word of the destination language. C# works around this problem by its concept of verbatim identifiers. By prefixing an identifier with the at (@) character, it becomes verbatim and may be named like a reserved word without conflicts, as shown in Listing 2.16. The actual name of the identifier is still only the name that follows the at character; **for example, ‘class’ is the actual name of the identifier in ‘@cl ass’.**

Listing 2.16

C#

```

1 // 'class' is a reserved word in C#.
2 public class Alpha
3 {
4     // '@class' is a verbatim identifier.
5     public void Bravo(int @class) { }
6 }

```

The translation rules presented in the following chapters may employ normalisation of identifiers. This means that they are brought into the conventionally correct form, cf. Table 2.15, and made verbatim if necessary.

2.4 Compilers

One of the goals of this thesis is to construct a compiler – more specifically, a transcompiler – from VDM-SL to C#. Definitions 2.17 and 2.18 define a compiler and a transcompiler as follows:

*	<p>Definition 2.17 Compiler</p> <p>A compiler is a piece of software that translates a program from a source language into a semantically equivalent program of a target language.</p>
*	<p>Definition 2.18 Transcompiler</p> <p>A transcompiler, also known as a transpiler or a source-to-source compiler, is a compiler between two languages of approximately the same level of abstraction.</p>

Traditionally, a compiler performs the translation from the source language to the target language in multiple phases [28]: lexical analysis, syntactical analysis, semantic analysis, optimisation and code generation, as shown in Figure 2.19. It is composed of two main components: a frontend and a backend. The frontend performs the analysing phases and deals mostly with the source language. The backend performs optimisation and code generation to the target language.

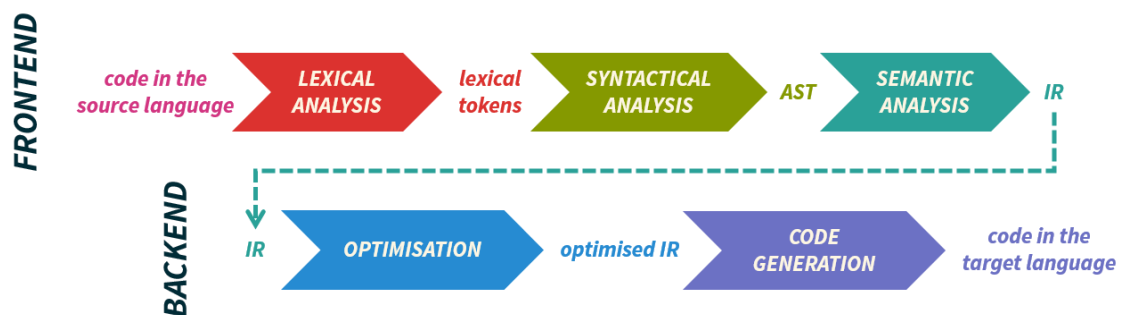


Figure 2.19

The phases of a traditional compiler.

2.4.1 Frontend phases

Lexical analysis transforms the raw text characters of the source code into a sequence of meaningful *tokens* representing the various literals, symbols, reserved words and punctuation of the source language. It is performed by a lexer.²

Syntactical analysis, or parsing, matches the sequence of tokens from the lexer to a context-free grammar of the source language, transforming them into an abstract syntax tree, see Definition 2.20, while determining whether the program is syntactically valid or not. It is performed by a parser.



Definition 2.20

Abstract syntax tree

An abstract syntax tree, abbreviated AST, is a language-specific, syntactical tree representation of a program. Created upon parsing the program, it is a simplification of the derivation tree, also known as the parse tree or concrete syntax tree, produced by matching the program to a context-free grammar of the language.

Semantic analysis constructs a symbol table, also known as an environment, that associates identifiers with information about their types, signatures and usages. Compilers for statically typed source languages like VDM-SL, C# and Java use the symbol table to check that the types of the identifiers are consistent within their context. This particular analysis, called *type checking*, is performed by a type checker. In addition to enforcing static type checking, sophisticated compilers may check other aspects of the source program during the semantic analysis phase, for example by analysing whether the data flow and the control flow of the program obey the rules of the source language. Semantic analysis concludes with the construction of an intermediate representation of the source program, as defined by Definition 2.21:



Definition 2.21

Intermediate representation

An intermediate representation, abbreviated IR, is a language-independent tree representation of a program. It contains information about the program that is neither specific to the source language nor the target language so that it is convenient to construct during semantic analysis and translate during code generation.

² Occasionally, a lexer is also known as a *scanner* or a *tokeniser*.

2.4.2 Backend phases

The optimisation phase transforms the IR tree into a semantically equivalent, yet more optimal, IR tree under a certain notion of optimality such as computational efficiency or resource consumption. It is performed by an optimiser and may involve multiple kinds of optimisations such as DEAD CODE ELIMINATION and CONSTANT FOLDING.

Code generation, or code emission, maps the optimised IR tree into language constructs of the target language and produces the final output of the compiler. It is performed by a code generator.

2.4.3 Code generation in Overture

Overture provides a compiler frontend for parsing and type-checking VDM-SL specifications. It takes a VDM-SL specification in raw text as input and outputs a VDM-SL abstract syntax tree (AST) with type information, if the specification is syntactically and semantically valid. The parser and type-checker in Overture are adaptations of the ones shipped with the VDMJ toolkit [29]. Specifically, they conform to the AST nodes generated by the AstCreator tool as Java classes [30].

The code generation platform of Overture defines an intermediate representation (IR) of VDM-SL specifications – also via AstCreator [12]. It provides an IR generator component, which is an implementation of the VISITOR design pattern [31] that traverses the AST and outputs a corresponding IR tree, thus completing the compiler frontend.

The usual approach to implementing the code generating backend in Overture is to transform the IR tree into another IR tree whose structure resembles the syntax of the target language very closely. Afterwards, the transformed IR tree is translated into source code of the target language via the Apache Velocity template engine [32]. Figure 2.22 shows the code generation process.

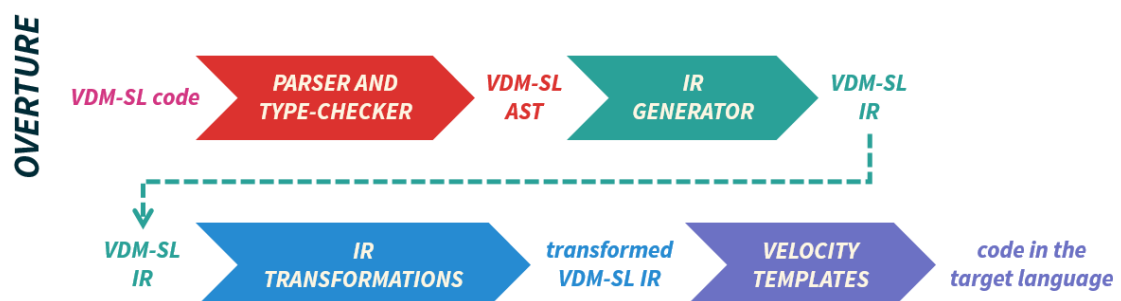


Figure 2.22

The common code generation process in Overture.

The set of necessary IR transformations depends on the code generator, but object-oriented target languages may have transformations in common. The Java code generator, being the pilot code generator, provides many of such transformations, which are also available to other code generators. For example, it provides transformations that replace modules with classes and functions with methods that have inborn return statements.

2.5 Automated testing

Software is of little use if it behaves incorrectly. Such behaviour can be caused by many factors, including wrongly specified software requirements and defective implementations.

Software tests exercise the implementation to ensure that it behaves correctly. The traditional approach to testing is manual testing in which the programmer manually interacts with the software and verifies that it works correctly. While simple to carry out, it becomes tedious when the software system grows since the programmer needs to ensure that modifications of the software has not led to REGRESSIONS anywhere in the software.

An alternative to manual testing is automated testing in which the programmer lets the computer do the software interaction and verification. This is particular useful for regression testing as the testing procedure is automatically repeated and executed by the computer [10].

2.5.1 Unit testing and integration testing

There are many different kinds of automated tests, which tackle the testing procedure differently. For unit tests and integration tests, see Definitions 2.23 and 2.24, the programmer specifies a series of test cases that define the software interactions to carry out and the expected outcome of these interactions [33].



Definition 2.23

Unit test

A unit test is an execution of a single software unit in isolation to observe its behaviour under a predefined input. If the software unit does not behave as expected by the unit test, the test is said to *fail*. Otherwise, it is said to *pass*.



Definition 2.24

Integration test

An integration test is an execution of multiple software units to observe their collaborative behaviour under a predefined input.

A testing framework assists the testing effort by offering an API for specifying test cases as well as a test runner for executing them. A good testing framework gives detailed feedback to the programmer when a test fails so that it is evident to determine what software component caused the failure. Two examples of testing frameworks are JUnit [34] for Java and xUnit.net [35] for C# and other .NET-based languages.

2.5.2 Test-driven development

Test-driven development (TDD) is a software development technique that makes extensive use of unit tests to guide the development. Coined in 2003 by Kent Beck [11], it outlines a development cycle of five steps – the TDD rhythm – meant to increase both developer productivity and the quality of the code produced.

It contains five steps [10]:

1. Pick a feature to implement from the list of requirements.
2. Write a unit test that specifies the expected behaviour of the implementation, run it and observe it fail (as the feature has not been implemented yet).
3. Make a little change in the production code; as small as possible, but just large enough to satisfy the unit test from Step 2.
4. Run the unit test again and observe it pass.
5. Clean up the production code and the test code by refactoring.

Figure 2.25 shows the TDD rhythm as a software development cycle:

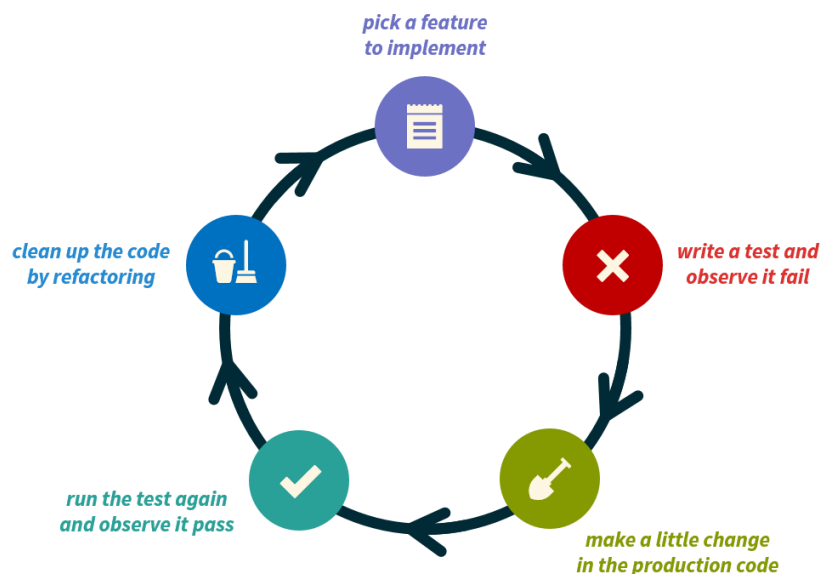


Figure 2.25
The five steps of the TDD rhythm.

“ Software engineering is the part of computer science which is too difficult for the computer scientist.

– Friedrich L. Bauer (1924-2015)

3 Specifications

VDM-SL offers two ways to structure a specification: A flat specification has a single global environment with all definitions, whereas a modular specification spreads the definitions across separate modules. A C# program, on the other hand, consists entirely of classes organised in namespaces.

This chapter presents the translations of VDM-SL specifications to corresponding C# classes. The sections below follow the common structure outlined in Section 1.3.1.



Chapter contents

Section 3.1 proposes two translations of VDM-SL flat specifications to C# classes.

Section 3.2 proposes two analogous translations of VDM-SL modules to C# classes and discusses the treatment of exports and imports in modules.

3.1 Flat specifications

A flat VDM-SL specification consists of blocks in which definitions appear. For example, a function must be located in a `functions` block, and a type must be located in a `types` block. The same goes for operations, states and values. All definitions are declared within a globally scoped environment.



Example 3.1

Flat specifications

Listing 3.1a shows a flat specification in VDM-SL with a type definition `Amount` (lines 2-3), a value definition `VAT` (line 6) and a function definition `ApplyVAT` (lines 9-10) in their respective definition blocks.

Listing 3.1a

VDM-SL

```

1  types
2      Amount = rat
3      inv a == a >= 0;
4
5  values
6      VAT = 0.18;
7
8  functions
9      ApplyVAT: Amount -> Amount
10     ApplyVAT(amount) == (1 + VAT) * amount;

```

In C#, there is no global environment like the one in VDM-SL; all program functionality must be declared in classes. Thus, a flat specification is represented by a class. This section presents two approaches to translating flat specifications in VDM-SL to classes in C#.



Translation proposals

[Section 3.1.1](#) describes the approach of using a static class in C# to represent a flat specification in a traditional, imperative manner.

[Section 3.1.2](#) describes the approach of using a non-static class in C# to represent a flat specification in an object-oriented manner.

3.1.1 Static class

While a flat specification in VDM-SL is unnamed, all class declarations in C# must have a name [3]. In this case, a suitable class **name is ‘Global’**. The definitions in the flat specification become members of the `Global` class by applying the respective translations described in the following chapters. There is no concept of definition blocks in C# so the class members will just appear in the same order as their corresponding VDM-SL definitions.

Note that a member of a C# class is not allowed to have the same name as its enclosing class [3]. This means that no definition in the flat specification may use the name `Global`, as this name is already used by the enclosing class. It is possible to avoid this issue by prefixing the name of the class with an underscore (`_`) character, naming it `_Global` instead of `Global`. This name will not clash with any definition because VDM-SL does not allow a leading underscore in an identifier [7].

However, it is unconventional to prefix a class name in C# by an underscore [8]. Thus, the name ‘Global’ is recommended. The workaround is then to let the transcompiler rename the conflicting definitions when they become class members in C#, for example by appending suffixes such as ‘1’, ‘2’ and so on to their names.

Since Global is just a placeholder for the translated VDM-SL definitions, it must not be instantiable nor inheritable. Therefore, it is marked with the `static` modifier [3]. To make its residing definitions accessible from external software modules, it is also marked with the `public` modifier [3]. Because the class is static, all of its members, except type declarations, must be static as well.

VDM-SL does not restrict a flat specification to a single file. The definitions can be spread across multiple files if necessary. Similarly, C# does not restrict a class declaration to a single file. Its members can be spread across multiple files so long as all parts of the class are marked with the `partial` modifier [3].



Example 3.1, continued

Flat specifications

Listing 3.1a showed a flat specification in VDM-SL with a type definition `Amount`, a value definition `VAT` and a function definition `ApplyVAT`. Listing 3.1b shows the resulting C# class `Global`. It is marked with the `public` and `static` modifiers and contains members for `Amount` (lines 3-7), `Vat` (line 9) and `ApplyVat` (lines 11-16). To save space, the non-essential parts of `Amount` and `ApplyVat` have been omitted in this example. Sections 4.1, 4.5 and 5.6 present the related translations for function, value and type definitions, respectively.

Listing 3.1b

C# translation

```

1 public static class Global
2 {
3     public sealed class Amount
4         : ICopyable<Amount>, IEquatable<Amount>
5     {
6         // Implementation omitted ...
7     }
8
9     public static decimal Vat { get; } = 0.18m;
10
11     [Pure]
12     public static Amount ApplyVat(Amount amount)
13     {
14         // Code omitted ...
15         return (1 + Vat) * amount;
16     }
17 }
```

3.1.2 Non-static class

Using a static class is the traditional, imperative approach. The static class acts as a service provider, making the VDM-SL definitions directly available to external software modules. An alternative, object-oriented approach to a static class is an instantiable class with non-static members.

By letting the `GI obal` class implement an `I GI obal` interface of the definitions exposed by the VDM-SL specification, it fits into the scheme of DEPENDENCY INJECTION [36] [37]. The class must be instantiable to be applicable in dependency injection so it must provide a public constructor. Nevertheless, the constructor has no parameters since the `GI obal` class is self-contained and has no external dependencies. A public, parameterless constructor is called a *default constructor* in C#. It is defined automatically behind the scenes when there are no other constructors [3].

To disallow inheritance, the `GI obal` class is marked with the `sealed` modifier [3]. Note that this does not affect the extensibility of the class: Due to the `I GI obal` interface, its functionality can be enhanced by a structural design pattern such as DECORATOR [37] [31]. Both the `I GI obal` interface and the `GI obal` class are marked with the `public` modifier. If the flat specification is spread across multiple files, the `GI obal` class is also marked with the `partial` modifier.



Example 3.1, continued

Flat specifications

Listing 3.1a showed a flat specification in VDM-SL with a type definition `Amount`, a value definition `VAT` and a function definition `ApplyVAT`. Listing 3.1c shows the resulting `I GI obal` interface (lines 1-6) and matching `GI obal` class (lines 7-23) in C#. They both have members for `Vat` (lines 3 and 15) and `ApplyVat` (lines 4 and 17-22). `GI obal` has an additional type declaration for `Amount` (lines 9-13), as type declarations are not allowed in C# interfaces [3]. Interface members are always public so they have no access modifiers in `I GI obal` [3].

Listing 3.1c

C# translation

```

1 public interface I GI obal
2 {
3     decimal Vat { get; }
4
5     GI obal . Amount ApplyVat(GI obal . Amount amount);
6 }

```

... *Continues on the next page ...*

```

... Continued from the previous page ...

7 public sealed class Global : IGlobal
8 {
9     public sealed class Amount
10        : ICopyable<Amount>, IEquatable<Amount>
11        {
12            // Implementation omitted ...
13        }
14
15     public decimal Vat { get; } = 0.18m;
16
17     [Pure]
18     public Amount ApplyVat(Amount amount)
19     {
20         // Code omitted ...
21         return (1 + Vat) * amount;
22     }
23 }

```

Other software modules that use this class may adhere to the principle of programming to an interface, thereby enabling object-oriented techniques such as design patterns [31] and TEST DOUBLES [38] [33], which are otherwise difficult to achieve with static classes. Even though flat specifications are not meant to be instantiated and should exist as singletons, an object-oriented system may configure a dependency injection container to implement the SINGLETON pattern [31] to ensure that only one instance exists at a time [37].

3.1.3 Rule summarisation

While the second approach with interface implementation and dependency injection is fitting in an object-oriented system, the first approach with static classes is the one that resembles VDM-SL the most. It is therefore selected as the primary translation rule for flat specifications, as summarised by Rule 3.2.



Rule 3.2

Flat specifications

A flat VDM-SL specification becomes a class in C# named 'Global'. It is marked with the `public` and `static` modifiers. If the flat specification is spread across multiple files, it is also marked with the `partial` modifier.

The definitions in the flat specification are translated accordingly and become members of the class in the same order as they appear in the flat specification.



Limitations

Rule 3.2

In C#, a member is not allowed to have the same name as its enclosing class. Therefore, no definitions in the flat specification may use the name ‘Global’ because this name is reserved for the enclosing static class.

The Java code generator in Overture uses a similar approach with static class members [12]. Java does not have static classes so in order to prevent instantiation and inheritance, it marks the class constructor as private [39]. The class is named ‘DEFAULT’ instead of ‘Global’ since this is the name used internally by Overture for flat specifications [18]. The translation of Listing 3.1a to Java is shown in Listing 3.1d. Note that it does not translate the type definition for `Amount`. This behaviour is further explained in Section 5.6.

Listing 3.1d

Java translation

```

1  // @ nullable_by_default
2  final public class DEFAULT implements Serializable
3  {
4      public static final Number VAT = 0.18;
5
6      private Report() { }
7
8      /* @ pure @ */
9      public static Number ApplyVAT(final Number amount)
10     {
11         Number ret_1 = (1L + VAT.doubleValue())
12                        * amount.doubleValue();
13         return ret_1;
14     }
15 }
```

3.2 Modular specifications

With VDM-SL modules, it is possible to split up a specification into separate components. Each module has a name and its own environment of definitions, which is isolated from the other modules, making definitions in a module inaccessible from other modules by default.

To make them accessible, a module may define an interface of definitions to export. It is declared with the `exports` keyword and lists the signatures of the definitions to export.

Other modules cannot use exported definitions right away. They must first import the needed definitions through an interface of imported types. It is declared by the `imports` keyword and lists the names of the imported definitions and the modules that define them. It is possible to expose an imported definition under a different name in the module by renaming it in the interface. After the interfaces of exports and imports and preceded by the `definitions` keyword, the definitions of the module appear in definition blocks as usual.



Example 3.3

Modules

Listing 3.3a shows two VDM-SL modules `VATModule` (lines 1-17) and `ShoppingSystemModule` (lines 19-28). `VATModule` contains the same three definitions (lines 6-15) as the flat specification in Listing 3.1a. The name of `Amount` and `ApplyVAT` are exported (lines 2-3). `ShoppingSystemModule` imports `Amount` (line 20) and `ApplyVAT` (line 21) from `VATModule` and renames them to `Price` and `WithVAT`, respectively. It also defines a function `ComputePrice` (lines 24-25) which uses the renamed, imported definitions.

Listing 3.3a

VDM-SL

```

1  module VATModule
2  exports types Amount
3         functions ApplyVAT: Amount -> Amount
4
5  definitions
6  types
7      Amount = rat
8      inv a == a >= 0;
9
10 values
11     VAT = 0.18;
12
13 functions
14     ApplyVAT: Amount -> Amount
15     ApplyVAT(amount) == (1 + VAT) * amount;
16
17 end VATModule
18
19 module ShoppingSystemModule
20 imports from VATModule types Amount renamed Price
21                        functions ApplyVAT renamed WithVAT
22
23 definitions
24 functions
25     ComputePrice: () -> Price
26     ComputePrice() == WithVAT(34.99);
27
28 end ShoppingSystemModule

```

This section presents two approaches to translating modules in VDM-SL to classes in C#.



Translation proposals

Section 3.2.1 describes the approach of using a static class in C# to represent a module in a traditional, imperative manner.

Section 3.2.2 describes the approach of using a non-static class in C# to represent a module in an object-oriented manner.

3.2.1 Static class

When translating a VDM-SL module to C#, the definitions reside in a class as they do for flat specifications. The C# class that represents a module is named after that module rather than using the default name of ‘Global’. The name is normalised according to the .NET naming conventions for classes, bringing it into UpperCamelCase form [8]. **Note that C#’s naming restriction on class members** still applies: No definition in a module can use the same name as the module, even though this is allowed in VDM-SL [7]. Since modules are not instantiable nor inheritable in VDM-SL, the resulting C# class is marked with the `static` modifier. It is also marked with the `public` modifier.

To ensure that non-exported definitions are inaccessible from other classes, their corresponding C# members are marked with the `private` modifier [3]. The exported definitions remain public.

Public, static class members are accessible right away from another class in the same namespace and do not need to be explicitly imported in C#. Type declarations may be renamed by using `alias` directives in C# [3]. The remaining class members, which are methods and properties, can be renamed by wrapping them in new methods and properties that have the desired names. Note that all underlying definitions are still accessible by their original name in addition to their new names in contrast to VDM-SL, which hides the original names and only exposes definitions under their new names [7].

The `struct` keyword for type exports covers the exposure of the type constructor and type invariants, among others [7]. However, the instance constructor in the corresponding nested C# class cannot be made private as this would prevent instantiations of the class everywhere.

Another solution is to let the nested class implement an empty interface and only expose this interface from the enclosing class instead of exposing the nested class itself. Since the interface is empty, no details about the internal structure of the type are exposed. However, the interface hides any implicit type cast operators defined in the nested class, see Section 5.6.2.



Example 3.3, continued

Modules

Listing 3.3a showed two VDM-SL modules `VATModule` and `ShoppingSystemModule`. Listing 3.3b shows the corresponding C# class for `VATModule`. The `Amount` (lines 5-9) and `Vat` (line 11) members are private, whereas `IAmount` and `ApplyVat` are public (lines 3 and 13-18). The empty `IAmount` interface exposes the name of the `Amount` type from VDM-SL and is used in place of `Amount` in the signature of the `ApplyVat` method (line 14). Instances of `IAmount` are type cast to `Amount` to enable type conversion as explained in Section 5.6.2.

Listing 3.3c shows the C# class for `ShoppingSystemModule`. The `IAmount` type is renamed to `Price` via a `using alias` directive (line 1) and `ApplyVat` is wrapped by the `WithVat` method (lines 14-15). The `ComputePrice` method uses the `Price` alias (lines 6 and 11) and the `WithVat` wrapper method (line 11) instead of the underlying `IAmount` type and `ApplyVat` method. However, the `IAmount` interface hides the implicit type cast operator defined in `Amount`, preventing any type conversions between the `decimal` type and the `Price` alias (line 11). This behaviour is not surprising given that no instances of `Amount` may be created outside `VatModule`.

Listing 3.3b

C# translation

```

1 public static class VatModule
2 {
3     public interface IAmount { }
4
5     private sealed class Amount
6         : IAmount, ICopyable<Amount>, IEquatable<Amount>
7     {
8         // Implementation omitted ...
9     }
10
11     private static decimal Vat { get; } = 0.18m;
12
13     [Pure]
14     public static IAmount ApplyVat(IAmount amount)
15     {
16         // Code omitted ...
17         return (Amount) ((1 + Vat) * (Amount) amount);
18     }
19 }

```

Listing 3.3c

C# translation

```

1  using Price = VatModule.IAmount;
2
3  public static class ShoppingSystemModule
4  {
5      [Pure]
6      public static Price ComputePrice()
7      {
8          // Code omitted ...
9
10         // This line does not compile.
11         return WithVat((Price) 34.99m);
12     }
13
14     private static Price WithVat(Price amount)
15         => VatModule.ApplyVat(amount);
16 }

```

3.2.2 Non-static class

The interface-based approach of flat specifications is also feasible in a modular setting. The interface that is implemented by the class reflects the VDM-SL interface of exported definitions: Only exported definitions should be members of the C# interface.

Because the class is instantiable, it may have a public constructor. Via CONSTRUCTOR INJECTION [37], injected interfaces provide means for exposing imported definitions while preventing access to non-imported definitions. They may even expose the imported definitions under different names. Doing so, however, requires the use of the ADAPTER pattern [31] to connect the exporting interfaces with the importing interfaces, giving rise to boilerplate code in the translated output. Note that the interface-based approach only applies to functions, operations and values. Type definitions are still accessed directly by name or via a `using alias` directive and cannot be hidden in the class that imports them.



Example 3.3, continued

Modules

Listing 3.3a showed two VDM-SL modules `VATModule` and `ShoppingSystemModule`. Listing 3.3d shows the `IVatModule` interface (lines 1-4) and the implementing `VatModule` class (lines 6-24) in C#.

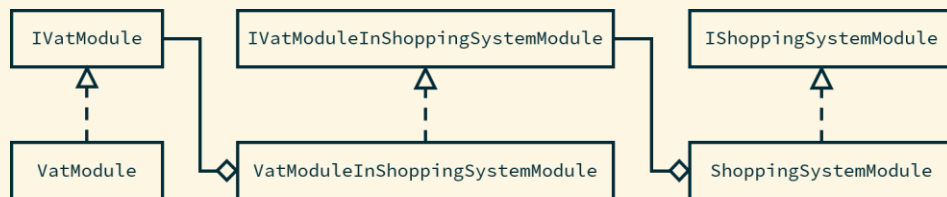
...

Continues on the next page ...

...

Continued from the previous page ...

Listing 3.3e shows the translation of `ShoppingSystemModule` (lines 25-42). Since it does not export its only definition, `ComputePrice`, the `IShoppingSystemModule` interface (line 23) is empty. As it imports and renames two definitions from `VatModule`, it employs a `using alias` directive for renaming `Amount` to `Price` (line 1) and constructor injection for accepting an `Adapter` object to access `ApplyVat` as `WithVat` (lines 27-33). The `Adapter` interface is named `IVatModuleInShoppingSystemModule` to point out its specific role between `VatModule` and `ShoppingSystemModule` (lines 3-6). The concrete implementation `VatModuleInShoppingSystemModule` forwards calls of `WithVat` to `ApplyVat` of a `VatModule` instance (line 20), which it has acquired through constructor injection (lines 11-17).



Listing 3.3d

C# translation

```

1  public interface IVatModule
2  {
3      Amount ApplyVat(Amount amount);
4  }
5
6  public sealed class VatModule : IVatModule
7  {
8      public interface IAmount { }
9
10     private sealed class Amount
11         : IAmount, ICopyable<Amount>, IEquatable<Amount>
12     {
13         // Implementation omitted ...
14     }
15
16     private decimal Vat { get; } = 0.18m;
17
18     [Pure]
19     public IAmount ApplyVat(IAmount amount)
20     {
21         // Code omitted ...
22         return (Amount) ((1 + Vat) * (Amount) amount);
23     }
24 }

```

Listing 3.3e

C# translation

```



1  using Pri ce = VatModul e. I Amount;
2
3  public interface I VatModul eI nShoppi ngSystemModul e
4  {
5      Pri ce Wi thVat(Pri ce amount);
6  }
7
8  public class VatModul eI nShoppi ngSystemModul e
9      : I VatModul eI nShoppi ngSystemModul e
10 {
11     private I VatModul e vatModul e;
12
13     public VatModul eI nShoppi ngSystemModul e(
14         I VatModul e vatModul e)
15     {
16         thi s.vatModul e = vatModul e;
17     }
18
19     public Pri ce Wi thVat(Pri ce amount)
20         => vatModul e. Appl yVat(amount);
21 }
22
23 public interface I Shoppi ngSystemModul e { }
24
25 public class Shoppi ngSystemModul e : I Shoppi ngSystemModul e
26 {
27     private I VatModul eI nShoppi ngSystemModul e vatModul e;
28
29     public Shoppi ngSystemModul e(
30         I VatModul eI nShoppi ngSystemModul e vatModul e)
31     {
32         thi s.vatModul e = vatModul e;
33     }
34
35     private Pri ce ComputePri ce()
36     {
37         // Code omi tted ...
38
39         // Thi s l i ne does not compi l e.
40         return vatModul e. Wi thVat((Pri ce) 34.99m);
41     }
42 }

```

3.2.3 Rule summarisation

Though the second approach enables object-oriented principles to a greater degree than the first one, it forces the translation to output lots of boilerplate code for constructor injection and the Adapter design pattern. To stay in line with the translation of flat specifications, the first approach with static classes is selected.

Because neither `using` aliases nor wrapper members hide the underlying definitions, they are omitted from the translation so that imported definitions will always use their declaration name. This also implies that an exported type definition always includes details about its internal structure. Rule 3.4 summarises the translation of modules by this approach.

	<div> <div>Rule 3.4</div> <div>Modules</div> <p>A VDM-SL module becomes a class in C# with the normalised name of the module. It is marked with the <code>public</code> and <code>static</code> modifiers.</p> <p>The definitions in the module are translated accordingly and become members of the class in the same order as they appear in the module. Non-exported definitions are marked with the <code>private</code> modifier, whereas exported definitions are marked with the <code>public</code> modifier.</p> </div>
	<div> <div>Limitations</div> <div>Rule 3.4</div> <p>No definitions may use the UpperCamelCase-normalised name of their enclosing module due to the naming restrictions on class members in C#. Likewise, no other modules may have a name that results in the same normalisation.</p> <p>Exported definitions are accessible from everywhere even if they have not been imported in the VDM-SL module. They are always accessed by their declaration name and are not renamed.</p> </div>

Like flat specifications, the Java code generator in Overture translates modules to classes with static members and a private constructor [12]. It names the class after the module, but it does not normalise the name according to the naming conventions for classes in Java. All members are marked with the `public` access modifier, even if they are not exported. Moreover, it completely refuses to translate modules that rename imported definitions because it does not support this particular translation and therefore gives up immediately to avoid outputting invalid code. The translation of Listing 3.3a to Java is shown in Listing 3.3f, except for the renamed definitions within `ShoppingSystemModule`.

Listing 3.3f

Java translation

```

1  // @ nullable_by_default
2  final public class VATModule implements Serializable
3  {
4      public static final Number VAT = 0.18;
5
6      private VATModule() { }
7
8      /* @ pure */
9      public static Number ApplyVAT(final Number amount)
10     {
11         Number ret_2 = (1L + VAT.doubleValue())
12                        * amount.doubleValue();
13         return ret_2;
14     }
15
16     // Code omitted ...
17 }
18
19 // @ nullable_by_default
20 final public class ShoppingSystemModule
21     implements Serializable
22 {
23     private ShoppingSystemModule() { }
24
25     /* @ pure */
26     public static Number ComputePrice()
27     {
28         Number ret_3 = VATModule.ApplyVAT(34.99);
29         return ret_3;
30     }
31
32     // Code omitted ...
33 }

```


“ The most important property of a program is whether it accomplishes the intention of its user.

– Tony Hoare (1934-)

4 Functionality

VDM-SL has three different ways of describing units of functionality: functions, pure operations and impure operations. Functions and pure operations are referentially transparent, whereas impure operations are referentially opaque. All three constructs are allowed to specify pre- and postconditions to guard their behaviour. Besides functions and operations, VDM-SL values define constants and states define persistent information stored by a model. The consistency of the information can be enforced by state invariants. Being a class-based object-oriented language, C# encapsulates most functionality in methods and properties, which are members of classes. Persistent information – that is, the program state – is stored in fields.

This chapter presents the translations of VDM-SL functionality to corresponding C# constructs. The sections below follow the common structure outlined in Section 1.3.1.



Chapter contents

Section 4.1 translates VDM-SL functions and operations to C# methods.

Section 4.2 translates VDM-SL preconditions and postconditions to invocations of the `Contract.Requires` and `Contract.Ensures` methods of the .NET Code Contracts library.

Section 4.3 translates VDM-SL states to C# properties.

Section 4.4 translates VDM-SL state invariants to `Contract.Invariant`. It also describes the translation of multiple assignments in atomic statements.

Section 4.5 translates VDM-SL values to C# constant fields and read-only properties.

4.1 Functions and operations

In VDM-SL, functions and pure operations are referentially transparent, while impure operations are referentially opaque. This section presents the translations of functions and operations to methods in C#.



Translation proposals

Section 4.1.1 describes a single approach to translating impure operations in VDM-SL to ordinary methods in C#.

Section 4.1.2 describes a single approach to translating functions and pure operations in VDM-SL to pure methods in C#.

Section 4.1.3 describes a single approach to translating implicit functions and operations in VDM-SL to methods in C# that throw exceptions.

4.1.1 Impure operations

An impure operation is defined in the `operations` block. It is referentially opaque, may both access and modify the state, may call all kinds of functions and operations and is allowed to have side effects. It is not required to produce a result value, but may do so via the `return` keyword.



Example 4.1

Impure operations

Listing 4.1a shows the definitions of two impure operations `Greet` (lines 2-3) and `UpdatePrice` (lines 5-6). The former prints a text string in the console output by calling the externally defined `IO`println` operation (line 3), whereas the latter assigns a new value to the `price` field of the state (line 6). None of them produce a result value.

Listing 4.1a

VDM-SL

```

1  operations
2    Greet: () ==> ()
3    Greet() == IO`println("What's up?");
4
5    UpdatePrice: Price ==> ()
6    UpdatePrice(newPrice) == price := newPrice

```

An impure operation in VDM-SL is equivalent to an ordinary method in C#, which is also permitted to access and modify program state and call other methods. The method is named after the operation and gets an equivalent signature. The name is normalised according to the .NET naming conventions for methods, bringing it into UpperCamelCase form if necessary [8]. Note that no two members of a C# class may have the same name, except for overloaded methods [3]. This implies that no other definitions in the enclosing specification may have a normalised name that clashes with the normalised name of the impure operation, except for overloaded operations. The parameter types and the return type are translated accordingly, using the guidelines presented in Chapter 5. Special care is taken for operations that do not produce a result value: They are equivalent to methods with the `void` return type in C#.

The method is marked with the `public` access modifier if it originates from a flat specification or is exported in a module. Otherwise, it is marked with the `private` access modifier. Since both flat specifications and modules become static classes, it is obliged to have the `static` modifier as well. The body of the operation becomes the body of the method; usually as an ordinary code block in curly braces, but whenever possible, it is converted to an expression-body instead.



Example 4.1, continued

Impure operations

Listing 4.1a showed two impure operations `Greet` and `UpdatePrice` with no result values in VDM-SL. Listing 4.1b shows the corresponding C# methods with the `void` return type. `Greet` appears as an expression-body (lines 1-2), while the body of `UpdatePrice` appears in ordinary curly braces (lines 4-9) because it consists of multiple statements. Here, only the last statement is shown – the other statement, an inferred precondition for type consistency of `newPrice`, has been omitted.

Listing 4.1b

C# translation

```

1 public static void Greet()
2     => Console.WriteLine("What's up?");
3
4 public static void UpdatePrice(Pri ce newPri ce)
5 {
6     // Code omitted ...
7     State.Pri ce = newPri ce;
8 }
```

Rule 4.2 summarises the translation of impure operations in VDM-SL to methods in C#.



Rule 4.2

Impure operations

An impure operation in VDM-SL becomes a method in C# with the name of the operation normalised to UpperCamelCase.

If it is defined in a flat specification or is exported in a module, it is marked with the `public` access modifier. Otherwise, it is marked with the `private` access modifier. Additionally, it is always marked with the `static` modifier.

The method parameters appear in the same order as in the operation signature and they have corresponding names in lowerCamelCase. The types of the parameters and the result are translated accordingly. If the operation does not produce a result value, the return type is `void`.

The operation body is translated to a method body. Whenever possible, it is converted to an expression-body instead of a code block in curly braces.



Limitations

Rule 4.2

The name normalisation may result in conflicts with other definitions of the same specification. Method parameters can be reassigned in C#.

The Java code generator in Overture also translates impure operations to static methods in Java, arranging the return type and parameters in a similar manner. Additionally, it marks all parameters with the `final` modifier of Java, making them immutable like the parameters of VDM-SL operations. C# does not have an equivalent modifier for achieving this. The translation of Listing 4.1a to Java is shown in Listing 4.1c.

Listing 4.1c

Java translation

```
1 public static void Greet()  
2 {  
3     IO.println("What's up?");  
4 }  
5  
6 public static void UpdatePrice(final Number newPrice)  
7 {  
8     // Code omitted ...  
9     St.set_price(newPrice);  
10 }
```

4.1.2 Functions and pure operations

Like an impure operation, a pure operation is defined in the `operations` block. On the other hand, a function is defined in the `functions` block. Functions and pure operations are referentially transparent and must produce a result value. Both constructs are allowed to call other functions and pure operations. Furthermore, functions can be parametrically polymorphic and pure operations are allowed to read the fields of the state directly.



Example 4.3

Functions and pure operations

Listing 4.3a shows two functions `Square` (lines 2-3) and `Identity` (lines 5-6) and a pure operation `GetPrice` (lines 9-10) marked with the `pure` modifier. `Identity` is parametrically polymorphic and defines the type variable `@input` in its signature (line 5).

Listing 4.3a

VDM-SL

```

1  functions
2      Square: int -> int
3      Square(number) == number * number;
4
5      Identity[@input]: @input -> @input
6      Identity(i) == i;
7
8  operations
9      pure GetPrice: () ==> Price
10     GetPrice() == return price;
```

The C# language does not distinguish between referentially transparent methods and referentially opaque ones. Thus, VDM-SL functions and pure operations are translated to methods in the same way as impure operations, described in Section 4.1.1, with the addition that type variables of polymorphic VDM-SL functions become generic type parameters of the C# method. Their **names are prefixed by ‘T’ in UpperCamelCase form to follow the .NET naming conventions for type parameters** [8].

Nevertheless, the Code Contracts library of .NET does make this distinction as it requires all contract predicates to exhibit some degree of purity and provides the `PureAttribute` type for this purpose [9]. This attribute informs the Code Contracts library that a method is referentially transparent and therefore suitable as a contract predicate. All C# methods that originate from functions and pure operations are tagged with `[Pure]`.

In .NET terminology, they become *pure methods*. They are equivalent to pure operations in VDM-SL, since they are allowed to access the program state, but not modify it. Note that `PureAttribute` is only an indicator for referential transparency, not a guarantee. Neither the C# compiler nor the .NET runtime environment enforces it [9].



Example 4.3, continued

Functions and pure operations

Listing 4.3a showed two functions `Square` and `Identity` and a pure operation `GetPrice`. Listing 4.3b shows the corresponding C# methods tagged with `[Pure]`. The `Identity` method has a type parameter `TInput` (line 5) originating from the `@input` type variable of the VDM-SL function. An inferred postcondition for type consistency of the return value in `GetPrice` has been omitted.

Listing 4.3b

C# translation

```

1  [Pure]
2  public static int Square(int number) => number * number;
3
4  [Pure]
5  public static TInput Identity<TInput>(TInput i) => i;
6
7  [Pure]
8  public static Price GetPrice()
9  {
10     // Code omitted ...
11     return State.Price;
12 }
```

Rule 4.4 summarises the translation of functions in VDM-SL to pure methods in C#.



Rule 4.4

Functions

A function in VDM-SL becomes a method in C# following Rule 4.2. Furthermore, the method is tagged with `[Pure]`. Type variables of the function become generic type parameters of the method prefixed by **“T” in UpperCamelCase**.





Limitations

Rule 4.4

Same limitations as Rule 4.2. Pure methods in C# are allowed to read the state directly, which is forbidden for functions in VDM-SL. Referential transparency is not enforced.

Rule 4.5 summarises the translation of pure operations in VDM-SL to pure methods in C#.

	<div>Rule 4.5</div> <div>Pure operations</div> <p>A pure operation in VDM-SL becomes a method in C# following Rule 4.2. Furthermore, the method is tagged with [Pure].</p>
	<div>Limitations</div> <div>Rule 4.5</div> <p>Same limitations as Rule 4.2. Referential transparency is not enforced.</p>

Like C#, Java does not distinguish between referentially transparent and opaque methods. The Java code generator in Overture translates pure operations and functions to static methods. Only functions are annotated with the pure modifier of JML [13], which serves the same purpose as the `PureAttribute` type of .NET. However, the JML tools enforce referential transparency of pure methods in contrast to .NET [23]. The translation of Listing 4.3a to Java is shown in Listing 4.3c.

	<div>Listing 4.3c</div> <div>Java translation</div> <pre> 1 /*@ pure @*/ 2 public static Number Square(final Number number) 3 { 4 Number ret_60 = number.LongValue() * number.LongValue(); 5 return ret_60; 6 } 7 8 /*@ pure @*/ 9 public static <input> input Identity(final input i) 10 { 11 input ret_61 = i; 12 return ret_61; 13 } 14 15 public static Number GetPrice() 16 { 17 // Code omitted ... 18 Number ret_59 = St.getPrice(); 19 return ret_59; 20 }</pre>
--	--

4.1.3 Implicit functions and operations

Functions and operations do not necessarily have to define explicit bodies. As VDM-SL is a modelling language, it is possible to define an implicit function or operation that only describes its nature through pre- and postconditions without specifying the exact computation that lies behind. This feature complements the `is not yet specified` construct, which is a placeholder for an explicit function or operation body that has yet to be defined [7].

In general, implicit functions and operations are not executable – they are meant for modelling and software verification. If such a function or operation is invoked anyway, the VDM-SL interpreter in Overture throws an exception by default. However, Lausdahl et al. [40] have carried out experiments with the ProB model checker to enable interpretation of a subset of implicit functions and operations by constraint solving.



Example 4.6

Implicit functions

Listing 4.6a shows an implicit function `SquareRoot` (lines 2-4) that is guarded by a pre- and a postcondition. Note that the syntax is slightly different from explicit functions. In particular, the result value is explicitly named ‘`r`’ (line 2) instead of being referred to by the `RESULT` keyword in the postcondition.

Listing 4.6a

VDM-SL

```
1 functions
2   SquareRoot(n: real) r: real
3   pre n >= 0
4   post r * r = n and r >= 0;
```

.NET provides the `NotImplementedException` type, which is defined in the `System` namespace, for a similar purpose. While the signatures of implicit functions and operations can be translated to method signatures along with their pre- and postconditions, the method bodies will throw this exception.



Example 4.6, continued

Implicit functions

Listing 4.6a showed an implicit function `SquareRoot`. Listing 4.6b shows the corresponding C# method that throws an instance of `NotImplementedException` (line 5). The translations of the pre- and postconditions have been omitted.

Listing 4.6b

C# translation

```
1 [Pure]
2 public static decimal SquareRoot(decimal n)
3 {
4     // Code omitted ...
5     throw new NotImplementedException();
6 }
```

Rule 4.7 summarises the translation of implicit functions and operations in VDM-SL to methods in C# that throw an exception.



Rule 4.7

Implicit functions and operations

An implicit function or operation in VDM-SL becomes a method in C# by their respective translations described in Rules 4.2 to 4.4. Furthermore, the method throws an instance of `NotImplementedException`.

The Java code generator in Overture uses a similar approach – it throws an instance of `UnsupportedOperationException` for implicit functions and operations as well as the `is not yet specified` construct. The translation of Listing 4.6a to Java is shown in Listing 4.6c, except for the pre- and postconditions.

Listing 4.6c

Java translation

```
1 // Code omitted ...
2 /*@ pure @*/
3 public static Number SquareRoot(final Number n)
4 {
5     throw new UnsupportedOperationException();
6 }
```

4.2 Pre- and postconditions

Function and operations may be guarded by pre- and postconditions, which are logical predicates that must hold upon entrance and exit, respectively. This section presents their C# translations using the .NET Code Contracts library [9].

In .NET Code Contracts, preconditions are specified by calling the `Contract.Requires` method in the beginning of the method being protected. Postconditions are specified similarly by calling `Contract.Ensures`. Both methods take a single predicate as input and throws an instance of `ContractException` if the predicate evaluates to false when the method is entered and exited. Because VDM-SL automatically treats the logical predicates of pre- and postconditions as functions [7], they become pure methods in C# by Rule 4.4. Therefore, Code Contracts permits invoking these methods from `Contract.Requires` and `Contract.Ensures` to guarantee that the pre- and postconditions have been satisfied upon entrance and exit.

For postconditions, it is also possible to refer to the return value by calling `Contract.Result<TReturn>`, where `TReturn` is the return type, and the pre-state by calling `Contract.OldValue`, which takes an expression as input and returns the value of this expression at the time of method entrance.



Example 4.6, continued

Pre- and postconditions

Listing 4.6a showed an implicit function `SquareRoot` with a pre- and a postcondition. Listing 4.6d shows the corresponding `SquareRoot` method in C# with its predicates `PreSquareRoot` (lines 11-12) and `PostSquareRoot` (lines 14-16).

The precondition is enforced by calling `Contract.Requires` in the beginning of the `SquareRoot` method (line 4). The postcondition is enforced by calling `Contract.Ensures` immediately after (lines 5-6). It uses `Contract.Result` to retrieve the return value for the postcondition.

Listing 4.6d

C# translation

```

1  [Pure]
2  public static decimal SquareRoot(decimal n)
3  {
4      Contract.Requires(PreSquareRoot(n));
5      Contract.Ensures(
6          PostSquareRoot(n, Contract.Result<decimal>()));
7
8      throw new NotImplementedException();
9  }
10
11 [Pure]
12 public static bool PreSquareRoot(decimal n) => n >= 0;
13
14 [Pure]
15 public static bool PostSquareRoot(decimal n, decimal r)
16     => r * r == n && r >= 0;

```

Rule 4.8 summarises the translation of preconditions in VDM-SL to .NET Code Contracts.



Rule 4.8

Preconditions

The predicate of a precondition becomes a pure method by Rule 4.4. It is enforced by calling `Contract.Requires` in the beginning of its host method.

Rule 4.9 summarises the translation of postconditions in VDM-SL to .NET Code Contracts.



Rule 4.9

Postconditions

The predicate of a postcondition becomes a pure method by Rule 4.4. It is enforced by calling `Contract.Ensures` in the beginning of its host method, immediately after any calls to `Contract.Requires`.

References to the return value becomes `Contract.Result<TReturn>` where `TReturn` is the translated return type of the host method. Within the postcondition **predicate method, this parameter is named 'result'**.

References to the pre-state are wrapped into `Contract.OldValue`. Within the postcondition predicate method, this parameter is named after the state and **prefixed with 'old' in lowerCamelCase**.

JML supports pre- and postconditions through the `requires` and `ensures` annotations, which are applied by the Java code generator in Overture [13]. JML also offers the `\result` and `\old` annotations for retrieving the return value and pre-state, respectively. Unlike `Contract.Result<TReturn>`, the `\result` annotation infers the return type automatically. Listing 4.6e shows the Java translation of the `SquareRoot` function in Listing 4.6a.

Listing 4.6e

Java translation

```
1 //@ requires pre_SquareRoot(n);
2 //@ ensures post_SquareRoot(n, \result);
3 /*@ pure @*/
4 public static Number SquareRoot(final Number n)
5 {
6     throw new UnsupportedOperationException();
7 }
```

4.3 States

A state defines a collection of fields to persist information in the model, corresponding to a composite type definition, see Section 5.5. The fields can be read and updated at any time from an operation within their scope, cf. Table 2.3. The state is defined in the `state` block.



Example 4.10

States

Listing 4.10a shows a definition of a state named `St` (lines 1-6) with two fields `x` (line 2) and `y` (line 3), both of type `int`. The invariant in the `inv` clause (line 4) proclaims that `x` must be less than `y`. From the `init` clause (line 5), `x` is initialised to the value of 1 and `y` is initialised to the value of 2.

Listing 4.10a

VDM-SL

```

1  state St of
2      x: int
3      y: int
4      inv s == s. x < s. y
5      init s == s = mk_St(1, 2)
6  end

```

C# provides similar language constructs which are also called fields. They are divided into two groups: instance fields and static fields. Instance fields store information of a single object, whereas static fields contain class-level information. Fields are members of classes [3].

It is often beneficial to encapsulate C# fields in *getter* and *setter* methods since this allows the class to track accesses and modifications of the state easily. State encapsulation is indeed built into the C# language through the concept of properties [3]. An auto-property implicitly defines a field – a so-called *backing field* – as well as getter and setter methods in a single line of code. This section presents two approaches to translating states in VDM-SL to properties in C#.



Translation proposals

[Section 4.3.1](#) describes the approach of using a single property in C# to represent the state as a whole.

[Section 4.3.2](#) describes the approach of using multiple properties in C# to represent the state as a collection of separate fields.

4.3.1 Single common property

Because VDM-SL organises all fields of the state in a single state definition, which is denotable by a name and an underlying composite type, the resulting C# class needs two members: a nested class declaration for the composite type, see Section 5.5 for details, and an auto-property to represent the state definition. Since two members of a C# class cannot share the same name, one of them must be named differently. Naming the auto-property **'State'** is suitable although no other definitions may use this name then. The state fields are accessible via the object contained in the property. The state invariant is handled by the nested class declaration, see Section 4.4.

As flat specifications and modules become static classes, the property becomes a static member of the enclosing class, indicated by the `static` modifier. It is also marked with the `private` access modifier because a state is always private to its flat specification or module [7]. The expression in the `init` clause of the state is translated to an expression in C# for initialising the backing field of the property.



Example 4.10, continued

States

Listing 4.10a showed a state `St` with two fields `x` and `y`. Listing 4.10b shows the corresponding auto-property `State` of type `St` in C# (line 1). The values of the state fields are initialised immediately by instantiating `St` in the property initialiser. The translations of the invariant and the state type `St` to a nested class declaration are omitted, see Sections 4.4 and 5.5 for details.

Listing 4.10b

C# translation

```
1 private static St State { get; set; } = new St(1, 2);
```

This straightforward approach faces a minor issue, though. When an operation in VDM-SL accesses a field, it does so by referring directly to the name of the field. In the C# class, all fields are contained in a single property. Therefore, when a method needs to access a field, it must qualify the name of the field with the name of the C# property – **'State'** – using the dot-notation as shown in Listings 4.1b (line 7) and 4.3b (line 11).

4.3.2 Multiple separate properties

The qualification issue can be avoided by splitting the single static property `State` into multiple static properties, each representing one state field. Now, a method can access the state field directly through its separate property. This also eliminates the need of translating the underlying composite type of the state to a nested class declaration.



Example 4.10, continued

States

Listing 4.10a showed a state `St` with two fields `x` and `y`. Listing 4.10c shows the corresponding auto-properties `X` (line 1) and `Y` (line 3).

Listing 4.10c

C# translation

```
1 private static int X { get; set; } = 1;  
2  
3 private static int Y { get; set; } = 2;
```

This approach turns out to be a double-edged sword, though, when the state is protected by an invariant. The function that represents the logical predicate in the invariant takes only a single parameter as input: the state as a whole. Therefore, the corresponding pure method in C# cannot blindly follow Rule 4.4, but needs special treatment to split the single argument for the state into multiple arguments for the separate fields.

The greatest disadvantage of this approach, however, is its incompatibility with the .NET Code Contracts library. As explained in the next section, invariants in Code Contracts are only capable of protecting non-static fields, but the backing fields of the properties in this approach are static. In the first approach in Section 4.3.1, the backing fields are non-static due to the nested class declaration.

4.3.3 Rule summarisation

The first approach that represents the state by a single auto-property and a nested class declaration does not face compatibility issues with Code Contracts. It also resembles VDM-SL the most by treating the state as a whole. It is therefore chosen as the translation of states, as summarised by Rule 4.11.



Rule 4.11

States

A state becomes an auto-property named **'State'** marked with the `private` and `static` modifiers. The underlying composite type and state invariant becomes a nested class declaration by Rule 5.16. The `init` clause of the state becomes an initialising expression of the `State` property.



Limitations

Rule 4.11

The name **'State'** is reserved for the auto-property.

Java does not have a concept of properties [41]; hence, the Java code generator in Overture uses a single `private`, `static` field to represent the state and an external class to represent the underlying composite type. Since Java does not have the same name restriction on class members as C#, the field is named directly after the state definition. The translation of Listing 4.10a to Java is shown in Listing 4.10d, except for the external class declaration that corresponds to the composite type as well as the state invariant defined therein.

Listing 4.10d

Java translation

```
1  /*@ spec_public @*/
2  private static St St = new St(1L, 2L);
```

4.4 Invariants

An invariant proclaims a condition on a piece of information, for example the state, that must always hold at any point in time. This section presents its C# translation using the .NET Code Contracts library [9].

An invariant in C# is specified by calling the `Contract.Invariant` method of the Code Contracts library. Like `Contract.Requires` and `Contract.Ensures`, it takes a predicate as input. Because VDM-SL automatically treats the logical predicates of invariants as functions, they become pure methods in C# by Rule 4.4 and thus suitable for contractual use.

Since invariants protect the state of the program rather than the input and output of methods, they are treated a bit differently in .NET Code Contracts than pre- and postconditions. All calls to `Contract.Invariant` must reside in a special helper method that returns `void` and is tagged with `[ContractInvariantMethod]`, which is an attribute provided by the Code Contracts library. This helper method is specific to the resulting C# class and is not defined in VDM-SL like the predicates for invariants. By convention, it is named ‘ObjectInvariant’ [9], thus disqualifying this name among the other definitions.



Example 4.10, continued

Invariants

Listing 4.10a showed a state `St` with two fields `x` and `y` protected by an invariant that proclaimed `x` to be less than `y`. Listing 4.10e shows the corresponding nested class declaration `St` with two auto-properties `X` (line 4) and `Y` (line 6) representing the state fields along with the public constructor (line 8-12) for initialising the backing fields of `X` and `Y` and the `ObjectInvariant` method (lines 14-16) and predicate method `InvSt` (lines 18-23) representing the state invariant.

Listing 4.10e

C# translation

```

1  [Serializable]
2  public sealed class St : ICopyable<St>, IEquatable<St>
3  {
4      public int X { get; set; }
5
6      public int Y { get; set; }
7
8      public St(int x, int y)
9      {
10         X = x;
11         Y = y;
12     }
13
14     [ContractInvariantMethod]
15     private void ObjectInvariant()
16         => Contract.Invariant(InvSt(this));
17
18     [Pure]
19     public static bool InvSt(St s)
20     {
21         Contract.Requires(s != null);
22         return s.X < s.Y;
23     }
24
25     // Code omitted ...
26 }
```


Invariants in .NET Code Contracts are only checked upon leaving any public method [9] [27] in contrast to VDM-SL which states that invariants must hold in every single program point. An invariant that has been satisfied so far can only be violated by manipulating the program state it is protecting. Therefore, in practice, it is only necessary to enforce invariants after program state mutations. By encapsulating all program state – for example the state fields in the nested class – in properties, the Code Contracts library will check the invariants after invoking a public property setter, as this is equivalent to calling a public setter method. When instantiating a complex value like a state or a record from a composite type, the invariants are also checked after calling the public constructor. Rule 4.12 summarises the translation of invariants in VDM-SL to .NET Code Contracts.



Rule 4.12

Invariants

The predicate of an invariant becomes a pure method by Rule 4.4. It is enforced by calling `Contract.Invariant` within the special `ObjectInvariant` helper method of its host class.

JML supports invariants through the `invariant` annotation, which is applied by the Java code generator in Overture [13]. Unlike invariants in .NET Code Contracts, invariants in JML are checked both upon entrance and exit of methods that are not annotated with the `helper` modifier [23]. Listing 4.10f shows the Java translation of the `St` state in Listing 4.10a.

Listing 4.10f

Java translation

```

1  final public class St implements Record, java.io.Serializable
2  {
3      public Number x;
4      public Number y;
5
6      //@ public instance invariant invChecksOn ==>
7          inv_St(x, y);
8      public St(final Number _x, final Number _y)
9      {
10         x = _x;
11         y = _y;
12     }

```

... Continues on the next page ...

... Continued from the previous page ...

```

12      /*@ pure @*/
13      /*@ helper @*/
14      public static Boolean Inv_St(final Number _x,
15                                  final Number _y)
16      {
17          return _x.LongValue() < _y.LongValue();
18      }
19  }

```

4.4.1 Multiple assignments

VDM-SL offers a multiple assignments construct via the `atomic` keyword, which disables invariant checking temporarily to allow assignments of interdependent variables without violating their invariant [7]. The Code Contracts library does not offer a direct way of turning off invariants temporarily, but the same outcome can be achieved by slightly changing the predicate input to `Contract.Invariant` using the logical operation of material conditional:

Let *invariantsOn* be a logical statement and *invariant* be the invariant predicate to turn on and off. Then the predicate

$$invariantsOn \Rightarrow invariant$$

always holds when *invariantsOn* is false or when *invariant* is true. Therefore, it is possible to disable invariant checking by setting *invariantsOn* to false and enable invariant checking by setting *invariantsOn* to true.

C# has no *implies* (\Rightarrow) operator like VDM-SL, but it is possible to rewrite it as a logically equivalent expression using the ordinary *not* (\neg) and *or* (\vee) operators:

$$\begin{aligned}
 invariantsOn &\Rightarrow invariant \\
 &\Downarrow \\
 \neg invariantsOn &\vee invariant
 \end{aligned}$$

Listing 4.13 shows a quick example of this rewriting in action where `AreInvariantsEnabled` is a static auto-property of type `bool` that corresponds to *invariantsOn*.

Listing 4.13

C#

```

1  Contract.Invariant(!AreInvariantsEnabled || InvSt(this));

```

It is conditionally compiled via the `CONTRACTS_FULL` symbol so that it only appears in the CIL bytecode when code contracts are enabled.


JML has a native *implies* operator (`==>`), which is used by the Java code generator as shown in Listing 4.10f (line 6). Here, `invChecksOn` is a so-called ghost variable that corresponds to *invariantsOn*. In JML, ghost variables are variables that are meant specifically for contracts and not for execution [23]. Therefore, they are only visible to the JML tool. They are declared through the ghost modifier as shown in Listing 4.14:

Listing 4.14 Java

```
1 /*@ public ghost static boolean invChecksOn = true; @*/
```

4.5 Values

A value in VDM-SL defines a named constant that can be accessed from anywhere in the specification. It is defined in the `values` block.


 **Example 4.15** Values

Listing 4.15a shows two values `VAT` (line 2) and `ItemForSale` (line 3) whose types are inferred by the VDM-SL type-checker from their initialising expressions.

Listing 4.15a VDM-SL

```
1 values
2   VAT = 0.18;
3   ItemForSale = mk_Item("A hat", 34.99);
```

This section presents two approaches to translating values in VDM-SL to corresponding constructs in C#.

 **Translation proposals**

[Section 4.5.1](#) describes the approach of using a constant field in C# to represent a value.

[Section 4.5.2](#) describes the approach of using a read-only property in C# to represent a value.

4.5.1 Constant fields

C# offers a similar feature to values – constant fields – via the `const` modifier, which defines a static, compile-time constant [3]. A reference to a constant field is replaced directly by its value and is consequently subject to constant folding upon compilation [42].

However, a value definition in VDM-SL may contain any kind of value, including an instance of a composite type or a collection type, which become a reference type in C# as explained in Chapter 5. The values of reference types, except for string literals, cannot be resolved on compile-time, making the C# compiler refuse to compile such constant fields.



Example 4.15, continued

Values

Listing 4.15a showed two values `VAT` and `ItemForSale` in VDM-SL. Listing 4.15b shows the corresponding C# constant fields. `Item` is the translation of a composite type of the same name, see Section 5.5, and is a reference type. The `Item` instance cannot be resolved on compile-time, making the constant field `ItemForSale` illegal (line 4).

Listing 4.15b

C# translation

```
1 public const decimal Vat = 0.18m;
2
3 // This constant field does not compile.
4 public const Item ItemForSale = new Item("A hat", 34.99m);
```

4.5.2 Read-only properties

A more versatile approach is to use a static, read-only auto-property. It is initialised when the program begins and retains this initial value during the entire lifetime of the program.

The property is marked with the `public` access modifier if it originates from a flat specification or is exported in a module. Otherwise, it is marked with the `private` access modifier. It is also marked with the `static` modifier. It is named after the value definition, bringing the name into UpperCamelCase form which is the naming convention for properties in C# [8]. To make it read-only, the property setter is omitted, leaving only the property getter.



Example 4.15, continued

Values

Listing 4.15a showed two values VAT and ItemForSale in VDM-SL. Listing 4.15c shows the corresponding read-only auto-properties in C#.

Listing 4.15c

C# translation

```

1 public static decimal Vat { get; } = 0.18m;
2
3 public static Item ItemForSale { get; }
4   = new Item("A hat", 34.99m);

```

4.5.3 Rule summarisation

Since constant fields do not support all kinds of values, the second approach with read-only properties is chosen as the translation of VDM-SL values, as summarised by Rule 4.16.



Rule 4.16

Values

A value in VDM-SL becomes a read-only auto-property in C# with the name of the value normalised to UpperCamelCase.

If it is defined in a flat specification or is exported in a module, it is marked with the `public` access modifier. Otherwise, it is marked with the `private` access modifier. Additionally, it is always marked with the `static` modifier.

The value initialiser is translated to a C# expression and used to initialise the property.

Due to the lack of properties in Java [41], the Java code generator in Over-ture resorts to using `public, static, final` fields to represent values. It is equivalent to declaring a static field in C# marked with the `readonly` modifier. The translation of Listing 4.15a to Java is shown in Listing 4.15d.

Listing 4.15d

Java translation

```

1 public static final Number VAT = 0.18;
2
3 public static final Item ItemForSale
4   = new Item("A hat", 34.99);

```


“ The purpose of computing is insight, not numbers.

– Richard Hamming (1915-1998)

5 Types

VDM-SL and C# are both statically typed languages. There are two kinds of types in VDM-SL: basic types of atomic values and compound types of structurally complex values. All basic types as well as the underlying data structures for collections are built into VDM-SL. The .NET standard libraries provide similar types for C#. The remaining kinds of types are defined by hand.

This chapter presents the translations of VDM-SL types to corresponding C# types. It also treats collection-based expressions such as enumerations, comprehensions and quantified expressions. The sections below follow the common structure outlined in Section 1.3.1.



Chapter contents

Section 5.1 translates the VDM-SL `rat` type to 64-bit and 128-bit floating-point numbers plus arbitrarily precise numbers in C#.

Section 5.2 translates mapping enumerations, set comprehensions and universal quantifications in VDM-SL to LINQ-based collections in C# and discusses different syntax options.

Section 5.3 translates VDM-SL union types to subtyping and generics in C#.

Section 5.4 translates VDM-SL quote types to C# enums.

Section 5.5 translates VDM-SL composite types to structs and data classes in C#.

Section 5.6 translates VDM-SL type aliases to `using` aliases and data classes in C# and discusses the treatment of self-referential types to ensure that they always terminate in C#.

Section 5.7 translates the VDM-SL `nat1` type to unsigned integers and invariants in C# and generalises the discussion to all kinds of type invariants.

5.1 Numeric types

VDM-SL has five built-in types for representing numbers: `nat1` (\mathbb{N}_1), `nat` (\mathbb{N}_0), `int` (\mathbb{Z}), `rat` (\mathbb{Q}) and `real` (\mathbb{R}). All of them support numbers of arbitrary magnitude and precision, which is feasible due to the fact that VDM-SL is a modelling language. C#, on the other hand, is a programming language, which faces technical limitations on how to store numbers effectively in memory while retaining computational efficiency.



Example 5.1

Rational numbers

Listing 5.1a shows a VDM-SL value `MysteryConstant` (line 2) that defines a rational constant of 0.123456 and a function `IsMysteryConstant` (lines 5-6) that tests if a rational number `q` is equal to the value of `MysteryConstant`.

Listing 5.1a

VDM-SL

```

1  values
2      MysteryConstant = 0.123456;
3
4  functions
5      IsMysteryConstant: rat -> bool
6      IsMysteryConstant(q) == q = MysteryConstant;
```

One of the built-in types is the `rat` numeric basic type, which represents rational numbers of arbitrary magnitude and precision. This section presents three approaches to translating the `rat` type in VDM-SL to corresponding numeric types in C#.



Translation proposals

[Section 5.1.1](#) describes the approach of using the 64-bit double floating-point type in C# to represent rational numbers of finite magnitude and mediocre precision.

[Section 5.1.2](#) describes the approach of using the 128-bit decimal floating-point type in C# to represent rational numbers of finite magnitude and great precision.

[Section 5.1.3](#) describes the approach of using the `BigRational` type in C# to represent rational numbers of arbitrary magnitude and arbitrary precision.

5.1.1 Binary floating-point numbers

In C#, a reasonable equivalent for the `rat` type is the `double` type, which is an alias for the 64-bit `System.Double` type in the .NET standard libraries. It represents binary floating-point numbers, which are just approximations of numbers and therefore have limited magnitude and precision. It also suffers from `LOSS OF SIGNIFICANCE` in arithmetic calculations, in which the precision of the numbers deteriorates even further when numeric inaccuracies add up.

This gives rise to additional concerns regarding the translations of arithmetic operations from VDM-SL to C#, which must take precautions to avoid loss of significance. For instance, comparing two floating-point numbers for equality is equivalent to comparing their difference to a small threshold number close to zero, permitting small inaccuracies within the threshold due to loss of significance.



Example 5.1, continued

Rational numbers

Listing 5.1a showed a value `MysteryConstant` and a function `IsMysteryConstant` in VDM-SL. Listing 5.1b shows the corresponding C# translation which contains a read-only property `MysteryConstant` (line 1) and a method `IsMysteryConstant` (lines 3-5) that compares the difference between `q` and `MysteryConstant` to a threshold value of 0.00001, thus permitting inaccuracies from the sixth decimal place and onwards.

Listing 5.1b

C# translation

```
1 public static double MysteryConstant { get; } = 0.123456;
2
3 [Pure]
4 public static bool IsMysteryConstant(double q)
5     => Math.Abs(q - MysteryConstant) < 0.00001;
```

5.1.2 Decimal floating-point numbers

An alternative to `double` is `decimal`, which is an alias for the 128-bit `System.Decimal` type of .NET that represents decimal floating-point numbers, that is, floating-point numbers in base 10 rather than base 2. It stores the exact representations of numbers rather than approximations and does not suffer from loss of significance in arithmetic computations. Its magnitude and precision is greater than `double`, but it is still not arbitrary.

Conversely, it requires twice as much memory and computations involving `decimal` are significantly slower than those involving `double` [43] [44]. Furthermore, essential methods in the `System.Math` library, for example `Pow` and `Log`, operate on `double` values and are not geared to dealing with `decimal`.



Example 5.1, continued

Rational numbers

Listing 5.1a showed a value `MysteryConstant` and a function `IsMysteryConstant` in VDM-SL. Listing 5.1c shows the C# translation using the `decimal` type to represent rational numbers. `decimal` literals carry a suffix of ‘m’ (line 1). Note that `IsMysteryConstant` compares the numbers for exact equality (line 4).

Listing 5.1c

C# translation

```
1 public static decimal MysteryConstant { get; } = 0.123456m;
2
3 public static bool IsMysteryConstant(decimal q)
4     => q == MysteryConstant;
```

5.1.3 Arbitrary magnitude and precision

A third approach is to use a `BigRational` type to achieve arbitrary magnitude and precision like VDM-SL. The .NET platform provides this type as an experimental library under the Base Class Libraries project [45], but there is no built-in language support for `BigRational` in C#. The disadvantages of having arbitrary magnitude and precision are the expensive computations and the large memory consumption.



Example 5.1, continued

Rational numbers

Listing 5.1a showed a value `MysteryConstant` and a function `IsMysteryConstant` in VDM-SL. Listing 5.1d shows the C# translation using the `BigRational` type to represent rational numbers. It is constructed from another floating-point number (line 2), but retains arbitrary precision in computations.



Listing 5.1d

C# translation

```
1 public static BigRational MysteryConstant { get; }
2     = new BigRational(0.123456m);
3
4 public static bool IsMysteryConstant(BigRational q)
5     => q == MysteryConstant;
```

5.1.4 Rule summarisation

It is important to have in mind that VDM-SL is used by software engineers to model mission-critical systems, including financial systems that use the `rat` type for monetary calculations. This rules out the inaccurate `doubl e` type as a candidate in C# because it is not suitable in such applications. Indeed, the .NET documentation points out that the `deci mal` type is the appropriate choice for financial applications, leaving `Bi gRati onal` as an overkill. Rule 5.2 summarises the translation of the `rat` type in VDM-SL to the `deci mal` type in C#.

	<p>Rule 5.2 Rational numbers</p> <p>The <code>rat</code> type of VDM-SL becomes the <code>deci mal</code> type of C#. The numeric literal receives a suffix of ‘m’ to indicate that it is an instance of the <code>deci mal</code> type.</p>
	<p>Limitations Rule 5.2</p> <p>The <code>deci mal</code> type has only finite magnitude and precision.</p>

The Java code generator in Overture translates all numeric types of VDM-SL to the abstract `Number` class of Java, which represents any kind of numeric value. Among its subclasses are `Doubl e` and `Bi gDeci mal`, which are equivalent to `Doubl e` and `Bi gRati onal` in .NET. When the generated Java code has to interoperate with `rat` literals, it calls the `doubl eVal ue` method on the `Number` instance to retrieve its numeric value, thereby narrowing it to 64-bit binary floating-point precision. The translation of Listing 5.1a to Java is shown in Listing 5.1e. Listing 5.1f shows a fragment of the `Uti l s. equal s` helper method.

	<p>Listing 5.1e Java translation</p> <pre> 1 public static final Number MysteryConstant = 0.123456; 2 3 /*@ pure @*/ 4 public static Boolean IsMysteryConstant(final Number q) 5 { 6 Boolean ret_41 = Uti l s. equal s(q, MysteryConstant); 7 return ret_41; 8 }</pre>
--	--

Listing 5.1f

Utils.java

```

1  public static boolean equals(Object left, Object right)
2  {
3      // Code omitted ...
4
5      if (left instanceof Number && right instanceof Number)
6      {
7          Double leftNumber = ((Number) left).doubleValue();
8          Double rightNumber = ((Number) right).doubleValue();
9
10         return leftNumber.compareTo(rightNumber) == 0;
11     }
12
13     // Code omitted ...
14 }

```

5.2 Collection types

There are three kinds of collections in VDM-SL: sets, sequences and mappings. Sets are finite collections of unordered items without duplications; sequences are finite collections of ordered items with possible duplications; and mappings are finite collections of unordered key-value pairs where all keys are unique. Furthermore, VDM-SL provides multiple built-in operators for manipulating these collections [7].

The corresponding collection types for sets, sequences and mappings in .NET are `HashSet<T>`,³ `List<T>` and `Dictionary<TDomain, TRange>`, respectively. In fact, .NET unifies all collection types under a single generic interface, `IEnumerable<T>`, and provides a standard library of extension methods for manipulating `IEnumerable<T>`-based collections: LINQ [46].⁴ The collection types are defined in the `System.Collections.Generic` namespace and the LINQ extension methods are defined in `System.Linq`.

This section presents the translations of mapping enumerations, set comprehensions and universal quantifications to .NET collections and LINQ-based expressions in C#.

³ .NET has chosen the name ‘HashSet’ rather than just ‘Set’ because Set is a reserved word in the Visual Basic .NET language.

⁴ Language-Integrated Query.



Translation proposals

Section 5.2.1 describes a single approach to translating mapping enumerations in VDM-SL to initialised dictionaries in C#.

Section 5.2.2 describes a single approach to translating set comprehensions in VDM-SL to LINQ queries in C# from two syntactical points of view.

Section 5.2.3 describes a single approach to translating universally quantified expressions in VDM-SL to LINQ queries in C#.

5.2.1 Mapping enumerations

A mapping enumeration in VDM-SL creates a fixed set of key-value pairs, called maplets, by enumerating them within a pair of curly braces.



Example 5.3

Mapping enumerations

Listing 5.3a shows two values `Nothing` and `Successor`, both of type `map int to int`. `Nothing` is initialised with an empty mapping enumeration (line 2), corresponding to the empty mapping $\emptyset \rightarrow \emptyset$, whereas `Successor` is initialised with a mapping enumeration of two maplets that corresponds to $\{1 \mapsto 2, 2 \mapsto 3\}$ (line 3).

Listing 5.3a

VDM-SL

```
1  values
2      Nothing: map int to int = { |-> };
3      Successor: map int to int = { 1 |-> 2, 2 |-> 3 };
```

The straightforward way to create a mapping in C# is to instantiate the `Dictionary<TDomain, TRange>` type. It creates an empty dictionary – corresponding to an empty mapping – but it can be populated from the beginning by invoking its collection initialiser immediately upon instantiation.



Example 5.3, continued

Mapping enumerations

Listing 5.3a showed two values `Nothing` and `Successor` of type `map int to int` in VDM-SL. Listing 5.3b shows the C# translation output two read-only properties `Nothing` and `Successor` initialised to instances of the `Dictionary<int, int>` type. The `Dictionary` object in `Nothing` contains no elements (line 2), whereas the `Dictionary` object of `Successor` is populated with two key-value pairs via a collection initialiser (line 5).

Listing 5.3b

C# translation

```

1 public static Dictionary<int, int> Nothing { get; }
2   = new Dictionary<int, int>();
3
4 public static Dictionary<int, int> Successor { get; }
5   = new Dictionary<int, int> { [1] = 2, [2] = 3 };

```

Rule 5.4 summarises the translation of mapping enumerations from VDM-SL to dictionaries with collection initialisers in C#.



Rule 5.4

Mapping enumerations

A mapping enumeration in VDM-SL becomes an instance of `Dictionary<TDomain, TRange>` in C#. The maplets are added as elements to the dictionary through a collection initialiser.

The Java standard library offers the `HashMap<TDomain, TRange>` type, which is equivalent to `Dictionary<TDomain, TRange>` type. However, since Java has no concept of collection initialisers like C#, the Java code generator in Overture employs its own `VDMMap` type instead, which is complemented by the `MapUtil` utility class to create mappings easily. Furthermore, it provides a `Maplet` class for representing key-value pairs. The Java translation of Listing 5.3a is shown in Listing 5.3e.

Listing 5.3e

Java translation

```

1 public static final VDMMap Nothing = MapUtil.map();
2
3 public static final VDMMap Successor
4   = MapUtil.map(new Maplet(1L, 2L), new Maplet(2L, 3L));

```

5.2.2 Set comprehensions

A set comprehension instantiates a set using the set-builder notation, which consists of three components in VDM-SL [1] [7]: a term, one or more set or type bindings and an optional predicate. It filters the elements in the bindings according to the predicate and projects them into a new set according to the term.



Example 5.5

Set comprehensions

Listing 5.5a shows three values `ThreeToTen`, `EvenNumbers`, both of type `set of int`, and `Matrix3x3x3` of type `set of (int * int * int)`. `ThreeToTen` is initialised by a set range corresponding to the set $\{3, \dots, 10\}$ (line 2). `EvenNumbers` is initialised by a set comprehension with a type binding to `nat1` and corresponds to the set $\{2x \mid x \in \mathbb{N}_1 \wedge x \leq 5\}$ (line 3). `Matrix3x3x3` is initialised by a set comprehension with a set binding and corresponds to the set $\{(x, y, z) \mid x, y, z \in \{1, \dots, 3\}\}$ (lines 4-5).

Listing 5.5a

VDM-SL

```

1  val ues
2      ThreeToTen: set of int = { 3, ..., 10 };
3      EvenNumbers: set of int = { 2 * x | x: nat1 & x <= 5 };
4      Matrix3x3x3: set of (int * int * int)
5          = { mk_(x, y, z) | x, y, z in set { 1, ..., 3 } };

```

While there is no dedicated set comprehension construct in C#, LINQ provides a similar feature through its `Where` and `Select` extension methods. The `Where` method filters the elements in a collection and is equivalent to the predicate in set comprehensions. The `Select` method performs element projections and is equivalent to the term. Both methods take a lambda expression as input from a single element in the collection to a predicate and a projection, respectively.

The binding in the set comprehension is reflected by the fact that `Where` and `Select` are extensions of `IEnumerable<T>` and therefore operate on the source collection. When the set comprehension has multiple bindings, the `Select` method is replaced by the `SelectMany` method. It takes another collection as input in addition to the one being manipulated and computes a projection from elements of both collections.

Type bindings need special treatment because types in .NET do not inherently represent sets of values as type bindings do in VDM-SL. Countably infinite types are translated to finite collections that contain all representable elements of the type; for example, a type binding of `nat1` becomes a collection of all values of the C# `int` type from 1 to $2^{31} - 1$. A smaller upper bound can be justified, as the VDM-SL interpreter in Overture represents a type binding of `nat1` by the elements in $\{1, \dots, 255\}$.

A set range in VDM-SL is a special case of a set comprehension. It creates a set of integers that lie within a given lower bound and upper bound, both inclusive. It is translated to `Enumerable.Range` in C#, which generates a collection of integers from a given lower bound and onwards until it contains a certain number of elements. Note that it generates the collection lazily like a `STREAM`.



Example 5.5, continued

Set comprehensions

Listing 5.5a showed three values `ThreeToTen`, `EvenNumbers`, both of type `set of int`, and `Matrix3x3x3` of type `set of (int * int * int)`. Listing 5.5b shows the corresponding C# translations, all of which use `Enumerable`. `ThreeToTen` creates the set `{3, ..., 10}` by calling `Enumerable.Range` with arguments for the lower bound, 3, and the number of elements, 8 (line 2). `EvenNumbers` calls `Where` (line 6) and `Select` (line 7) to filter and project the elements of the `nat1` type binding. `Matrix3X3X3` projects multiple collection sources into a single collection via `SelectMany` (lines 13-17). In the process, it uses an anonymous type (line 14) to save the `x` and `y` elements before merging them with the `z` element into an instance of `Tuple<int, int, int>` (line 17).

Listing 5.5b

C# translation

```

1 public static HashSet<int> ThreeToTen { get; }
2   = Enumerable.Range(3, 8).ToHashSet();
3
4 public static HashSet<int> EvenNumbers { get; }
5   = Enumerable.Range(1, int.MaxValue)
6       .Where(x => x <= 5)
7       .Select(x => 2 * x)
8       .ToHashSet();
9
10 public static HashSet<Tuple<int, int, int>>
11     Matrix3X3X3 { get; }
12   = Enumerable.Range(1, 3)
13       .SelectMany(x => Enumerable.Range(1, 3),
14                   (x, y) => new { x, y })
15       .SelectMany(_ => Enumerable.Range(1, 3),
16                   (_, z) =>
17                       Tuple.Create(_, x, _.y, z))
18       .ToHashSet();

```

Based on the functionality in LINQ, the C# language goes one step further by providing built-in language support for the `Where-Select` pattern. It is known as the query syntax as opposed to method-based queries [47].

A query is initiated with a `from...i n` clause that specifies the collection to use as a starting point. It also defines a *range variable*, which is a placeholder for the individual elements in the collection. The `from...i n` clause is followed by an optional `where` clause, which states a filtering predicate, and a mandatory `sel ect` clause, which states a projection. The `where` and `sel ect` clauses may use the range variable in their computations. Compound `from...i n` clauses are useful for set comprehensions with multiple bindings. Here, the first `from...i n` clause is simply followed by another one that specifies the second collection and defines its corresponding range variable.

Since the query syntax builds on top of LINQ, it returns instances of `I Enumerabl e<T>`. They are converted to sets with `ToHashSet`, which does not have an equivalent query syntax and is called as usual as an extension method on the query result. Note that `ToHashSet` is a utility method provided by the runtime library of the transcompiler because it is not built into LINQ unlike its sibling methods `ToLi st` and `ToDi cti onary`.



Example 5.5, continued

Set comprehensions

Listing 5.5a showed three values `ThreeToTen`, `EvenNumbers`, both of type `set of i nt`, and `Matri x3x3x3` of type `set of (i nt * i nt * i nt)`. Listing 5.5c shows the C# translations of `EvenNumbers` (lines 1-4) and `Matri x3X3X3` (lines 6-11) using the query syntax of LINQ. The `from...i n` clause for `EvenNumbers` defines a single range variable `x` and the compound `from...i n` clause for `Matri x3X3X3` defines three range variables `x`, `y` and `z`.

Listing 5.5c

C# translation

```

1 public static HashSet<i nt> EvenNumbers { get; }
2     = (from x i n Enumerabl e. Range(1, i nt. MaxVal ue)
3         where x <= 5
4         sel ect 2 * x). ToHashSet();
5
6 public static HashSet<Tupl e<i nt, i nt, i nt>>
7     Matri x3X3X3 { get; }
8     = (from x i n Enumerabl e. Range(1, 3)
9         from y i n Enumerabl e. Range(1, 3)
10        from z i n Enumerabl e. Range(1, 3)
11        sel ect Tupl e. Create(x, y, z)). ToHashSet();

```

The query syntax gets rid of the lambda expressions in method-based queries due to the range variable which the `where` and `select` clauses may refer to. The lambda expressions in method-based queries do not define range variables; instead, they take the individual collection element as input. Therefore, they must repeatedly state its name in every query method.

When there are multiple bindings in the VDM-SL set comprehensions, the query syntax is also more compact than method-based queries due to the conciseness of compound `from...in` clauses versus `SelectMany`.

For these reasons, the .NET documentation recommends using the query syntax over method-based queries whenever possible [47]. Rule 5.6 summarises the translation of set comprehensions in VDM-SL to LINQ-based query syntax in C#.



Rule 5.6

Set comprehensions

A set comprehension in VDM-SL becomes an instance of `HashSet<T>` via LINQ-based query syntax in C#. A single binding becomes a `from...in` clause, whereas multiple bindings become a compound `from...in` clause. The predicate becomes a `where` clause. The term becomes a `select` clause. The result is converted to a set by calling `ToHashSet` on the result of the LINQ query. Set ranges and type bindings become `Enumerable.Range`.



Limitations

Rule 5.6

A type binding represents only a finite subset of the elements.

Java has no concept of extension methods like C#; thus, there is no equivalent functionality to LINQ in Java although the Stream API of Java 8 provides something similar minus the built-in language support [48]. However, the Java code generator in Overture targets Java 7 and thus resorts to adding set elements from set comprehensions iteratively.⁵ The result is an instance of `VDMSet`, its own set implementation. As a consequence of this approach, the Java code generator cannot generate set comprehensions in VDM-SL values because these are represented by fields, cf. Section 4.5.3. It will only translate set comprehensions within functions and operations. Furthermore, it does not support type bindings in set comprehensions at all.

⁵ As of June 2016 in Overture 2.3.

The Java translation of Listing 5.5a is shown in Listing 5.5d after refactoring the VDM-SL values into parameterless functions and swapping the type binding in EvenNumbers with an ordinary set binding of {1, ..., 255}.

Listing 5.5d

Java translation

```

1  /*@ pure @*/
2  public static VDMSet ThreeToTen()
3  {
4      VDMSet ret_1 = SetUtil.range(3L, 10L);
5      return Utils.copy(ret_1);
6  }
7
8  /*@ pure @*/
9  public static VDMSet EvenNumbers()
10 {
11     VDMSet setCompResult_1 = SetUtil.set();
12     VDMSet set_1 = SetUtil.range(1L, 255L);
13
14     for (Iterator iterator_1 = set_1.iterator();
15          iterator_1.hasNext();)
16     {
17         Number x = ((Number) iterator_1.next());
18
19         if (x.longValue() <= 5L)
20         {
21             setCompResult_1.add(2L * x.longValue());
22         }
23     }
24
25     VDMSet ret_2 = Utils.copy(setCompResult_1);
26     return Utils.copy(ret_2);
27 }
28
29 /*@ pure @*/
30 public static VDMSet Matrix3x3x3()
31 {
32     VDMSet setCompResult_2 = SetUtil.set();
33     VDMSet set_2 = SetUtil.range(1L, 3L);
34
35     for (Iterator iterator_2 = set_2.iterator();
36          iterator_2.hasNext();)
37     {
38         Number x = ((Number) iterator_2.next());
39
40         for (Iterator iterator_3 = set_2.iterator();
41              iterator_3.hasNext();)
42         {

```

... *Continues on the next page ...*

```

...      Continued from the previous page ...

43      Number y = ((Number) iterator_3.next());
44
45      for (Iterator iterator_4 = set_2.iterator();
46           iterator_4.hasNext();)
47      {
48          Number z = ((Number) iterator_4.next());
49          setCompResult_2.add(Tuple.mk_(x, y, z));
50      }
51  }
52  }
53
54  VDMSet ret_3 = Utils.copy(setCompResult_2);
55  return Utils.copy(ret_3);
56  }

```

5.2.3 Universal quantifications

Set and type bindings also occur in quantified expressions, for example universal quantifications, which compute whether a predicate is satisfied for all elements in a set.



Example 5.7

Universal quantifications

Listing 5.7a shows an implicit function `Max` (lines 2-3) whose postcondition is a universal quantification proclaiming that the result value `r` is greater than or equal to all elements of the input set `S` of natural numbers (line 3). In mathematical terms: $\forall n \in S: r \geq n$.

Listing 5.7a

VDM-SL

```

1  functions
2      Max(S: set of nat) r: nat
3      post forall n in set S & r >= n;

```

The LINQ query syntax can be used to solve this challenge in C#. The translation of bindings in universal quantifications is identical to the one for set comprehensions: a `from...in` clause, compound if necessary.

The predicate in a universal quantification behaves differently than the predicate in a set comprehension whose purpose is to filter elements accordingly. Therefore, it is not desirable to use the `where` clause in the LINQ query as it would just filter out elements that do not satisfy the predicate, hence affecting the final result of the universal quantification. Instead, the predicate is computed directly in the `select` clause so that the final result of the LINQ query is a collection of Boolean values.

LINQ provides an `All` method which tests whether a predicate is satisfied for all elements in a collection. In this case, it simply tests that all Boolean values are true. Like the `Select`, `Where` and `SelectMany` methods of LINQ, it takes a lambda expression as input.

In principle, the `All` method could simply evaluate the predicate of the universal quantification directly for all elements; however, the `select` clause is mandatory and the range variable of the `from...in` clause would not be in the scope of `All` so it must be provided instead through the parameter of the lambda expression. This becomes even more complex for compound `from...in` clauses with multiple range variables, cf. Listing 5.5b.



Example 5.7, continued

Universal quantification



Listing 5.7a showed an implicit function `Max` with a universally quantified postcondition. Listing 5.7b shows the C# translation of the postcondition function `PostMax` (lines 1-6) which computes the universal quantification through a LINQ query. The `select` clause evaluates the predicate of the universal quantification (line 5). The `All` method tests that each resulting Boolean value is true – its lambda expression argument is the identity function for Booleans (line 5). Two inferred preconditions for type consistency of `s` and `r` have been omitted.

Listing 5.7b

C# translation

```
1 [Pure]
2 public static bool PostMax(HashSet<int> s, int r)
3 {
4     // Code omitted ...
5     return (from n in s select r >= n).All(_ => _);
6 }
```

Rule 5.8 summarises the translation of universally quantified expressions in VDM-SL to LINQ queries in C#.

	<p>Rule 5.8 Universal quantifications</p> <p>A universally quantified expression in VDM-SL becomes a LINQ query in C#. A binding is translated identically to bindings in Rule 5.6. The predicate becomes a select clause. The final result is computed by calling <code>AI I (_ => _)</code> on the result of the LINQ query.</p>
	<p>Limitations Rule 5.8</p> <p>Same limitations as Rule 5.6.</p>

Like for set comprehensions, the Java code generator in Overture uses an iterative approach to evaluating universal quantifications, bringing along the limitation that it will only translate universal quantifications within functions and operations. It evaluates the predicate for every element in the binding until it completes all elements or hits an element that does not satisfy the predicate, making the entire result false. The Java translation of the postcondition in Listing 5.7a is shown in Listing 5.7c.

	<p>Listing 5.7c Java translation</p> <pre> 1 /*@ pure @*/ 2 public static Boolean post_Max(final VDMSet S, 3 final Number r) 4 { 5 // Code omitted ... 6 Boolean forAllExpResult_1 = true; 7 VDMSet set_1 = Utils.copy(S); 8 9 for (Iterator iterator_1 = set_1.iterator(); 10 iterator_1.hasNext() && forAllExpResult_1;) 11 { 12 Number s = ((Number) iterator_1.next()); 13 forAllExpResult_1 = r.longValue() >= s.longValue(); 14 } 15 16 Boolean ret_1 = forAllExpResult_1; 17 return ret_1; 18 } </pre>
--	---

5.3 Union types

VDM-SL has built-in support for untagged unions, which accept values from two or more separate types.



Example 5.9

Union types

Listing 5.9a shows a function `Check` (lines 2-3) which takes as input a union of an `int` and a `seq of char` and produces a value of either type `bool` or type `int`. When the input is an integer, the result value of `Check` is the same integer. Otherwise, the result value is the Boolean value of `false`.

Listing 5.9a

VDM-SL

```

1  functi ons
2      Check: int | seq of char -> bool | int
3      Check(x) == if is_int(x) then x else false;
```

C# does not have a direct equivalent to union types, but similar semantics are achievable through subtyping. This section presents three approaches to translating union types in VDM-SL to subtyping in C#.



Translation proposals

[Section 5.3.1](#) describes the approach of using the `object` type in C# to represent union types with type checking on runtime.

[Section 5.3.2](#) describes the approach of using interface inheritance to represent union types with type checking on compile-time.

[Section 5.3.3](#) describes the approach of using the `object` type in conjunction with generics to represent union types with type checking on compile-time.

5.3.1 Plain object-based subtyping

All classes, enums and structs in .NET are subclasses of `System.Object`, which is the root of the class hierarchy and thus the only class that does not have a base class [3], see Figure 5.10. Therefore, the `object` type, which is an alias for `System.Object` in C#, accepts any class instance [3].

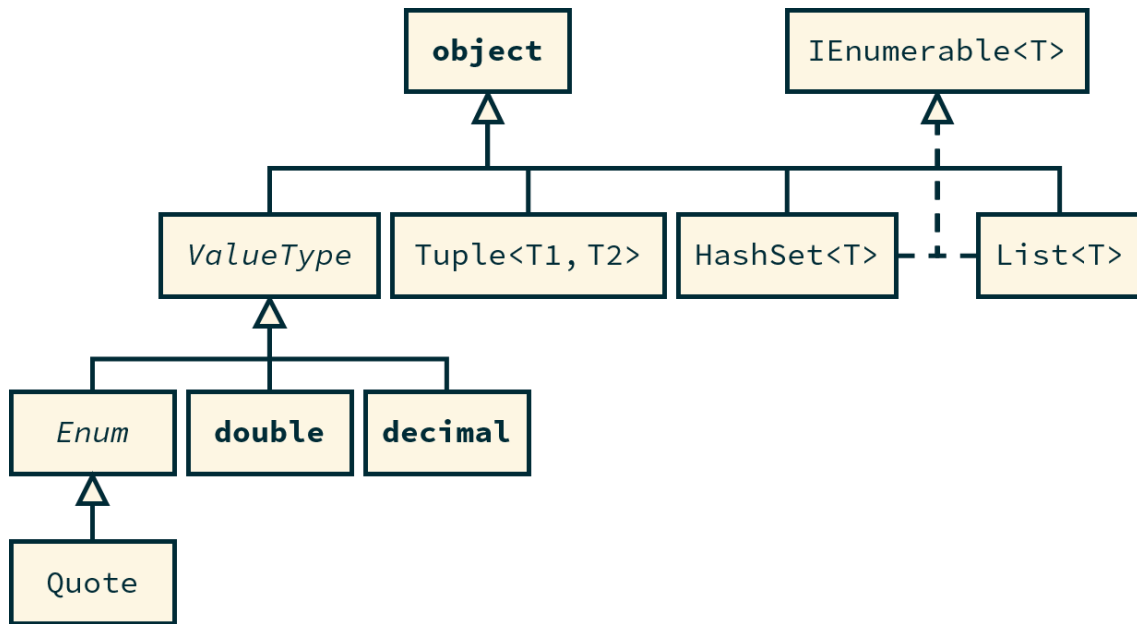


Figure 5.10

The object type is the root of the class hierarchy in .NET.

Note that even though an interface in C# may only inherit other interfaces and is not derived from `object`, all interface implementations are compatible with `object`. Due to the mechanisms of subtyping, this implies that `object` acts as a union of all .NET class types, accepting instances of any type. Hence, in the translation from VDM-SL to C#, all occurrences of union types become `object`. Whenever the underlying contents of the union is a value type such as `int`, it is subject to the `Boxing` phenomenon.



Example 5.9, continued

Union types

Listing 5.9a showed a function `Check` whose parameter type and result type are unions. Listing 5.9b shows the corresponding `Check` method in C# with the `object` type as parameter type and return type. An inferred precondition and postcondition for type consistency of `x` and the result value have been omitted.

Listing 5.9b

C# translation

```

1 [Pure]
2 public static object Check(object x)
3 {
4     // Code omitted ...
5     return (x is int) ? x : false;
6 }
  
```


A limitation of this solution is the fact that the `object` type accepts instances of any type, not only the ones covered by the union type in VDM-SL. Therefore, to ensure program correctness, it must be verified that the value of the union is an instance of an anticipated type. This can be done by type invariants, as described in Section 5.7. Nevertheless, the C# code still loses ANALYSABILITY **since ‘object’ is less describing than** for instance **‘union of int and string’ and ‘union of bool and int’**.

Boxing is a challenging issue because it obscures the real, underlying type of the value. For instance, it disables any pass-by-value semantics of structs and enums, deferring to pass-by-sharing of object references. This is problematic from a VDM-SL perspective, which relies solely on pass-by-value.

Another problem with boxing is the fact that the equality operator (`==`) falls back to referential equality even if it has been overloaded by the underlying type, for example to provide structural equality. Overloaded operators in C# are resolved on compile-time rather than runtime and thus, the `object` type hides the underlying type from the compiler. Using the ordinary `Equals` method solves this problem in general because it is *overridden* by the underlying type rather than *overloaded*. Therefore, it is resolved on runtime by SUBTYPE POLYMORPHISM. Still, many of the built-in types in the .NET standard libraries, for example `HashSet<T>`, `List<T>` and `Dictionary<TDomain, TRange>`, do not override `Equals` to provide structural equality as desired from a VDM-SL perspective. They provide type-specific methods for comparisons instead, such as `SetEquals` for `HashSet<T>` and `SequenceEqual`⁶ for `List<T>`, but they are not available without type casting the `object`-typed variable to the specific underlying type.

5.3.2 Interface-based subtyping

Another subtyping-based approach is to utilise interfaces, see Figure 5.11. Each union type in VDM-SL is represented by an empty interface in C#, which is implemented by each class constituting the union. The interface type only accepts instances of the implementing types, which correspond exactly to the types covered by the union. In contrast to the `object` type, the consistency of the constituent types can be checked on compile-time. With proper naming of the interfaces, the code may also retain its analysability. **Due to C#’s support for multiple interface inheritance**, a class may take part in multiple unions.

⁶ It is named ‘SequenceEqual’ **without a trailing ‘s’**, unlike `Equals` and `SetEquals`. It is in fact a LINQ extension method unlike the `SetEquals` method, which is implemented by the `HashSet<T>` class.

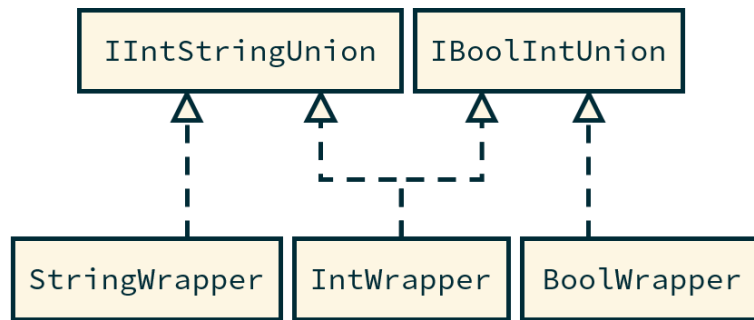


Figure 5.11
Subtyping with interfaces.

The interface-based approach faces another issue, though. The built-in types of VDM-SL are translated to built-in types of .NET, for example `int` and `List<T>`, whose implementations are closed for modification. Therefore, they cannot implement the interfaces that represent union types. A workaround is to create wrapper classes of the constituent .NET types. These wrapper classes are under control of the transcompiler and may implement the necessary interfaces. Nevertheless, this leads to very undesirable boilerplate code.



Example 5.9, continued

Union types

Listing 5.9a showed a function `Check` whose parameter type and result type are unions. Listing 5.9c shows an interface-based approach to implementing unions in C#. The union type of `int` and `seq of char` is represented by the empty `IIntStringUnion` interface (line 1) and implemented by `IntWrapper` (lines 5-8) and `StringWrapper` (lines 9-12) which are wrapper classes for `int` and `string`. The same goes for the union type of `bool` and `int` and the `IBoolIntUnion` interface (line 3). `BoolWrapper` (lines 14-17) is a wrapper class for `bool`. An inferred precondition and postcondition for type consistency of `x` and the result value have been omitted.

Listing 5.9c

C# translation

```

1 public interface IIntStringUnion { }
2
3 public interface IBoolIntUnion { }
4
5 public class IntWrapper : IIntStringUnion, IBoolIntUnion
6 {
7     // Implementation omitted ...
8 }

```

... *Continues on the next page ...*

```
... Continued from the previous page ...

9 public class StringWrapper : IIntStringUnion
10 {
11     // Implementation omitted ...
12 }
13
14 public class BoolWrapper : IBoolIntUnion
15 {
16     // Implementation omitted ...
17 }
18
19 [Pure]
20 public static IBoolIntUnion Check(IIntStringUnion x)
21 {
22     // Code omitted ...
23     return x is IntWrapper ? x : new BoolWrapper(false);
24 }
```

5.3.3 object-based subtyping with generics

An alternative to direct subtyping is to describe all kinds of union types with a generic helper class that provides all the needed functionality out of the box. A suitable name for this helper class is `Union`. Its type parameters indicate the types covered by the union and help retaining the analysability of the code. Behind the scenes, it has a property of type `object` which contains the instance of the corresponding union type, but implies that boxing remains an issue as in the first approach. It also has a constructor overload for each valid type.

By utilising the generic type system of .NET, the compiler will only accept instances of valid types to be stored in the `object`-typed property, thus ensuring type safety from compile-time generics rather than runtime type invariants.

It is possible to retrieve the underlying value by accessing the property and casting it to the appropriate type. Alternatively, the `Union` helper class may provide generic casting operators to avoid accessing the property value explicitly. Type casts may fail on runtime, though, if it turns out that the underlying value is an instance of a different type than anticipated. Hence, they should be guarded carefully by checking the type in advance and branching the program flow accordingly, for example by a conditional statement.



Example 5.9, continued

Union types

Listing 5.9a showed a function `Check` whose parameter type and result type are unions. Listing 5.9d shows the `Check` function with a parameter of type `Union<int, string>` (line 2), corresponding to the union of `int` and `seq of char`, and a return type of `Union<bool, int>` (line 2), corresponding to the union of `bool` and `int`. The underlying value is accessed through the object-typed property, which is named `Value` (line 5). An inferred precondition and postcondition for type consistency of `x` and the result value have been omitted.

Listing 5.9d

C# translation

```

1  [Pure]
2  public static Union<bool, int> Check(Union<int, string> x)
3  {
4      // Code omitted ...
5      return x.Value is int
6          ? new Union<bool, int>(x)
7          : new Union<bool, int>(false);
8  }
```

The main disadvantage of resorting to a helper class is the overhead and boilerplate code endured by the fact that it wraps the instance of the union type into an instance of `Union`. Even with explicit return types of `Union<T1, T2>` and implicit type casts from `T1` and `T2`, the C# compiler cannot infer the type of the return value automatically. It requires a manual instantiation of `Union<T1, T2>`.

Another disadvantage is the shipment of the helper class along with the translated output that it complements. In VDM-SL, there is no upper bound to the number of types covered by a union type. A generic C# class, however, is born with a fixed number of type parameters. There are no variadic generics in .NET for supporting an arbitrary number of type parameters; hence, `Union<T1, T2>` and `Union<T1, T2, T3>` are defined as two distinct classes. The transcompiler must provide a helper class for each needed arity of union types.

5.3.4 Rule summarisation

The simplicity of the first approach using the object type outshines the analysability gains of the two complicated approaches using interfaces and generics. Rule 5.12 summarises the translation of union types in VDM-SL to the object type in C#.



Rule 5.12

Union types

A union type of VDM-SL becomes object in C#.



Limitations

Rule 5.12

Type consistency is enforced by type invariants on runtime rather than compile-time. Boxing disables pass-by-value semantics and structural equality. The analysability of the code is impaired.

Similar to C#, there are no union types in Java like those of VDM-SL. However, the class hierarchy of Java resembles the one of .NET: All classes inherit from `Object`. The primitive types in Java, for instance `int` and `double`, do not inherit from `Object`, but they are complemented by wrapper classes such as `Integer` and `Double`, which box their values. The Java code generator uses the `Object` type to represent union types. The Java translation of Listing 5.9a is shown in Listing 5.9e.

Listing 5.9e

Java translation

```

1  /*@ pure @*/
2  public static Object Check(final Object x)
3  {
4      if (Utils.is_int(x))
5      {
6          Object ret_1 = x;
7          return ret_1;
8      }
9      else
10     {
11         Object ret_2 = false;
12         return ret_2;
13     }
14 }
```

5.4 Quote types

A quote type consists of a single quote literal representing a constant, named identifier. Quote types are often used in conjunction with union types, making them equivalent to ordinary enumerations in programming languages.



Example 5.13

Quote types

Listing 5.13a shows two type definitions `StatusCode` (line 2) and `ClientError` (line 3), both of which are unions of three quote types. The `BadRequest` quote type is a part of both types.

Listing 5.13a

VDM-SL

```

1 types
2   StatusCode = <OK> | <BadRequest> | <InternalServerError>;
3   ClientError = <BadRequest> | <Forbidden> | <NotFound>;

```

C# supports enumerations as well through its `enum` construct. This section presents the translation of quote types in VDM-SL to enums in C#.

A difference between VDM-SL quotes and C# enums is the fact that a quote type may occur in multiple union types, whereas an enum member is bound to its enclosing enum. Fortunately, enum members in C# are just enumerated `int` constants in disguise. The automatic enumeration mechanism can be overridden by assigning hardcoded `int` values to the enum members instead. The transcompiler can use this feature to keep track of all quote literals and assign `int` values to their corresponding enum members.

In particular, it defines a common enum `Quote` which holds enum members for all occurring quote literals. Each VDM-SL union type that consists entirely of quote types becomes an enum with enum members for the constituent quotes of the union. Their `int` values are retrieved from the common `Quote` enum to make quote literals across multiple enums consistent with each other. The enum is named after the type definition in UpperCamelCase – the naming convention for enums in .NET [8].



Example 5.13, continued

Quote types

Listing 5.13a showed two type definitions `StatusCode` and `ClientError` consisting entirely of quote types. Listing 5.13b shows the corresponding C# enums `StatusCode` (lines 10-15) and `ClientError` (lines 17-22) plus the common `Quote` enum (lines 1-8) that contains enum members for all quotes. The `int` values of the enum members in `StatusCode` and `ClientError` are retrieved from the respective enum members in `Quote`.

Listing 5.13b

C# translation

```

1 public enum Quote
2 {
3     Ok, // = 0
4     BadRequest, // = 1
5     InternalServerError, // = 2
6     Forbidden, // = 3
7     NotFound // = 4
8 }
9
10 public enum StatusCode
11 {
12     Ok = Quote.Ok, // = 0
13     BadRequest = Quote.BadRequest, // = 1
14     InternalServerError = Quote.InternalServerError // = 2
15 }
16
17 public enum ClientError
18 {
19     BadRequest = Quote.BadRequest, // = 1
20     Forbidden = Quote.Forbidden, // = 3
21     NotFound = Quote.NotFound // = 4
22 }

```

Creating enums for each union of quotes has the advantage that the C# compiler will only accept valid enum members in each case corresponding to valid quotes and will reject all other values.

Union types that contain quotes are not restricted to quotes exclusively. For instance, VDM-SL permits a union of a quote type and a seq of char type. Enums in C#, however, are limited to constant int members and cannot contain more complex types like text strings or variable integers. In this case, resorting to Rule 5.12 for union types is sufficient. Rule 5.14 summarises the translation of quote types in VDM-SL to enums in C#.



Rule 5.14

Quote types

All quote literals in the VDM-SL are tracked and become enum members of a common Quote enum.

A union type that consists entirely of quote types in VDM-SL becomes an enum in C# with an enum member for every constituent quote type whose int value is retrieved from the corresponding enum member in Quote. It is named after the type definition normalised to UpperCamelCase.

A union type that does not consist entirely of quote types becomes object rather than an enum by Rule 5.12.

Enumerations in Java are called *enum types* and they are different than enums in C#. The members of enum types, *enum constants*, are in fact objects rather than integers. It is therefore not possible to assign certain instances of enum types to other enum constants.

Defining a common enum type similar to Quote and retrieving all quote literals from here is not an option for the Java code generator in Overture. As it supports the VDM++ language, it needs to support overloaded functions and operations, which are not present in VDM-SL. Using a common Quote enum type whenever a quote type occurs may result in multiple overloaded methods having the same signature in Java even though their signatures differ in VDM++.

Instead, it creates a class for each quote literal and applies the Singleton design pattern to keep all occurrences of a quote literal in sync. The Java translation of the OK quote type in Listing 5.13a is shown in Listing 5.13c.

Listing 5.13c

Java translation

```
1 final public class OKQuote implements Serializable
2 {
3     private static int hc = 0;
4     private static OKQuote instance = null;
5
6     public OKQuote()
7     {
8         if (Utils.equals(hc, 0))
9         {
10             hc = super.hashCode();
11         }
12     }
13
14     public static OKQuote getInstance()
15     {
16         if (Utils.equals(instance, null))
17         {
18             instance = new OKQuote();
19         }
20
21         return instance;
22     }
23
24     public int hashCode() { return hc; }
25
26     public boolean equals(final Object obj)
27     {
28         return obj instanceof OKQuote;
29     }
30 }
```


5.5 Composite types

A composite type defines the named structure of a record-like type in VDM-SL. It is composed of mutable data fields, which exhibit pass-by-value semantics and structural equality. A field can be ignored in equality comparisons by marking it as an equality abstraction field [7]. Unlike built-in types and other compound types in VDM-SL, composite types may only appear in type definitions – or state definitions – and not directly as parameter types or result types in functions and operations.



Example 5.15

Composite types

Listing 5.15a shows a composite type definition `Person` with four fields: `name` (line 2), `age` (line 3), `favouriteColour` (line 4) and `likesDogs` (line 5). The two last fields are equality abstraction fields, indicated by the minus (-) character, implying that only the `name` and `age` fields will be used in comparisons. `Colour` is a union of quote types defined elsewhere.

Listing 5.15a

VDM-SL

```
1 types
2   Person :: name: seq of char
3           age: nat
4           favouriteColour: - Colour
5           likesDogs: - bool ;
```

This section presents two approaches to translating composite types in VDM-SL to structs and data classes in C#.



Translation proposals

[Section 5.5.1](#) describes the approach of using structs in C# to represent composite types with built-in pass-by-value semantics and structural equality.

[Section 5.5.2](#) describes the approach of using data classes to represent composite types with manually implemented pass-by-value semantics and structural equality.

5.5.1 Structs

Being influenced by the C-like programming languages, C# programs may declare data records with the `struct` language feature, which serves the same purpose as composite types in VDM-SL.

A struct is a type declaration that automatically inherits the `ValueType` class, providing it with pass-by-value semantics and structural equality instead of referential equality out of the box. It is named after the composite type in UpperCamelCase, which is the naming convention for structs in .NET [8]. It is defined as a public member of the class that represents the enclosing VDM-SL specification. Since the struct stands for a data record, it is tagged with `[Serializable]` to enable object serialisation. The fields of the composite type become auto-property members of the struct marked with the `public` modifier to make them accessible from the enclosing class. As mentioned in Sections 4.3 and 4.4, properties have an advantage over instance fields when it comes to flexibility and invariants. They are initialised from a public constructor.

When all fields of a struct are value types themselves, the structural equality feature provided by `ValueType` compares instances in memory byte-for-byte. Otherwise, it uses reflection to reveal all backing fields of the auto-properties and do a field-for-field comparison [49]. Consequently, it does not ignore equality abstraction fields in comparisons. To do so, the struct must explicitly override the `Equals` method of the `object` base class and implement the appropriate comparison itself.

Note that overriding `object.Equals` gives rise to boxing because the parameter of `Equals` is an `object` type. Before carrying out the comparison, the struct must unbox the parameter by type casting it to the type of the struct. This causes a small penalty in performance, especially when objects of structs are stored in arrays or generic collections. By letting the struct implement the `IEquatable<T>` interface, where `T` is the type name of the struct, it eliminates this penalty by implementing the strongly typed `Equals` method of `IEquatable<T>` alongside the `Equals` method of `object`.

When overriding the `Equals` method, it is obligatory to override `GetHashCode` as well to ensure that the hash value of the struct instance is consistent with the behaviour of `Equals`, which is expected by certain data structures such as `HashSet<T>` and `Dictionary<TDomain, TRange>` which depend on the hash values of objects to behave correctly.

An implementation of `Equals` must ensure that it upholds an EQUIVALENCE RELATION between objects. Likewise, an implementation of `GetHashCode` must ensure that equal objects have equal hash values while striving to produce distinct hash values for unequal objects to improve the performance of the hash-based data structures. Joshua Bloch [39] provides good solutions for both methods, which are adapted to this case and implemented by the struct.



Example 5.15, continued

Composite types

Listing 5.15a showed a composite type definition `Person` with four fields. Listing 5.15b shows the equivalent C# struct `Person` which implements the `IEquatable<Person>` interface. It has four auto-properties (lines 4-7) corresponding to the four fields of the composite type, a public constructor (lines 9-18) and implementations of `IEquatable.Equals` (lines 22-28), `object.Equals` (lines 30-34) and `GetHashCode` (lines 35-42), following the guidelines of Joshua Bloch. Note that the equality comparison – and consequently the hash value computation – only takes the `Name` and `Age` properties into account as they represent significant values. Two inferred invariants for type consistency of `Name` and `Age` have been omitted.

Listing 5.15b

C# translation

```

1  [Serializable]
2  public struct Person : IEquatable<Person>
3  {
4      public string Name { get; set; }
5      public int Age { get; set; }
6      public Colour FavouriteColour { get; set; }
7      public bool LikesDogs { get; set; }
8
9      public Person(string name,
10                     int age,
11                     Colour favouriteColour,
12                     bool likesDogs)
13      {
14          Name = name;
15          Age = age;
16          FavouriteColour = favouriteColour;
17          LikesDogs = likesDogs;
18      }
19
20      // Invariants omitted ...
21
22      public bool Equals(Person that)
23      {
24          if (ReferenceEquals(null, that)) return false;
25          if (ReferenceEquals(this, that)) return true;
26          return string.Equals(Name, that.Name)
27                 && Age == that.Age;
28      }
29
30      public override bool Equals(object that)
31      {
32          if (ReferenceEquals(this, that)) return true;
33          return that is Person && Equals((Person) that);
34      }

```

... *Continues on the next page ...*

... *Continued from the previous page ...*

```
35     public override int GetHashCode()
36     {
37         var result = 17;
38         result = 31 * result + Name.GetHashCode();
39         result = 31 * result + Age;
40         return result;
41     }
42 }
```

The .NET documentation suggests that a struct never exceeds sixteen bytes in size since it is allocated on the stack rather than the heap like objects of regular classes [50]. This is just enough to store a single decimal type or two object references on a 64-bit operating system.

Besides the manually defined constructor that initialises the fields of the struct, it always has a default, parameterless constructor, which is declared implicitly by C# [3]. It is used to allocate struct arrays and sets all fields to their default value, which is false for Booleans, 0 for numeric values and null for reference types. Since it is public, it is always possible to construct an instance of a struct that has default values for all fields – whether such an instance makes sense or not.

Because of the pass-by-value semantics exhibited by a struct, the .NET documentation recommends that a struct is immutable to avoid confusion about which particular object is being modified [50]. In fact, when an instance of a struct is assigned to another variable, it is copied shallowly. Thus, if the struct contains a field of a reference type, only the reference is copied while the referred object is shared between the original instance and the copy. This is problematic if the reference type is mutable which is the case for collections in .NET.

In VDM-SL, however, composite types – and state definitions in particular – are mutable and may contain fields of collection types. While instances of collections in VDM-SL are immutable, representing a composite type by a struct in C# violates the .NET guidelines.

One should also take care when dealing with hash values of mutable objects. The `GetHashCode` method includes all significant properties in its computation of the hash value; however, all properties are mutable which means that the hash value of the object may change as the properties are reassigned. Such a change is not anticipated by hash-based data structures, making their behaviour unreliable and in worst case incorrect [51].

5.5.2 Data classes

Rather than using structs to represent composite types, ordinary data classes may solve the task. By default, classes have referential equality and pass-by-reference semantics so that it is just the object references, not the objects themselves, that are copied on assignments to variables.

The overall translation from a composite type to a class is the same as for a struct. The normalised name of the class is UpperCamelCase, it is tagged with `[Serializable]` to enable object serialisation and it becomes a public member of the enclosing class – a *nested class* so to speak. Similarly, the fields of the composite type become public auto-property members of the class which are initialised from a constructor. Like the struct, it implements the `IEquatable<T>` interface and overrides the `Equals` and `GetHashCode` methods of `object` to provide structural equality.

It obtains pass-by-value semantics by implementing a `Copy` method which creates a duplicate of the class instance. The signature of `Copy` is defined in an `ICopyable<T>` interface provided by the runtime library of the transcompiler. The class is additionally marked with the `sealed` modifier to prevent inheritance. This is not necessary for structs, which are always sealed automatically in C# [3].



Example 5.15, continued

Composite types

Listing 5.15a showed a composite type definition `Person` with four fields. Listing 5.15c shows the equivalent C# class `Person` which implements the `ICopyable<Person>` and `IEquatable<Person>` interfaces. It contains the same members as the struct in Listing 5.15b (they are omitted) plus an implementation of the `Copy` method (lines 7-8) from the `ICopyable<Person>` interface.



Listing 5.15c

C# translation

```
1  [Serializable]
2  public sealed class Person : ICopyable<Person>,
3                                IEquatable<Person>
4  {
5      // Code omitted ...
6
7      public Person Copy()
8          => new Person(Name, Age, FavouriteColour, LikesDogs);
9  }
```

5.5.3 Rule summarisation

Since representing composite types by structs in C# violates the .NET guidelines, the preferred solution is to use data classes, as summarised by Rule 5.16.

	<p>Rule 5.16 Composite types</p> <p>A composite type in VDM-SL becomes a class in C# that implements the <code>ICopyable<T></code> and <code>IEquatable<T></code> interfaces, overrides the <code>Equals</code> and <code>GetHashCode</code> methods of <code>object</code> and is marked with the <code>sealed</code> modifier.</p> <p>If it is defined in a flat specification or is exported in a module, it is also marked with the <code>public</code> access modifier. Otherwise, it is marked with the <code>private</code> access modifier.</p> <p>The fields of the composite type become auto-property members of the class that are initialised in a constructor. Type invariants are translated accordingly by Rule 4.12.</p>
	<p>Limitations Rule 5.16</p> <p>The computation in <code>GetHashCode</code> includes mutable properties. Changes to the hash value may make hash-based data structures unstable.</p>

Java does not have a concept of structs so the Java code generator in *Overture* translates composite types to ordinary classes by a similar approach as outlined in Section 5.5.2. Instead of properties, it uses public fields complemented by getter and setter methods.

It overrides the `equals` and `hashCode` methods to provide structural equality; however, it compares all fields of the composite type and does not ignore equality abstraction fields. The implementation of `hashCode` does not follow the guidelines of Joshua Bloch. In particular, it uses 0 as a starting point and does not multiply the hash values of the individual fields by a prime number – this increases the probability of hash value collisions, which is not desirable [39]. The translation of Listing 5.15a to Java is shown in Listing 5.15d. Listing 5.15e shows the `Utils.hashCode` helper method.

Listing 5.15d

Java translation

```

1  // @ nullable_by_default
2  final public class Person implements Record, Serializable
3  {
4      public String name;
5      public Number age;
6      public Object favouriteColour;
7      public Boolean likesDogs;
8
9      public Person(final String _name,
10                  final Number _age,
11                  final Object _favouriteColour,
12                  final Boolean _likesDogs)
13      {
14          name = (_name != null) ? _name : null;
15          age = _age;
16          favouriteColour = (_favouriteColour != null)
17                          ? _favouriteColour
18                          : null;
19          likesDogs = _likesDogs;
20      }
21
22      /* @ pure */
23      public boolean equals(final Object obj)
24      {
25          if (!(obj instanceof Person)) return false;
26          Person other = ((Person) obj);
27
28          return (Utils.equals(name, other.name))
29              && (Utils.equals(age, other.age))
30              && (Utils.equals(favouriteColour,
31                              other.favouriteColour))
32              && (Utils.equals(likesDogs, other.likesDogs));
33      }
34
35      /* @ pure */
36      public int hashCode()
37      {
38          return Utils.hashCode(name, age,
39                              favouriteColour, likesDogs);
40      }
41
42      /* @ pure */
43      public Person copy()
44      {
45          return new Person(name, age,
46                              favouriteColour, likesDogs);
47      }
48
49      // Getters and setters omitted ...
50  }

```

Listing 5.15e

Utils.java

```

1 public static int hashCode(Object... fields)
2 {
3     int hashCode = 0;
4
5     for (int i = 0; i < fields.length; i++)
6     {
7         Object currentField = fields[i];
8         hashCode += currentField != null
9                     ? currentField.hashCode()
10                    : 0;
11     }
12
13     return hashCode;
14 }

```

5.6 Type aliases

VDM-SL allows type definitions that define other names – aliases – for types which have already been defined such that two different type aliases of the same underlying type may be used interchangeably. Usually, a named type definition is accompanied by a type invariant to restrict its set of legal values, as described in Section 5.7.



Example 5.17

Type aliases

Listing 5.17a shows four type aliases. *PositiveNumber* is an alias for the *nat1* type (line 2), *Count* is an alias for *PositiveNumber* (line 3), *Text* is an alias for *seq of char* (line 4) and *Optional Predicate* is an alias for an optional *bool* type (line 5).

Listing 5.17a

VDM-SL

```

1 types
2   PositiveNumber = nat1;
3   Count = PositiveNumber;
4   Text = seq of char;
5   Optional Predicate = [bool];

```


This section presents two approaches to translating type aliases in VDM-SL to using alias directives and data classes in C#. Furthermore, Section 5.6.4 discusses the treatment of self-referential types to ensure that they always terminate in C#.



Translation proposals

Section 5.6.1 describes the approach of using using alias directives in C# to represent type aliases.

Section 5.6.2 describes the approach of using data classes to represent type aliases.

5.6.1 using alias directives

C# supports type aliases through its construct of using alias directives, which were introduced in Section 3.2.1. It has the advantage that type aliases are automatically interchangeable; however, C# does not allow a using alias directive to be an alias of another alias, which is otherwise allowed in VDM-SL. Moreover, a using alias directive does not support invariant declarations by itself. Instead, instances of the aliased type must be checked accordingly by pre- and postconditions in methods and type invariants in properties to see if they are in the domain of the named type definition. Finally, a using alias directive only applies in the file it appears in. Therefore, it must be copied to every file in the translated output.



Example 5.17, continued

Type aliases

Listing 5.17a showed four type aliases. Listing 5.17b shows the corresponding using alias directives in C#. The using alias directive for Count (line 2) does not compile as it defines an alias of another alias.

Listing 5.17b

C# translation

```
1 using PositiveNumber = int;
2 using Count = PositiveNumber; // Does not compile.
3 using Text = string;
4 using Optional Predicate = bool?;
```

5.6.2 Data classes

Another approach is to generate a wrapper data class with a single read-only auto-property that holds the value of the underlying type being aliased. A suitable name for **the property is ‘Value’**.

This approach will allow invariants to be declared in the wrapper class and enforced automatically by the Code Contracts library. It is declared as an inner class of the enclosing top-level class, is named after the VDM-SL type alias and is marked with the sealed modifier to prevent inheritance. It implements the `ICopyable<T>` and `IEquatable<T>` interfaces and overrides the `Equals` and `GetHashCode` methods of `object` accordingly to provide structural equality and pass-by-value semantics as explained in Section 5.5.

Different wrapper classes of the same type are not interchangeable, though, so such an action requires a type conversion, for example a type cast. In C#, the type cast operator can be overloaded to increase the number of valid type casts [3]. By overloading the implicit type cast operator, the C# compiler will automatically cast the underlying value to an instance of the data class and vice versa. However, the data class needs to overload the type cast operator for every other type alias that has been defined for the same underlying type in order to make them interchangeable. This will easily lead to a combinatorial explosion of overloads, which is not desirable.

To keep the number of overloads down, the data class only overloads the type cast operators to and from the underlying type so that literals, for example integers, are automatically cast to instances of the data class. By leaving the `Value` property public, it is always possible to retrieve the aliased value.



Example 5.17, continued

Type aliases

Listing 5.17a showed four type aliases. Listing 5.17c shows the corresponding data class for `PositiveNumber` which is an alias for the `nat1` type in VDM-SL. As explained in Section 5.7, the `nat1` type becomes `int` in C# and leads to a type invariant. Therefore, the `Value` property is of type `int` (line 4) and the `PositiveNumber` class is protected by an invariant (lines 8-10). `PositiveNumber` defines two implicit type cast operators from and to `int` (lines 12-13 and 15-16, respectively).

The implementations of `Copy`, `IEquatable.Equals`, `object.Equals` and `GetHashCode` have been omitted. Since `value` is a reserved word in C#, the `value` parameter in the constructor is prefixed by an `@` character to become verbatim.

Listing 5.17c

C# translation

```

1 public sealed class PositiveNumber
2     : ICopyable<PositiveNumber>, IEquatable<PositiveNumber>
3 {
4     public int Value { get; }
5
6     public PositiveNumber(int @value) { Value = @value; }
7
8     [ContractInvariantMethod]
9     private void ObjectInvariant()
10         => Contract.Invariant(Value > 0);
11
12     public static implicit operator PositiveNumber(int that)
13         => new PositiveNumber(that);
14
15     public static implicit operator int(PositiveNumber that)
16         => that.Value;
17
18     // Code omitted ...
19 }

```

5.6.3 Rule summarisation

using alias directives are of limited use when it comes to type invariants and they do not support aliases of other aliases. While data classes are not completely interchangeable due to the omission of certain type cast operators, they support type invariants and can be aliases of each other. Therefore, they are the preferred approach for translating type aliases from VDM-SL to C#, as summarised by Rule 5.18.



Rule 5.18

Type aliases

A type alias in VDM-SL becomes a class in C# that implements the `ICopyable<T>` and `IEquatable<T>` interfaces, overrides the `Equals` and `GetHashCode` methods of `object` and is marked with the `sealed` modifier.

If it is defined in a flat specification or is exported in a module, it is also marked with the `public` access modifier. Otherwise, it is marked with the `private` access modifier.

It has a single read-only auto-property member named **‘Value’** that holds the aliased value and is initialised in a constructor. It also overloads the implicit type cast operator from and to the underlying aliased type. Type invariants are translated accordingly by Rule 4.12.


The Java code generator in Overture uses a third approach when dealing with type aliases: It inlines the type alias to obtain the underlying type and enforces its type invariants via the `assert` annotation of JML. Therefore, it does not produce any Java code for Listing 5.17a. Listing 5.17d shows the translation of a function `Alpha` that takes an instance of `PositiveNumber` as input.

Listing 5.17d	Java translation
<pre> 1 /*@ pure @*/ 2 public static Boolean Alpha(final Number n) 3 { 4 /*@ assert (Utils.is_nat1(n)); 5 // Code omitted ... 6 }</pre>	

5.6.4 Self-referential types

By inlining the type alias, the Java code generator in Overture faces a problem when dealing with self-referential types.⁷ Unwinding self-referential types in the JML `assert` annotation leads to infinite cycles. The Java code generator only checks the type consistency of the topmost level of the recursion [13].

In C#, the approach in Rule 5.18 breaks the chain of self-reference by introducing a class which can refer to itself by name; hence, it never ends up in an infinite cycle as the Java code generator does.

	Example 5.19	Recursive type aliases
<p>Listing 5.17a shows a type definition <code>Tree</code>, which is a union of <code>nat</code> and a pair of <code>Tree</code>, corresponding to a binary tree structure. Listing 5.19b shows a fragment of the resulting C# class. The invariant breaks the chain of self-reference by referring to the <code>Tree</code> class itself (lines 6-8).</p>		
Listing 5.19a	VDM-SL	
<pre> 1 types 2 Tree = nat Tree * Tree;</pre>		

⁷ Self-referential types are also known as *recursive types*.

Listing 5.19b

C# translation

```

1 public sealed class Tree : ICopyable<Tree>, IEquatable<Tree>
2 {
3     public object Value { get; }
4
5     [ContractInvariantMethod]
6     private void ObjectInvariant()
7         => Contract.Invariant((Value is int && Value >= 0)
8                               || Value is Tuple<Tree, Tree>);
9
10    // Code omitted ...
11 }

```

5.7 Type invariants

Just like a state definition, a type definition can be protected by an invariant in VDM-SL. It constrains the set of legal values accepted by the type. This section presents the C# translation of type invariants.



Example 5.20

Type invariants

Listing 5.20a shows a value definition `Magi cConstant` of type `nat1` with the value 42 (line 2), two type definitions `TwoDi gi tNumber` (lines 5-6) and `Pari ty` (line 8) and a function definition `Pari tyOf` (lines 11-12). `TwoDi gi tNumber` is an alias for the `nat1` type and is protected by a type invariant (line 6) that restricts its set of legal values to the numbers between 10 and 99. `Pari ty` is a union of two quotes `Even` and `Odd`. `Pari tyOf` takes an instance of `TwoDi gi tNumber` as input and determines whether it represents an even number or an odd number.

Listing 5.20a

VDM-SL

```

1 values
2     Magi cConstant: nat1 = 42;
3
4 types
5     TwoDi gi tNumber = nat1
6     i nv n == n >= 10 and n <= 99;
7
8     Pari ty = <Even> | <Odd>;
9
10 functions
11     Pari tyOf: TwoDi gi tNumber -> Pari ty
12     Pari tyOf(n) == i f n rem 2 = 0 then <Even> el se <Odd>;

```

Some of the built-in types in VDM-SL have special constraints; for example, the `nat1` type represents the set of positive natural numbers. A reasonable equivalent type in .NET is `uint`, which represents unsigned integers including zero. However, `uint` is not compliant with the Common Language Specification and the documentation of .NET recommends using the ordinary `int` type, which represents signed integers, whenever possible [52]. To enforce the constraint of positivity, `nat1` is thus expressed as a signed integer with a type invariant that excludes all non-positive values from its set of legal values.

Type invariants must be enforced when dealing with a typed expression. In particular, there are enforced in five places: parameters to functions and operations, result values from functions and operations, variable initialisers, assignments and value definitions [13].

Type invariants for method parameters are enforced as preconditions, that is, by calling `Contract.Requires` and checking that the type invariant predicate is satisfied. Type invariants for return values are enforced similarly as postconditions via `Contract.Ensures`.

All persistent state is stored in properties which are subject to invariant enforcement through the calls to `Contract.Invariant` carried out by the nested class declarations, see Section 4.4. Temporary state in VDM-SL can be defined through the use of variables initialised in `let`-expressions and `define`-expressions. They are equivalent to local variable declarations in C#. Potential type invariants are enforced by calling the `Contract.Assert` method in .NET Code Contracts [9]. It takes a predicate as input and evaluates it immediately.

By Rule 4.16, values become static read-only properties that are initialised right away. Therefore, type invariants on values only need to be enforced once. This is done by calling `Contract.Assert` from a static constructor in the class that represents the enclosing VDM-SL specification. The property initialiser is always executed just before the static constructor [3].



Example 5.20, continued

Type invariants

Listing 5.20a showed a value definition `Magi cConstant`, two type definitions `TwoDi gi tNumber` and `Pari ty` and a function definition `Pari tyOf`. Listing 5.20b shows the corresponding C# translation. Since `Magi cConstant` is of type `nat1`, it becomes `int` in C# (line 6) with a type invariant that is enforced from a static constructor (lines 1-4). The `TwoDi gi tNumber` class has two type invariants: one from `nat1` (line 18) and one from the `inv` clause (lines 19 and 22-27). The `Pari tyOf` method is guarded by a precondition which checks that the `TwoDi gi tNumber` input is not null (line 35).

Listing 5.20b

C# translation

```

1  static Global ()
2  {
3      Contract.Assert(MagicConstant > 0);
4  }
5
6  public static int MagicConstant { get; } = 42;
7
8  public sealed class TwoDigitNumber
9      : ICopyable<TwoDigitNumber>, IEquatable<TwoDigitNumber>
10 {
11     public int Value { get; }
12
13     public TwoDigitNumber(int @value) { Value = @value; }
14
15     [ContractInvariantMethod]
16     private void ObjectInvariant()
17     {
18         Contract.Invariant(Value > 0);
19         Contract.Invariant(InvTwoDigitNumber(Value));
20     }
21
22     [Pure]
23     public static bool InvTwoDigitNumber(int n)
24     {
25         Contract.Requires(n > 0);
26         return n >= 10 && n <= 99;
27     }
28
29     // Code omitted ...
30 }
31
32 [Pure]
33 public static Parity ParityOf(TwoDigitNumber n)
34 {
35     Contract.Requires(n != null);
36     return n % 2 == 0 ? Parity.Even : Parity.Odd;
37 }

```

Rule 5.21 summarises the translation of type invariants in VDM-SL to code contracts in C#.



Rule 5.21

Type invariants

A type invariant on a parameter of a function or operation becomes a precondition in C# by Contract. *Requires*.

A type invariant on a result value of a function or operation becomes a postcondition in C# by Contract. *Ensures*.

A type invariant on a state field, a record field or the value of a type alias becomes an invariant in C# by Contract. *Invariant*.

A type invariant on a local variable definition becomes an assertion in C# by Contract. *Assert*.

A type invariant on a value becomes an assertion in C# by Contract. *Assert*, enforced from a static constructor.

The Java code generator in Overture applies the *invariant* annotation of JML for type invariants on persistent state. It uses the *assert* annotation of JML in the remaining four cases. The Java translation of Listing 5.20a is shown in Listing 5.20c.

Listing 5.20c

Java translation

```

1  //@ public static invariant Utils.is_nat1(MagicConstant);
2  public static final Number MagicConstant = 42L;
3
4  /*@ pure */
5  public static Object ParityOf(final Number n)
6  {
7      //@ assert (Utils.is_nat1(n)
8                  && inv_DEFAULT_TwoDigitNumber(n));
9      if (Utils.equals(Utils.rem(n.longValue(), 2L), 0L))
10     {
11         Object ret_1 = EvenQuote.getInstance();
12         //@ assert (Utils.is_(ret_1, EvenQuote.class)
13                     || Utils.is_(ret_1, OddQuote.class));
14         return ret_1;
15     }
16     else
17     {
18         Object ret_2 = OddQuote.getInstance();
19         //@ assert (Utils.is_(ret_2, EvenQuote.class)
20                     || Utils.is_(ret_2, OddQuote.class));
21         return ret_2;
22     }
23 }

```

... *Continues on the next page ...*


```
... Continued from the previous page ...  
23  /*@ pure @*/  
24  /*@ helper @*/  
25  public static Boolean inv_DEFAULT_TwoDigitNumber(  
26      final Object check_n)  
27  {  
28      Number n = ((Number) check_n);  
29      Boolean andResult_1 = false;  
30  
31      if (n.LongValue() >= 10L)  
32      {  
33          if (n.LongValue() <= 99L)  
34          {  
35              andResult_1 = true;  
36          }  
37      }  
38  
39      return andResult_1;  
40  }
```

This concludes the three chapters on translation rules.

“ Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

– Edsger W. Dijkstra (1930-2002)

6 Realisation

Having presented the guidelines for translating VDM-SL to C#, it is time to put them to the test! The translation rules have been implemented in a proof-of-concept transcompiler that builds upon the code generation platform of Overture. It has been developed with test-driven development to explore the possibilities of unit testing a compiler and evaluating the quality of the prototype.

This chapter presents the prototype implementation of the translation rules realised in this thesis. It also covers techniques to evaluate the quality of the prototype and discusses their suitability in a transcompiler setting.



Chapter contents

Section 6.1 gives an overview of the prototype implementation, which is built on top of the code generation platform of the Overture tool.

Section 6.2 describes the testing techniques applied for assessing the correctness of the prototype and discusses the use of automated testing on a syntactical level for compiler construction.



The source code and test suites for the prototype are available on GitHub:
<https://github.com/SPDiswal/VdmSL-to-Cs/tree/pvj/main/core/codegen/csgen>

6.1 Prototype implementation

A prototype of the VDM-SL-to-C# transcompiler has been implemented on top of the code generation platform provided by the Overture tool. It is written in the Kotlin programming language [16], which targets the Java platform and interoperates smoothly with the Java programming language. This is a requirement since Overture – being a variant of the Eclipse IDE – is written in Java.

Like C# and unlike Java, Kotlin is a purely object-oriented language with support for properties and extension methods. It offers strong type inference with null-safety and syntactical sugar for conditional statements, lambda expressions and data classes, making the code concise while retaining readability. The language is complemented by its compiler, which uses static program analysis to automatically cast variables after checking their type in a conditional statement – a concept, which is termed *smart casts*. All these language features make Kotlin suitable for implementing and unit testing a transcompiler.

6.1.1 Overture integration

The common approach to implementing a code generator in Overture is to transform the VDM-SL IR tree to make it resemble the syntax of the target language and then format it into code via Apache Velocity templates, as shown in Figure 6.1.

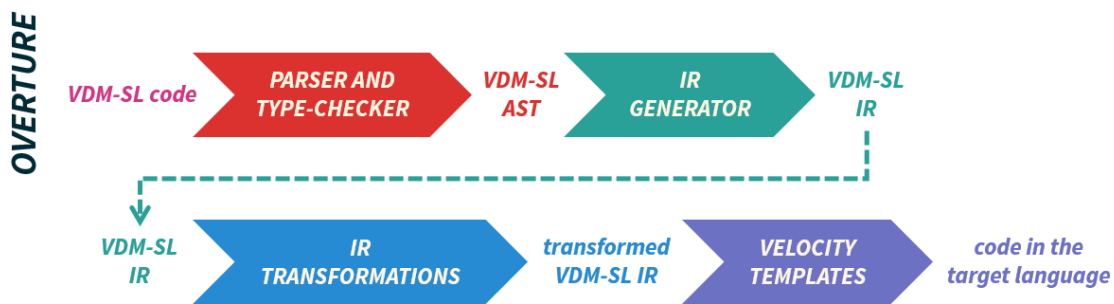


Figure 6.1

The common code generation process in Overture.

The prototype does not follow this approach for these reasons:

- The IR specification is biased towards Java and fails to capture certain C# and .NET concepts such as auto-properties and method attributes. This information must then be included as metadata on the IR nodes, sacrificing consistency and type safety.
- Enhancing the IR specification of AstCreator to take the necessary constructs into account has unfortunate consequences because it will require modifications to the IR generator and affect all code generators – in best case, it requires a recompilation of the affected modules in Overture due to tight class coupling; in worst case, it breaks the existing code either on compile-time or runtime as the definitions and behaviours of the IR visitors have changed.

- An IR transformation is implemented as an instance of the Visitor design pattern, taking an IR tree as input and manipulating this instance directly. It is difficult to unit test since the resulting IR tree must be inspected to see if it satisfies the expectations of the unit test. The IR nodes in Overture do not have sufficient implementations of the `equals` method to enable straightforward comparisons of trees.⁸ Instead, all fields of the IR nodes must be compared individually, which is a very tedious process due to the large number of different kinds of IR nodes.
- The use of Apache Velocity leads to fragmentation of the prototype code since C#-formatting code is scattered across Velocity templates, helper classes for Velocity and IR transformations.
- It is not possible to unit test the output from Apache Velocity since it requires running the template engine, which increases the complexity of the test by a magnitude so extensive that it becomes an integration test rather than a unit test. Hence, it tests more than just the translation from VDM-SL to C# – it also tests the quality of the template engine in Apache Velocity and will fail if it hits a bug here, even though the logic behind the translation rule has been implemented correctly.

Instead, the prototype has been designed to operate independently from the code generation platform of Overture, apart from requiring an IR tree as input. It translates this IR tree to an AST of the corresponding C# program and formats this AST into C# source code, as shown in Figure 6.2. It still utilises the existing VDM-SL parser, type-checker and IR generator in Overture.

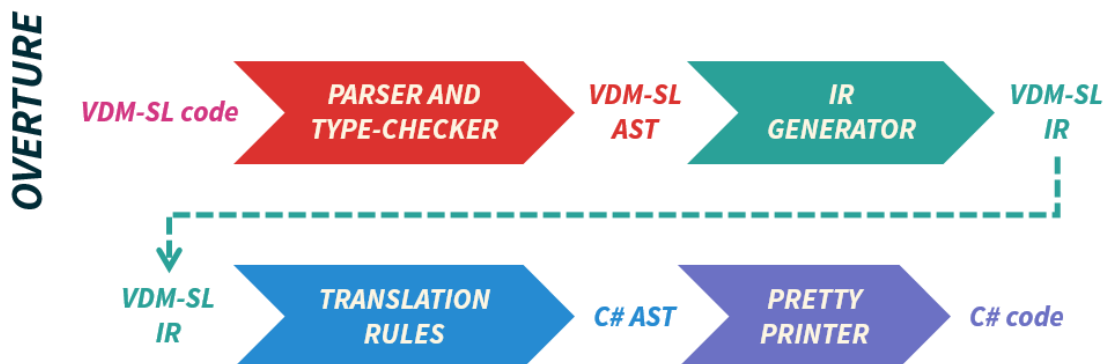


Figure 6.2

The code generation process employed by the prototype.

⁸ The `equals` method simply compares the string representation – via the `toString` method – of two IR nodes to see if they are equal. However, the string representation may fail to include all significant fields of an IR node. For instance, the string representation of `ARatNumeri cBasi cTypeI R` nodes fails to include the `opti onal` field that indicates whether it represents an optional type or a non-optional type.

The main advantage of this approach is the unit testability of the C# AST nodes which are defined in Kotlin using data classes; hence, they are already equipped with sufficient implementations of the `equals` method. Now, each translation rule can be tested in isolation.

6.1.2 Translation rules

The prototype utilises the existing VDM-SL parser, type-checker and IR generator of Overture so that its only focus is to translate VDM-SL to C#. Thus, it implements the backend of the transcompiler.

Every translation rule is reflected in the prototype as a mapping from a VDM-SL IR node to a C# AST node. It replaces one or more IR transformations from the common code generator structure. VDM-SL IR nodes implement the `PIR` interface defined by the code generation platform of Overture, whereas C# AST nodes implement the `CsNode` interface defined in the prototype.

All translation rules in the prototype implement the `translate` method of the generic `TranslationRule<TIRNode, TCsNode>` interface where `TIRNode` and `TCsNode` are type parameters constrained to be subtypes of `PIR` and `CsNode`, respectively. The `translate` method takes an instance of `PIR` as input and returns an instance of `CsNode`, reflecting the translation from VDM-SL to C# as shown in Figure 6.3.

The final output of the transcompiler is generated by a pretty-printing component, `CsFormatter`, which traverses the C# AST and transforms it into formatted C# source code. It replaces Apache Velocity from the common code generator structure and focuses solely on formatting the C# AST into source code.

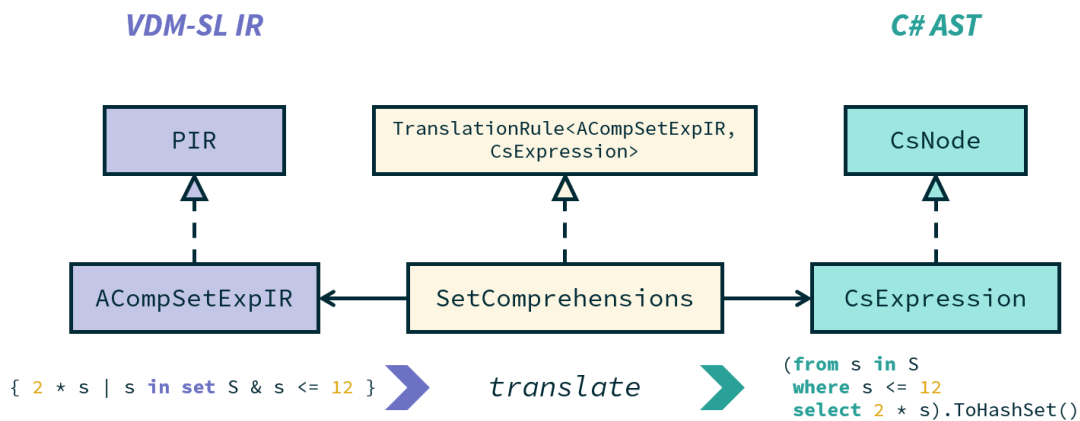


Figure 6.3

The `SetComprehensions` translation rule takes an instance of `ACompSetExpIR` as input and produces a corresponding instance of `CsExpression`.

6.1.3 Transcompiler runtime

Some of the translation rules make use of helper methods that simplify the translations although they have been kept to a minimum by utilising the built-in features of the .NET platform instead. Among the helper methods is the `ToHashSet` extension method for set manipulation in LINQ.

In order to work correctly, the C# output must import these helper methods from a transcompiler-specific runtime, which the transcompiler bundles as a .NET library along with the C# output. The particular components in the transcompiler runtime are described in Appendix B.

6.2 Automated testing

The prototype has been developed using test-driven development in which automated tests have driven the implementation process. This section discusses four approaches to testing the code generating component of the transcompiler on a syntactical level. In the first approach, the unit tests specify what they expect the C# AST to look like, given a VDM-SL IR tree. The three other approaches specify their expectations in terms of the raw text of the C# output, but use different means to conclude whether the C# output is satisfactory. After presenting the four testing approaches, the testing approach applied for implementing the prototype is outlined.

The JUnit-based Spek framework of Kotlin has been used to assist the testing effort [53]. It is a so-called *specification framework* which prescribes that unit tests must follow the ARRANGE-ACT-ASSERT pattern [33]. All test input is arranged within the given method, the unit under test is acted upon within the `on` method and expectations are checked individually by calling `assertThat` within the `it` method. In this case, the units under test are the translation rules of the prototype implementation.

The Arrange-Act-Assert pattern ensures that a unit test only exercises one thing in the software system and runs isolated from the other unit tests. Otherwise, an ill-fated unit test may affect the results of the other tests, which is not desirable.

6.2.1 Tree comparison

A translation rule takes a VDM-SL IR tree as input and outputs a C# AST. It is therefore simple to define the test input in terms of IR nodes and the test expectations in terms of C# nodes.

Example 6.4

Tree comparison

Listing 6.4a shows a Spek specification that tests the translation from a set comprehension IR to a C# AST. The IR is defined as test input within the `given` method (lines 3-11) and translated to a C# AST within the `on` method via the `SetComprehensions` object (line 15), which is the unit under test. The C# AST is expected to be a LINQ query expression, which is specified and asserted upon within the `it` method (lines 19-32).

Listing 6.4a

Spek in Kotlin

```

1  given("an IR node of a set comprehension")
2  {
3      // { 2 * s | s in set S & s > 1 }
4      val ir = vdmSetComprehension(
5          bindings = listOf(
6              vdmSetBinding(patterns = listOf(vdmSId("s")),
7                  set = vdmSId("S"))
8          ),
9          predicate = vdmSId("s") > 1,
10         projection = 2 * vdmSId("s")
11     )
12
13     on("transcompilation")
14     {
15         val actualCsAst = SetComprehensions.translate(ir)
16
17         it("is a LINQ query expression")
18         {
19             // (from s in S
20             //   where s > 1
21             //   select 2 * s).ToHashSet()
22             val expectedCsAst = csCall(
23                 receiver = csQuery(
24                     fromClauses = listOf(("s") to csId("S")),
25                     whereClause = csId("s") > 1,
26                     selectClause = 2 * csId("s")
27                 ),
28                 member = csId("ToHashSet"),
29                 arguments = emptyList()
30             )
31
32             assertThat(actualCsAst, isEqualTo(expectedCsAst))
33         }
34     }
35 }

```


Specifying a tree structure such as an IR tree or a C# AST quickly becomes tedious, though, because every single node and subtree must be included in order to construct an exact tree. Otherwise, the test input is not an authentic representation of the IR trees that occur in a realistic setting. Additionally, it implies a penalty in developer productivity since specifying an exact tree is bothersome and error-prone. Having two different trees in the unit tests makes the test code difficult to read, which is a problem as the test code becomes harder for the developer to analyse; and hence, bugs in the *unit tests* are more likely to appear [10].

6.2.2 Raw text comparison

The opposite approach to specifying tree structures is to write the test input and expectations in terms of raw text; in this case, the raw text comprises fragments of VDM-SL code and C# code.



Example 6.5

Raw text comparison

Listing 6.5a shows a Spek specification that tests the translation from a set comprehension IR to a C# AST. The test input is a fragment of VDM-SL code (lines 3-4) that is parsed by an `Expressi onReader` object in `Overture` (lines 8-10), augmented with type information by an `Expressi onTypeChecker` object in `Overture` (lines 11-12), generated to an IR tree by an `IRGenerator` object in the code generation platform of `Overture` (line 13) and finally translated and formatted to a fragment of C# code (lines 14-15). It is compared to the expected C# code fragment (lines 18-24).

Listing 6.5a

Spek in Kotlin

```

1 given("an IR node of a set comprehensi on")
2 {
3     val vdmSI =
4         "{ 2 * s | s in set S & s > 1 }"
5
6     on("transcompilati on")
7     {
8         val vdmSI Ast = Expressi onReader(
9             LexTokenReader(vdmSI , Di al ect. VDM_SL)
10        ). readSetExpressi on()
11        val typedVdmSI Ast = Expressi onTypeChecker(vdmSI Ast)
12            . typeCheck()
13        val ir = IRGenerator(). generateFrom(typedVdmSI Ast)
14        val actual CsAst = SetComprehensi ons. transl ate(ir)
15        val actual Cs = CsFormatter. format(actual CsAst)

```

... *Continues on the next page ...*

```

... Continued from the previous page ...
16     it("is a LINQ query expression")
17     {
18         val expectedCs = ""
19         (from s in S
20          where s > 1
21          select 2 * s).ToHashSet()
22         ""'.trimIndent()
23
24         assertEquals(actualCs, isEqualTo(expectedCs))
25     }
26 }
27 }

```

The approach with raw text comparison is shorter than the tree-comparing approach. **Unit tests are easier to write from a developer's point of view** since the test input and expectations are code fragments rather than tree structures.

Nevertheless, the unit tests depend on the parser, type-checker and IR generator in Overture in order to compile the VDM-SL code fragment to an IR tree. Furthermore, they depend on the C# pretty-printer to format the C# AST into a C# code fragment. This increases their complexity, making them resemble integration tests more than actual unit tests. In fact, the tests may fail due to errors in any of these components even though the tests are logically correct.

Raw text comparison is very fragile because the slightest change in the code fragments, for example a whitespace character or letter casing, may render the test a failure. Spek does not provide any sophisticated means for tracking differences in the actual and expected code fragments; therefore, it can be very difficult for the developer to track down a small, insignificant difference.

6.2.3 Regular expressions

Instead of comparing the code fragments directly, they can be compared part-for-part by regular expressions. The test input remains a VDM-SL code fragment to be compiled to an IR tree; however, the unit test consists of multiple expectations. Each expectation is only responsible for checking a small part of the C# code fragment, which is extracted from a regular expression.



Example 6.6

Regular expressions

Listing 6.6a shows a Spek specification that tests the translation from a set comprehension IR to a C# AST. The test input is a fragment of VDM-SL code (lines 3-4) that is compiled to an IR tree by Overture (lines 8-13) and finally translated and formatted to a fragment of C# code (lines 14-15). It is compared to the expected C# code fragment using four expectations.

Using the `matches` method for matching text strings to regular expressions, the first expectation checks that the C# output contains a `from...in` clause (lines 17-21), the second expectations checks that it contains a `where` clause (lines 23-27), the third expectation checks that it contains a `select` clause (lines 29-33) and the fourth expectation checks that it calls the `ToHashSet` method (lines 34-38).

Listing 6.6a

Spek in Kotlin

```

1  given("an IR node of a set comprehensi on")
2  {
3      val vdmSI =
4          "{ 2 * x | s in set S & s > 1}"
5
6      on("transcompilati on")
7      {
8          val vdmSI Ast = Expressi onReader(
9              LexTokenReader(vdmSI , Di al ect. VDM_SL)
10             ). readSetExpressi on()
11          val typedVdmSI Ast = Expressi onTypeChecker(vdmSI Ast)
12              . typeCheck()
13          val ir = IrGenerator(). generateFrom(typedVdmSI Ast)
14          val actual CsAst = SetComprehensi ons. transl ate(ir)
15          val actual Cs = CsFormatter. format(actual CsAst)
16
17          it("has a from...in clause")
18          {
19              val expectedCs = ". *\\Qfrom s in S\\E. *"
20              assertThat(actual Cs, matches(expectedCs))
21          }
22
23          it("has a where clause")
24          {
25              val expectedCs = ". *\\Qwhere s > 1\\E. *"
26              assertThat(actual Cs, matches(expectedCs))
27          }
28
29          it("has a select clause")
30          {
31              val expectedCs = ". *\\Qselect 2 \\* s\\E. *"
32              assertThat(actual Cs, matches(expectedCs))
33          }

```

... *Continues on the next page ...*

```

... Continued from the previous page ...
34         it("call is ToHashSet")
35         {
36             val expectedCs = "\\(. *\\)\\QToHashSet\\(\\)\\E"
37             assertThat(actual Cs, matches(expectedCs))
38         }
39     }
40 }

```

The benefit of splitting the comparison into multiple smaller expectations is that it becomes much clearer what part of the code fragment is incorrect since only that particular expectation will fail. Regular expressions allow the expectations to ignore whitespace characters and focus only on their part of the code fragment.

This approach comes with a price, though. Complex regular expressions tend to be difficult to both read and error-prone to write, which implies a penalty in developer productivity.

More importantly, an expectation only checks whether a certain piece of text is present or absent in the code fragment. However, when it is present, the expectation does not check for duplicate occurrences. Therefore, it does not provide any guarantees that the output really looks like intended.

6.2.4 Parsing

The text-comparing approaches can be combined into a solution that is robust against insignificant differences in the output, takes duplicate occurrences into account and is easy to specify.

By parsing the expected C# code fragment into a C# AST, it can be compared to the actual C# AST without needing to pretty-print the actual code fragment.



Example 6.7

Parsing

Listing 6.7a shows a Spek specification that tests the translation from a set comprehension IR to a C# AST. The test input is a fragment of VDM-SL code (lines 3-4) that is compiled to an IR tree by Overture (lines 8-13) and translated to a C# AST (lines 15-16) of type `CsCall Expression`. The expected C# code fragment is parsed into a C# AST of type `CsCall Expression` as well (lines 21-25). The subtrees of both ASTs are compared to each other using four expectations (lines 30-56).

Listing 6.7a

Spek in Kotlin

```

1 given("an IR node of a set comprehensi on")
2 {
3     val vdmSI =
4         "{ 2 * x | s in set S & s > 1 }"
5
6     on("transcompilati on")
7     {
8         val vdmSI Ast = Expressi onReader(
9             LexTokenReader(vdmSI, Di al ect. VDM_SL)
10        ). readSetExpressi on()
11        val typedVdmSI Ast = Expressi onTypeChecker(vdmSI Ast)
12            . typeCheck()
13        val i r = IrGenerator(). generateFrom(typedVdmSI Ast)
14
15        val actual CsAst = SetComprehensi ons. transl ate(i r)
16            as CsCal l Expressi on
17
18        val actual CsQueryAst = actual CsAst. recei ver
19            as CsQueryExpressi on
20
21        val expectedCsAst = CsParser. parse("""
22            (from s in S
23              where s > 1
24              select 2 * s). ToHashSet()
25        """, rule = CsParser. cal l Expressi on)
26
27        val expectedCsQueryAst = expectedCsAst. recei ver
28            as CsQueryExpressi on
29
30        it("has a from...in cl ause")
31        {
32            assertThat(
33                actual CsQueryAst. fromCl ause,
34                i sEqual To(expectedCsQueryAst. fromCl ause)
35            )
36        }
37
38        it("has a where cl ause")
39        {
40            assertThat(
41                actual CsQueryAst. whereCl ause,
42                i sEqual To(expectedCsQueryAst. whereCl ause)
43            )
44        }

```

... *Continues on the next page ...*

... *Continued from the previous page ...*

```

45         it("has a select clause")
46         {
47             assertThat(
48                 actual CsQueryAst. selectCl ause,
49                 isEqual To(expectedCsQueryAst. selectCl ause)
50             )
51         }
52
53         it("calls ToHashSet")
54         {
55             assertThat(actual CsAst, isEqual To(expectedCsAst))
56         }
57     }
58 }

```

The unit test maintains its readability by specifying the VDM-SL and C# code fragments in their respective languages. It maintains its robustness by translating the VDM-SL code fragment and parsing the expected C# code fragment into C# ASTs. In case the expected C# code fragment is syntactically incorrect, that is, the test is buggy, the error will be caught immediately by the parser.

Still, the C# parser does not come for free. It has to be implemented along with the translation rules being tested. Under complete practice of test-driven development, the parser must be unit tested, too. This implies a large productivity penalty for the developer; nonetheless, having a C# parser in the toolbox is useful not only for unit testing, but for the production code as well.

An advantage of implementing the C# parser is the inevitable definition of the C# AST structure that comes along. It can be used directly by the translation rules. Another advantage of having a C# parser is the fact that it can be employed by the translation rules to obtain readable code. Instead of specifying the exact tree structures to generate, they can parse the corresponding C# code.

6.2.5 Applied hybrid testing approach

The actual unit tests for the prototype follow a hybrid scheme between the tree-comparing approach in Section 6.2.1 and the parsing approach in 6.2.4.

The parser and type-checker in Overture suffer from fragmentation and are not easily applicable in practice due to essential methods being private and therefore not accessible from the tests. The test input is specified as VDM-SL IR trees.

The expectations are specified as C# ASTs complemented by parsed fragments of C# statements and expressions. Declarations such as classes, methods and properties are specified as AST nodes. Kotlin has a very concise way to express the BUILDER pattern [31] via syntactical sugar for lambda expressions [54]. It is useful for defining tree structures via test input builders [55] which is the case for C# declarations.

The C# parser is implemented as a parser combinator using the CakeParse library for Kotlin [56]. It has been manually tested, though, to save development efforts and only support a minimal subset of the C# language in order to work with the transcompiler prototype.

Unit testing the transcompiler gives rise to a large number of similar tests with similarly looking expectations. This is undesirable due to the MULTIPLE MAINTENANCE PROBLEM [10]. Therefore, a framework has been developed on top of Spek to accommodate the unit tests in a sleek manner. Since all unit tests follow the same structure of translating an IR tree to a C# AST, the Arrange-Act-Assert pattern is hidden from the test code so that only the input and expected output is specified. The test expectations on the C# AST are automatically discovered for each test by analysing the structure of the tree.



Example 6.8

Hybrid approach

Listing 6.5a shows a Spek specification that tests the translation from a set comprehension IR to a C# AST. The test input is a VDM-SL IR tree (lines 1-8) and the expected output is a C# code fragment (lines 9-11), which is parsed behind the scenes to a C# AST.

Listing 6.8a

Spek in Kotlin

```

1  vdmSI SetComprehension(
2      bindings = listOf(
3          vdmSI SetBinding(patterns = listOf(vdmSI Id("s")),
4              set = vdmSI Id("S"))
5      ),
6      predicate = vdmSI Id("s") > 1,
7      projection = 2 * vdmSI Id("s")
8  ) becomes """
9      (from s in S
10         where s > 1
11         select 2 * s). ToHashSet()
12  """

```


“ There never was a good knife made of bad steel.
– Benjamin Franklin (1706-1790)

7 Results

Implementing the prototype is only the first half of the story. The second half is to observe how well it fares when it is confronted with real VDM-SL specifications and not just small code fragments in unit tests.

This chapter presents a case study that evaluates the quality of the C# code generated by the transcompiler prototype for a VDM-SL specification that models an automated teller machine. It also benchmarks the performances of .NET Code Contracts and OpenJML for a VDM-SL specification that models an algorithm for obfuscating financial accounting district codes.



Chapter contents

Section 7.1 presents a case study for evaluating the quality of the prototype and observing the translation rules in action.

Section 7.2 benchmarks .NET Code Contracts and OpenJML and compares their computational performance.

7.1 An automated teller machine in VDM-SL

The VDM-SL specification for an automated teller machine (ATM) is used by Jørgensen et al. [13] for enhancing the Java code generator in Overture with support for JML. Hence, it focuses on design-by-contract elements.

7.1.1 Types

The ATM model defines types for representing bank accounts, payment cards, PINs and amounts of money, as shown in Listing 7.1a.

Account is a composite type that contains a set of payment cards and has a balance. It is protected by an invariant proclaiming that the balance must be at least -1000.

Card is composite type that has an ID and a PIN. PI N is an alias for the nat type whose values are constrained to be within 0 and 9999.

Amount is an alias for the nat1 type whose values must be below 2000.

Listing 7.1a

VDM-SL

```

1 types
2   Account :: cards: set of Card
3             balance: real
4   inv a == a.balance >= -1000;
5
6   Card :: id: nat
7           pin: Pin;
8
9   Pin = nat
10  inv p == 0 <= p and p <= 9999;
11
12  Amount = nat1
13  inv a == a < 2000;
```

Type definitions in VDM-SL are translated to data classes in C#. Composite types like Account and Card become data classes by Rule 5.16; type aliases like Pin and Amount become data classes by Rule 5.18. Listing 7.1b shows the C# translation of Account.

Listing 7.1b

C# translation

```

1 public sealed class Account
2   : ICopyable<Account>, IEquatable<Account>
3 {
4   public HashSet<Card> Cards { get; set; }
5
6   public decimal Balance { get; set; }
7
8   public Account(HashSet<Card> cards, decimal balance)
9   {
10      Cards = cards;
11      Balance = balance;
12   }
```

... *Continues on the next page ...*

```
... Continued from the previous page ...

13 [ContractInvariantMethod]
14 private void ObjectInvariant()
15 {
16     Contract.Invariant(
17         Cards != null
18         && Contract.ForAll(Cards, _ => _ != null));
19     Contract.Invariant(InvAccount(this));
20 }
21
22 [Pure]
23 public static bool InvAccount(Account a)
24 {
25     Contract.Requires(a != null);
26     return a.Balance >= -1000;
27 }
28
29 public Account Copy() => new Account(Cards.Copy(),
30                                     Balance);
31
32 public bool Equals(Account that)
33 {
34     if (ReferenceEquals(null, that)) return false;
35     if (ReferenceEquals(this, that)) return true;
36     return Cards.SetEquals(that.Cards)
37         && Balance == that.Balance;
38 }
39
40 public override bool Equals(object that)
41 {
42     if (ReferenceEquals(this, that)) return true;
43     return that is Account && Equals((Account) that);
44 }
45
46 public override int GetHashCode()
47 {
48     var result = 17;
49     result = 31 * result + Cards.GetHashCode();
50     result = 31 * result + Balance.GetHashCode();
51     return result;
52 }
53 }
```

In addition to the custom invariant on `Account`, the transcompiler has inferred invariants on the `Cards` property, which originates from the `cards` field of type `set of Card`. Specifically, it must be non-null and all of its `Card` items must be non-null as well. This is a consequence of the fact that reference types in C# are always nullable, unlike composite types in VDM-SL.

The Arel `invariantsEnabled` property has not been implemented and is therefore not present in `Contract.Invariant`. The `Equals` method takes the structural equality of the `Cards` property into account by calling `SetEquals` to compare two `Cards` properties.

7.1.2 State

The state of the ATM knows the set of valid payment cards and bank accounts and keeps track of any current sessions. To begin with, there are no payment cards or bank accounts registered. To remain consistent, it is protected by an invariant proclaiming that a card belongs to at most a single account. Listing 7.2a shows the state definition.

Listing 7.2a

VDM-SL

```

1 state St of
2   validCards: set of Card
3   currentCard: [Card]
4   pinOk: bool
5   accounts: map AccountId to Account
6 init St == St = mk_St({ }, nil, false, { |-> })
7
8 inv mk_St(v, c, p, a) ==
9   (p or c <> nil => c in set v) and
10  forall id1, id2 in set dom a &
11    id1 <> id2 => a(id1).cards inter a(id2).cards = { }
12 end

```

The translation rules do not support the pattern matching constructs of VDM-SL (line 9 in Listing 7.2a), which provides a fast way to decompose a composite type into its field values. C# does not have any pattern matching constructs and must access the values of the state fields by reading the properties on the state object individually. Listing 7.3b shows the same invariant in VDM-SL expressed without decomposing the composite type by pattern matching.

Listing 7.3b

VDM-SL

```

1 inv s ==
2   (s.pinOk or s.currentCard <> nil =>
3     s.currentCard in set s.validCards) and
4   forall id1, id2 in set dom s.accounts &
5     id1 <> id2 => s.accounts(id1).cards inter
6     s.accounts(id2).cards = { }

```

This translation is supported by Rule 4.11, as shown in Listing 7.2c. The `implies` operator in VDM-SL is rewritten as outlined in Section 4.4.1 and the universal quantification becomes a LINQ query by Rule 5.8. The `ToHashSet` call just before `Any` (line 13 in Listing 7.2c) is redundant since `Any` is an extension method on `IEnumerable<Card>`, which is the return type of the `Intersect` method.

Listing 7.2c C# translation

```

1  [Pure]
2  public static bool InvSt(St s)
3  {
4      Contract.Requires(s != null);
5
6      return (! (s.PinOk || !object.Equals(s.CurrentCard, null))
7              || s.ValidCards.Contains(s.CurrentCard))
8              && (from id1 in s.Accounts.Keys.ToHashSet()
9                  from id2 in s.Accounts.Keys.ToHashSet()
10                 select !(id1 != id2)
11                      || !s.Accounts[id1].Cards.Intersect(
12                        s.Accounts[id2].Cards)
13                      .ToHashSet().Any()).All(_ => _);
14  }
```

7.1.3 Operations

The ATM model supports many different operations. For example, `AddCard` registers a new payment card and `Withdraw` simulates the action of withdrawing money from the bank account associated with the card that is currently inserted into the machine. Listing 7.4a shows the definitions of these two operations. They are impure because they have side effects – specifically, they modify the state.

Listing 7.4a VDM-SL

```

1  operations
2  AddCard: Card ==> ()
3  AddCard(c) == validCards := validCards union { c }
4  pre c not in set validCards
5  post c in set validCards;
...  Continues on the next page ...
```

```

... Continued from the previous page ...

6   Withdraw: AccountId * Amount ==> real
7   Withdraw(id, amount) ==
8       let newBalance = accounts(id).balance - amount
9       in
10      (
11          accounts(id).balance := newBalance;
12          return newBalance;
13      )
14   pre currentCard in set validCards and pinOk and
15       currentCard in set accounts(id).cards and
16       id in set dom accounts
17   post let accountPre = accounts~(id),
18         accountPost = accounts(id)
19         in accountPre.balance = accountPost.balance +
20             amount and
21             accountPost.balance = RESULT;

```

Impure operations in VDM-SL become methods in C# by Rule 4.2. Since the AddCard and Withdraw operations are guarded by pre- and postconditions, their implicitly defined functions become pure methods in C# by Rule 4.4. Listing 7.4b shows the methods related to AddCard in C#.

Listing 7.4b

C# translation

```

1   public static void AddCard(Card c)
2   {
3       Contract.Requires(c != null);
4       Contract.Requires(PreAddCard(c, State));
5       Contract.Ensures(
6           PostAddCard(c, Contract.OldValue(State), State));
7
8       State.ValidCards.Add(c);
9   }
10
11   [Pure]
12   public static bool PreAddCard(Card c, State st)
13   {
14       Contract.Requires(c != null);
15       Contract.Requires(st != null);
16
17       return !st.ValidCards.Contains(c);
18   }
... Continues on the next page ...

```

```
... Continued from the previous page ...

19 [Pure]
20 public static bool PostAddCard(Card c, St oldSt, St st)
21 {
22     Contract.Requires(c != null);
23     Contract.Requires(oldSt != null);
24     Contract.Requires(st != null);
25
26     return st.ValidCards.Contains(c);
27 }
```

Since set instances are immutable in VDM-SL, adding an element to a set is achieved by uniting the current set with a singleton set containing the new element and then assigning the result to the corresponding variable. This is what the `AddCard` operation does to the set stored in the `validCards` state field. In .NET, however, instances of `HashSet<T>` are mutable and provides an `Add` method for this purpose. The translation rule for assignments knows about this and therefore replaces the assignment with a call to `Add` when constructing the C# AST (see line 8 in Listing 7.4b). All state fields are accessed as properties of the common `State` object in C# except in pre- and postconditions which access the pre-state and post-state through method parameters.

Listing 7.4c shows the C# translation of the `Withdraw` operation and its related predicate functions.

```
Listing 7.4c C# translation

1 public static decimal Withdraw(AccountId id, Amount amount)
2 {
3     Contract.Requires(id != null);
4     Contract.Requires(amount != null);
5     Contract.Requires(PreWithdraw(id, amount, State));
6     Contract.Ensures(
7         PostWithdraw(id, amount, Contract.Result<decimal>(),
8             Contract.OldValue(State), State));
9
10    var newBalance = State.Accounts[id].Balance - amount;
11    {
12        State.Accounts[id].Balance = newBalance;
13        return newBalance;
14    }
15 }

... Continues on the next page ...
```

```

... Continued from the previous page ...

16 [Pure]
17 public static bool PreWithdraw(AccountId id, Amount amount,
18                               St st)
19 {
20     Contract.Requires(id != null);
21     Contract.Requires(amount != null);
22     Contract.Requires(st != null);
23
24     return st.ValidCards.Contains(st.CurrentCard)
25           && st.PinOk
26           && st.Accounts[id].Cards.Contains(st.CurrentCard)
27           && st.Accounts.ContainsKey(id);
28 }
29
30 [Pure]
31 public static bool PostWithdraw(AccountId id, Amount amount,
32                                 decimal result,
33                                 St oldSt, St st)
34 {
35     Contract.Requires(id != null);
36     Contract.Requires(amount != null);
37     Contract.Requires(oldSt != null);
38     Contract.Requires(st != null);
39
40     return Let(() =>
41     {
42         var accountPre = oldSt.Accounts[id];
43         var accountPost = st.Accounts[id];
44         return accountPre.Balance == accountPost.Balance
45                + amount
46                && accountPost.Balance == result;
47     });
48 }

```

Local variable declarations in let-expressions and -statements are translated accordingly to locally-scoped variable declarations in C# using the `var` keyword. Furthermore, let-expressions produce result values. To accommodate this in C#, they are wrapped into immediately-invoked function expressions (IIFEs) via the `Let` helper method (see lines 40-47 in Listing 7.4c). The IIFE pattern is common in JavaScript [57], but not in C#. Still, without IIFEs in the C# code, either the result value must be stored in a temporary variable or the local scope of the declared variables cannot be respected.

This case study – while not exhaustive – has shown that the translation rules in the transcompiler prototype supports a large subset of the VDM-SL language; however, some constructs – for instance pattern matching – are still unsupported and others – for instance let-expressions – give rise to unusual C# code like employment of the IIFE pattern.

7.2 Benchmarking FAD codes

The VDM-SL specification for the algorithm that obfuscates financial accounting district (FAD) codes is used by Jørgensen et al. [24] for enhancing the Java code generator in Overture with support for VDM-SL traces and combinatorial testing. It has been used in practice by Fujitsu.

7.2.1 Setup

Like the ATM model, the FAD obfuscation model is rich in design-by-contract elements. A FAD code is a six-digit number that identifies a retail branch. The obfuscation algorithm defined in the `convert` function takes a mapping of digits as input and computes a permutation of all FAD codes so that no FAD code is mapped to itself, as shown in Listing 7.5.

Listing 7.5 VDM-SL

```

1  val ues
2      SIZE = 6;
3      MAX = 10 ** SIZE - 1; -- = 10^6 - 1 = 999999
4
5  types
6      Di gi tMap = inmap nat to nat
7      inv m ==
8          let di gi ts = { 0, ..., 9 }
9          in dom m = di gi ts and rng m = di gi ts and
10             forall c in set dom m & m(c) <> c;
11
12      FAD = nat
13      inv f == f <= MAX
14
15  functi ons
16      convert: FAD * Di gi tMap -> FAD
17      convert(fad, dm) ==
18          let di gi ts = di gi tsOf(fad)
19          in val Of([ dm(di gi ts(i)) | i in set inds di gi ts ])
20      post RESULT <> fad;
```

An exhaustive test checks that the `convert` function maps every FAD code to another FAD code. When the value of `SIZE` is 6, there are about one million FAD codes to be checked in total, specifically $10^6 - 1 = 999,999$.

This section carries out the exhaustive test in order to measure the computational performance of both the C# output from the transcompiler prototype and the output from the Java code generator in Overture. In C#, design-by-contract elements are implemented and checked by the .NET Code Contracts Library [9], whereas they are represented by JML annotations in Java and checked by the OpenJML tool [15].

Three experiments are carried out for each platform:

Experiment I: No contracts are specified. This corresponds to removing all design-by-contract elements from the VDM-SL specification and just executing the `convert` function plainly. In C#, the definition of the `CONTRACTS_FULL` symbol is omitted to disregard all contracts. In Java, the program is compiled with the normal Java compiler in OpenJDK 7, which disregards all JML annotation comments. Thus, no contracts are emitted in the bytecode.

Experiment II: Contracts are specified, but not checked during execution. This is the default mode of operation for production-ready software. In .NET Code Contracts, the contracts can be disabled via the Visual Studio extension. In **Java, they can be disabled by omitting the ‘-ea’ (enable assertions) command line argument**. The program is compiled with the OpenJML compiler.

Experiment III: Contracts are specified and checked during execution. This is the approach employed for software under development.

The benchmark program runs the exhaustive test eight times and reports the average time spent per iteration. The first iteration is a warm-up iteration for the virtual machines – .NET CLR and Java HotSpot VM, respectively – and is not included in the result. The return value of `convert` is stored in a field variable whose final value is printed to the console in order to avoid the virtual machine performing dead code elimination [58].

7.2.2 Results

Table 7.6 shows the results of the benchmarking experiments. The numbers denote the time spent executing a particular experiment. They have been performed with `SIZE` values of 1 to 6, cf. Listing 7.5 (line 2), yielding $10^{\text{SIZE}} - 1$ iterations in an exhaustive test.

Experiment I and II in .NET show that there is no notable difference between omitting the CODE_CONTRACTS symbol and configuring the Visual Studio extension to disable code contracts on runtime. In fact, when contracts are disabled in Visual Studio, they are erased from the bytecode, just like omitting the CODE_CONTRACTS symbol would do [9]. They complete the exhaustive test of million iterations in about 2,300 milliseconds.

In Experiment III, the code contracts are emitted in the CIL bytecode and checked during the execution, which results in a performance overhead. It completes the exhaustive test in about 3,600 milliseconds, implying an overhead from code contracts of about 60%.

In Java, Experiment I performs about as well as .NET, considering that .NET CLR and Java HotSpot VM are two different virtual machines and the C# transcompiler prototype and Java code generator in Overture take slightly different approaches to translating VDM-SL. It completes the exhaustive test in about 2,500 milliseconds.

However, Experiment II and III, which are compiled with OpenJML, suffer from a very large overhead. They complete the exhaustive test in about 440,700 and 443,500 milliseconds, respectively, making them approximately 175 times slower than Experiment I. This overhead has to be caused by the OpenJML compiler, otherwise the performance of Experiment II would be comparable to Experiment I, which it is clearly not. When assertions are disabled in Java, they should have no impact on the execution whatsoever [2]. This confirms the results of Jørgensen et al. [24].

	Table 7.6 Benchmark results (in milliseconds)						
	.NET			Java			
	SIZE	I	II	III	I	II	III
1	< 1	< 1	< 1	< 1	2	2	
2	< 1	< 1	< 1	2	20	22	
3	1	1	1	4	245	254	
4	15	15	23	22	3,103	3,212	
5	190	189	295	212	37,626	38,401	
6	2,273	2,279	3,610	2,498	440,716	443,523	

As shown in Experiment III, when code contracts enabled, .NET Code Contracts is about 120 times faster than OpenJML. Figure 7.7 shows the results in a logarithmically scaled graph.

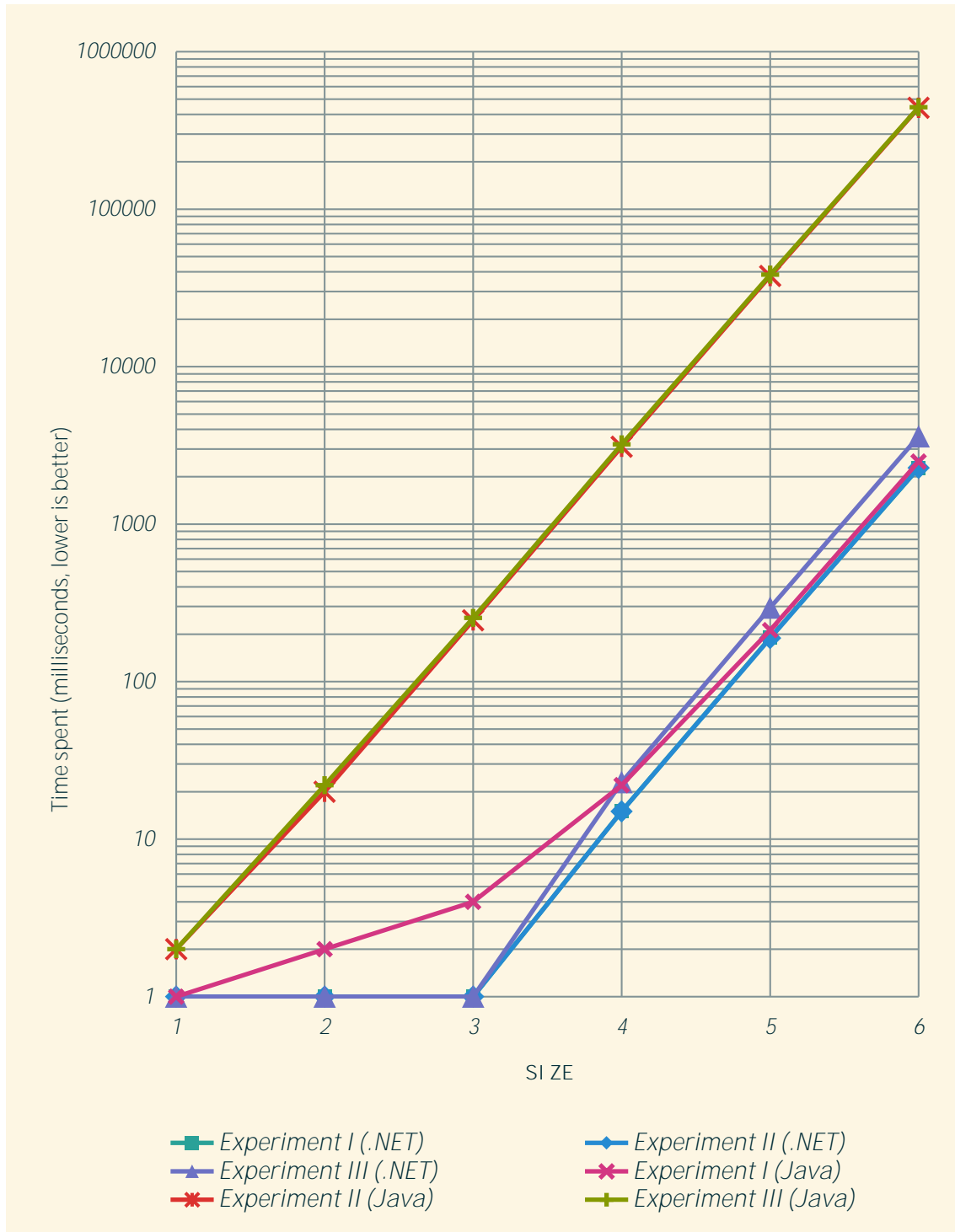


Figure 7.7

The benchmark results. The scale of the y-axis is logarithmic.

“ Architecture begins where engineering ends.
– Walter Gropius (1883-1969)

Conclusion

This thesis has presented rules for translating language constructs in VDM-SL to equivalent constructs in C#. They have been implemented in a transcompiler prototype, developed with test-driven development, and compared to the translations made by the Java code generator in Overture. Finally, the performance of the .NET Code Contracts library has been compared to the performance of the OpenJML tool.

This chapter discusses the validity of the results, revisits the objectives of the thesis and presents future work to be done.



Chapter contents

Section 8.1 discusses the validity of the results achieved in this thesis regarding the C# translations and the performance benchmarks between .NET Code Contracts and OpenJML.

Section 8.2 revisits the objectives of this thesis to finally verify or falsify the three hypotheses presented in Section 1.1.

Section 8.3 presents future work to be done that relates to this project, in particular regarding enhancements and optimisations for the transcompiler and another approach to automated testing.

8.1 Validity of results

The translation rules for VDM-SL to C# presented in this thesis have been chosen among multiple feasible approaches in C#. This does not guarantee that they represent the optimal translations from VDM-SL to C# – there may very well be more C# approaches that have not been considered here.

Whenever possible, translations that make use of native language constructs in C# and the standard library of .NET are favoured over translations that require manual implementations. This is done in accordance with Hypothesis I. However, this choice does not always lead to the most maintainable solution. Manually implemented utility methods may get the job done more efficiently. This is the philosophy followed by the Java code generator in Overture. It makes heavy use of utility methods in order to keep the resulting Java code as clean as possible.

This is not the only difference between the VDM-SL-to-C# translation rules and the Java code generator. They also enforce design-by-contract elements differently. Inferred type invariants such as bounds-checking for `nat1` and null-checking for composite types are implemented as pre- and postconditions in C# when they occur as method parameters and return values. The Java code generator, in the other hand, enforces the type invariants through JML assertions. Furthermore, type aliases are inlined in Java, but declared as separate types in C#. The possibility that these differences influence the benchmark results – as well as the fact that .NET CLR and Java HotSpot VM are two different environments – should be taken into consideration. Still, the difference in performance between .NET Code Contracts and OpenJML is so significant that it cannot be attributed to platform differences alone.

When investigating the suitability of test-driven development for compiler construction, only syntactical testing approaches have been considered. Their focus on the syntax of the output is beneficial when developing a transcompiler whose code output is micromanaged to ensure maintainability. However, there are other levels to test on, for example semantic tests that observe the *behaviour* of the translated output rather than its *appearance*. They are listed as future work.

8.2 Objectives, revisited

In Section 1.1, the objectives of this thesis were formulated as three hypotheses to be verified or falsified.

The first hypothesis concerned the VDM-SL and C# languages:



Hypothesis I


VDM-SL versus C#

A large subset of the language constructs in VDM-SL have semantically equivalent constructs in C# that result in maintainable C# source code.

This hypothesis has been verified in Chapters 3 to 5. VDM-SL specifications are expressible in C# through classes whose methods, properties and nested classes represent VDM-SL functionality well. The type system of C# is strong enough to handle the majority of VDM-SL expressions and types, including union types and self-referential types.

Nevertheless, some technical issues have arisen: Numeric values have finite magnitude and precision; value types in unions are subject to boxing which implies referential equality over structural equality and the implementation of `GetHashCode` in mutable data classes may lead to unstable behaviour of hash-based data structures.


The second hypothesis investigates the use of test-driven development (TDD) for compiler construction:

	Hypothesis II	TDD for compilers
	Test-driven development is feasible for compiler construction without sacrificing developer productivity, analysability of the test code or quality in unit testing.	

This hypothesis has been partially falsified in Chapter 6. Each testing approach presented would either sacrifice developer productivity or test quality; however, Chapter 6 does not cover every feasible testing approach, as pointed out in Section 8.1.

Tree comparison needs specifying the exact trees to translate and expect; raw text comparison is fragile to small differences in the output; regular expressions are error-prone to specify and fail to discover unanticipated duplicates in the output; parsing of the target language is expensive to implement. Nevertheless, the parsing-based approach is most beneficial as it results in the development of a parser for the target language which can be used by the translation rules to gain maintainable implementation code.

The third hypothesis compares the .NET Code Contracts library to the JML language and the OpenJML tool:

	Hypothesis III	.NET Code Contracts versus JML
	The .NET Code Contracts library supports the same set of design-by-contract elements in VDM-SL as JML. The performance of .NET Code Contracts surpasses the performance of OpenJML.	

This hypothesis has been verified in Chapters 4 and 7. The .NET Code Contracts library supports pre- and postconditions and invariants like JML. In particular, it supports referring to the result value and pre-state in postconditions and disabling invariants temporarily in `atomic` statements, even though it does not have a concept of ghost variables like JML. The fact that it only checks invariants at method exit and not also at method entrance does not make a difference when all state is represented by properties in C#.

.NET Code Contracts performs significantly better than the OpenJML tool in the FAD code obfuscation model, which covers all kinds of design-by-contract elements in VDM-SL. The results of the benchmarks reveal that .NET Code Contracts is about 120 times faster than OpenJML. This confirms the results of Jørgensen et al. [24], indicating that OpenJML causes a large computational overhead. Its impact is further demonstrated by the large gap in times between Experiment I and II.

8.3 Future work

The translation rules and transcompiler prototype targets only a subset of VDM-SL, leaving plenty of room for enhancements.

8.3.1 Enhancements

Among the unsupported language constructs of VDM-SL are pattern matching and decomposition. Besides patterns for decomposing composite types, as witnessed in Section 7.1.2, VDM-SL provides patterns for many of its built-in types to allow decomposition of tuples, sets and sequences, among others. Patterns are useful in conjunction with the `cases` construct, which is similar to the `switch` construct in C#, except for its extensive support for pattern matching.

Extending the set of translation rules to support the object-oriented VDM++ language in addition to VDM-SL faces many challenges such as the treatment of `OBJECT ALIASING` and the translation of overloaded methods and multiple class inheritance, given that C# only supports single class inheritance [3]. Furthermore, VDM++ supports concurrency via multi-threading and synchronisation constructs [7]. The standard library of .NET offers similar features in addition to native concurrency constructs in C#.

To facilitate software validation, the standard library of the VDM-10 platform provides a testing framework, `VDMUnit` [1], inspired by `JUnit`. Two prominent testing frameworks on the .NET platform are `NUnit` [59] and `xUnit.net` [35], both of which are also inspired by `JUnit` [34]. Translating test cases from `VDMUnit` to `NUnit` and `xUnit.net` is therefore an option worth exploring.

8.3.2 Optimisations

There is also room for improvement in the current transcompiler prototype. While it implements the code generation phase, it does not focus very much on optimisation.

The translation rules for manipulating sets, sequences and mappings make heavy use of the extension methods of LINQ. However, since they are based on the generic `IEnumerable<T>` type, the results need to be converted into the correct kind of collection, for example by calling `ToHashSet` to convert the results into instances of `HashSet<T>`. When an expression is composed of multiple collection operations, it is only necessary to call `ToHashSet` on the final result, not the intermediate results. However, the transcompiler does not take this into account and calls `ToHashSet` unnecessarily on temporary results, just to call a LINQ extension method that hides the collection behind `IEnumerable<T>` again.

Another possible optimisation is to remove unused `using` directives in the C# code so that external types are not needlessly brought into scope. In general, since the transcompiler manipulates a C# AST, it may analyse the C# program for redundancies such as redundant namespace qualifiers and unused variables.

While certain problems can be solved trivially in VDM-SL, the translated code may give rise to complex C# code. However, there are also problems which can be trivially solved in .NET, but require complex solutions in VDM-SL. One example is the computation of the sum of a set of numbers. In .NET, this is solved by simply calling the `Sum` method in LINQ. In VDM-SL, it requires the manual definition of a `Sum` function which operates either recursively or imperatively by iteration. Similarly, there is no native way in VDM-SL to convert a set into a sequence. In .NET, one would simply call the `ToList` method.

Being able to recognise these non-trivial patterns in VDM-SL may improve the quality of the transcompiler considerably. Nevertheless, this is a difficult task that requires great insight into static program analysis and pattern recognition.

8.3.3 Semantic testing

The final room for improvement concerns the testing approach used in test-driven development. This thesis has explored testing approaches on a syntactical level to make assertions about the appearance of the generated C# code, for example regarding the source code formatting and the concrete naming of classes and methods.

Semantic tests, on the other hand, make assertions about the behaviour of the generated C# program to see if it preserves the semantics of the VDM-SL specification. This is actually the testing approach applied for the Java code generator in Overture. From the command-line, it invokes the Java compiler on the generated Java code and runs the compiled Java program to see if it prints the expected messages to the console and returns the expected value.

It is another story for C#. The official C# compiler ships as a part of Visual Studio, which only targets the Windows operating system. Invoking the C# compiler from the command-line is difficult since it must be accessed through the command-line tools of Visual Studio.

A possible investigation for semantic testing is the penalty endured in developer productivity for implementing the communication between the testing framework, the C# compiler and the executed C# program. Furthermore, it would be interesting to see how the expected result of a test case is determined, perhaps via the interpreter tool in Overture.

References

- [1] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat and M. Verhoef, Validated Designs for Object-oriented Systems, London: Springer, 2005.
- [2] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley, The Java Language Specification, Oracle, 2013.
- [3] Microsoft, C# Language Specification 5.0, Microsoft, 2013.
- [4] **Microsoft**, “.NET,” 2016. [Online]. Available: <https://www.microsoft.com/net>. [Accessed 14 June 2016].
- [5] **TIOBE Software BV**, “TIOBE Index,” June 2016. [Online]. Available: http://www.tiobe.com/tiobe_index. [Accessed 12 June 2016].
- [6] **Stack Exchange Inc**, “Stack Overflow,” [Online]. Available: <http://stackoverflow.com/tags>. [Accessed 12 June 2016].
- [7] P. G. Larsen, K. Lausdahl, N. Battle, J. Fitzgerald, S. Wolff, S. Sahara, M. Verhoef, P. W. V. Tran-Jørgensen and T. Oda, VDM-10 Language Manual, 2016.
- [8] K. Cwalina and B. Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, Addison-Wesley Professional, 2009.
- [9] Microsoft, Code Contracts User Manual, Microsoft, 2013.
- [10] H. B. Christensen, Flexible, Reliable Software - Using Patterns and Agile Development, Boca Raton, Florida: Chapman & Hall/CRC, 2010.

- [11] K. Beck, *Test-driven Development: By Example*, Addison-Wesley Professional, 2003.
- [12] P. W. V. Tran-Jørgensen, M. Larsen and L. D. Couto, “A Code Generation Platform for VDM,” *Proceedings of the 12th Overture Workshop*, no. CS-TR-1446, January 2015.
- [13] P. W. V. Tran-Jørgensen, P. G. Larsen and G. T. Leavens, “Automated translation of VDM to JML annotated Java,” *International Journal on Software Tools for Technology Transfer*, 2016.
- [14] G. T. Leavens and Y. Cheon, “Design by Contract with JML,” 2006.
- [15] D. Cok, “OpenJML,” 2015. [Online]. Available: <http://www.openjml.org/>. [Accessed 11 June 2016].
- [16] JetBrains, “Kotlin Language Documentation,” 2016. [Online]. Available: <https://kotlinlang.org/docs/reference/>. [Accessed 14 June 2016].
- [17] The Overture community, “Overture,” 2016. [Online]. Available: <http://overturetool.org/>. [Accessed 14 June 2016].
- [18] P. G. Larsen, K. Lausdahl, P. W. V. Tran-Jørgensen, J. Coleman, S. Wolff, L. D. Couto and N. Battle, *Overture VDM-10 Tool Support: User Guide*, 2015.
- [19] JetBrains, “IntelliJ IDEA,” 2016. [Online]. Available: <https://www.jetbrains.com/idea/>. [Accessed 14 June 2016].
- [20] Microsoft, “Visual Studio,” 2016. [Online]. Available: <https://www.visualstudio.com/>. [Accessed 14 June 2016].
- [21] D. Bjørner, “Pinnacles of software engineering: 25 years of formal methods,” *Annals of Software Engineering*, vol. 10, no. 1-4, pp. 11-66, 2000.
- [22] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 1969.
- [23] G. T. Leavens, “JML Reference Manual,” 2013. [Online]. Available: <http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html>. [Accessed 14 June 2016].
- [24] P. W. V. Tran-Jørgensen, P. G. Larsen and N. Battle, “Using JML-based Code Generation to Enhance the Test Automation for VDM Models,” 2016.
- [25] C. Petzold, *.NET Book Zero*, 2007.

- [26] K. R. M. Leino and P. Müller, “Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs,” *Advanced Lectures on Software Engineering*, pp. 91-139, 2010.
- [27] RiSE, “Code Contracts for .NET,” 27 July 2015. [Online]. Available: <https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970>. [Accessed 14 June 2016].
- [28] A. W. Appel, *Modern Compiler Implementation in ML*, Cambridge, United Kingdom: Cambridge University Press, 1998.
- [29] N. Battle, “VDMJ User Guide,” *Fujitsu Services Ltd., UK, Tech. Rep*, 2009.
- [30] L. D. Couto, P. W. V. Tran-Jørgensen and K. Lausdahl, “Principles for Reuse in Formal Language Tools,” *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp. 1997-2000, 2016.
- [31] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Indianapolis, Indiana: Addison-Wesley, 1995.
- [32] The Apache Software Foundation, “The Apache Velocity Project,” 2010. [Online]. Available: <http://velocity.apache.org/>. [Accessed 14 June 2016].
- [33] R. Osherove, *The Art of Unit Testing*, Greenwich, Connecticut: Manning Publications Co., 2009.
- [34] JUnit, “JUnit,” 2016. [Online]. Available: <http://junit.org/junit4/>. [Accessed 14 June 2016].
- [35] Outercurve Foundation, “About xUnit.net,” 2015. [Online]. Available: <https://xunit.github.io/>. [Accessed 14 June 2016].
- [36] R. C. Martin, *Design Principles and Design Patterns*, Object Mentor, 2000.
- [37] M. Seemann, *Dependency Injection in .NET*, Shelter Island, New York: Manning Publications Co., 2010.
- [38] G. Meszaros, *Xunit Test Patterns: Refactoring Test Code*, Pearson Education, 2007.
- [39] J. Bloch, *Effective Java*, Second ed., Upper Saddle River, New Jersey: Addison-Wesley, 2008.
- [40] K. Lausdahl, H. Ishikawa and P. G. Larsen, “Interpreting Implicit VDM Specifications using ProB,” *Proceedings of the 12th Overture Workshop*, no. CS-TR-1446, January 2015.

- [41] Oracle, “Properties,” 2015. [Online]. Available: <http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html>. [Accessed 7 June 2016].
- [42] Microsoft, “const (C# Reference),” 2015. [Online]. Available: <https://msdn.microsoft.com/da-dk/library/e6w8fe1b.aspx>. [Accessed 7 June 2016].
- [43] 'Cnranger', “Passing for Programming,” 9 December 2010. [Online]. Available: <http://cnranger.blogspot.dk/2010/12/due-to-increased-accuracy-decimal-type.html>. [Accessed 7 June 2016].
- [44] R. Stephens, “C# Helper,” 11 July 2012. [Online]. Available: <http://csharpHelper.com/blog/2012/07/compare-the-performance-of-the-float-double-and-decimal-data-types-in-c/>. [Accessed 7 June 2016].
- [45] 'melitta', “BigRational,” 30 March 2010. [Online]. Available: <https://bcl.codeplex.com/wikipage?title=BigRational&referringTitle=Home>. [Accessed 14 June 2016].
- [46] Microsoft, “Language-Integrated Query (LINQ) (C#),” 2015. [Online]. Available: <https://msdn.microsoft.com/da-dk/library/mt693024.aspx>. [Accessed 7 June 2016].
- [47] Microsoft, “Query Syntax and Method Syntax in LINQ (C#),” 2015. [Online]. Available: <https://msdn.microsoft.com/en-us/library/Bb397947.aspx>. [Accessed 8 June 2016].
- [48] Oracle, “Interface Stream<T>,” 2016. [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>. [Accessed 7 June 2016].
- [49] Microsoft, “ValueType.Equals Method,” 2015. [Online]. Available: [https://msdn.microsoft.com/da-dk/library/2dts52z7\(v=vs.110\).aspx](https://msdn.microsoft.com/da-dk/library/2dts52z7(v=vs.110).aspx). [Accessed 14 June 2016].
- [50] Microsoft, “Choosing Between Class and Struct,” 2009. [Online]. Available: [https://msdn.microsoft.com/en-us/library/ms229017\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms229017(v=vs.110).aspx). [Accessed 14 June 2016].
- [51] E. Lippert, “Guidelines and rules for GetHashCode,” 28 February 2011. [Online]. Available: <https://blogs.msdn.microsoft.com/ericlippert/2011/02/28/guidelines-and-rules-for-gethashcode/>. [Accessed 14 June 2016].

- [52] Microsoft, “uint (C# Reference),” 2015. [Online]. Available: <https://msdn.microsoft.com/da-dk/library/x0sksh43.aspx>. [Accessed 14 June 2016].
- [53] H. Hariri, “Spek - A Specification Framework for the JVM,” 2016. [Online]. Available: <https://jetbrains.github.io/spek/>. [Accessed 12 June 2016].
- [54] JetBrains, “Type-Safe Groovy-Style Builders,” 2016. [Online]. Available: <https://kotlinlang.org/docs/reference/type-safe-builders.html>. [Accessed 12 June 2016].
- [55] S. Freeman and N. Pryce, Growing Object-Oriented Software, Guided by Tests, Boston, Massachusetts: Addison-Wesley, 2010.
- [56] S. Vohra, “CakeParse,” 2016. [Online]. Available: <http://sargunvohra.me/cakeparse/>. [Accessed 14 June 2016].
- [57] Mozilla Developer Network, “IIFE,” 15 December 2015. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Glossary/IIFE>. [Accessed 14 June 2016].
- [58] Oracle, “Frequently Asked Questions About the Java HotSpot VM,” [Online]. Available: <http://www.oracle.com/technetwork/java/hotspotfaq-138619.html>. [Accessed 14 June 2016].
- [59] NUnit, “NUnit,” 2015. [Online]. Available: <http://www.nunit.org/>. [Accessed 14 June 2016].

Glossary

ADAPTER PATTERN	A design pattern that enables collaboration between two classes that could not otherwise work together due to incompatible interfaces.
ANALYSABILITY	A quality of a software system that describes how easy it is to determine the cause of a software deficiency. The readability of the source code affects analysability.
ARRANGE-ACT-ASSERT	A design pattern in unit testing that streamlines the structure of a test case into three phases: arrangement of test input, acting upon the unit under test and checking test expectations.
BOXING	The act of converting a value type to a reference type in order to allocate the value type in the heap instead of the stack and receive a pointer to it. It unifies value types and reference types under a single notion, gaining the properties of reference types.
BUILDER PATTERN	A design pattern that encapsulates the construction of a complex object, for example a data structure, into separate, smaller tasks.
CONSTANT FOLDING	A compiler optimisation that evaluates constant expressions on compile-time to spare the evaluation on runtime.
CONSTRUCTOR INJECTION	A design pattern that enables dependency injection by acquiring dependencies through the constructor.
DEAD CODE ELIMINATION	A compiler optimisation that removes unreachable or redundant code from the emitted bytecode.
DECORATOR PATTERN	A design pattern that enhances the functionality of a class without modifying the class itself.

DEFENSIVE PROGRAMMING	The situation in which a program takes precautions against misuse and invalid input in order to stay properly functioning.
DEPENDENCY INJECTION	An object-oriented approach for decoupling high-level objects from the low-level objects that they depend on.
EQUIVALENCE RELATION	An operation on two objects that is reflexive, symmetric, transitive and consistent.
INTEGRATED DEVELOPMENT ENVIRONMENT	A computer application that facilitates software development, usually by means of a source code editor, a debugger and intelligent coding assistance.
LOSS OF SIGNIFICANCE	A situation that occurs in binary floating-point arithmetic in which the number of accurate digits is reduced due to the continuous propagation of relative inaccuracies.
MODEL	A high-level representation of a software system.
MULTIPLE MAINTENANCE PROBLEM	The situation in which the same code logic is duplicated, implying that a potential bugfix or refactoring must be performed in multiple locations.
OBJECT ALIASING	The situation in which two distinct object references point to the same object in memory so that modifying the object from one location will also affect the other, usually unintentionally.
PASS-BY-SHARING SEMANTICS	The act of copying only the reference to an object upon assignments.
PASS-BY-VALUE SEMANTICS	The act of copying values upon assignments.
POLYMORPHISM, PARAMETRIC	A property of a computation in which the computation can operate uniformly on different kinds of input. It corresponds to generics in object-oriented programming languages.
POLYMORPHISM, SUBTYPE	A property of a computation in which a computation that operates on a supertype may also operate uniformly on any subtypes thereof.
REFERENTIAL EQUALITY	The situation that two values are equal when they point to the exact same object in memory, that is, their values are <i>identical</i> .
REFERENTIAL OPACITY	The opposite of referential transparency.

REFERENTIAL TRANSPARENCY	A property of a computation in which the computation can be replaced by its result without changing the result or overall behaviour of the program.
REGRESSION	A bug that is introduced after modifying the production code, for example to correct another bug.
SIDE EFFECT	A secondary, usually adverse, action performed by a computation.
SINGLETON PATTERN	A design pattern which ensures that only one instance of a class exists at any point in time.
STREAM	A potentially infinite sequence of elements that is made available over time rather than evaluated eagerly.
STRUCTURAL EQUALITY	The situation that two values are equal when they represent the same piece of data.
TEST DOUBLE	A substitute for a software unit used in testing, usually to produce fake data.
VISITOR PATTERN	A design pattern that separates an algorithm from the data structure it operates on.

All translations

This appendix lists all translations realised in this thesis. A few of them have not been implemented in the prototype.

Flat specifications

A flat specification in VDM-SL becomes a class named 'Global' in C#. It is marked with the `public` and `static` modifiers and `partial` if spread across multiple files.

```
types                                     public static class Global
/* Definition 1 */                       {
/* Definition 1 */                       /* Definition 1 */
functions                               /* Definition 2 */
/* Definition 2 */                       }

/* Definition 2 */
```

Modules

A module in VDM-SL becomes a class in C#. It is marked with the `public` and `static` modifiers. Non-exported definitions are marked with the `private` modifier.

```
module Alpha                             public static class Alpha
{
definitions                               /* Definition 1 */
types                                     /* Definition 2 */
/* Definition 1 */                       }
/* Definition 2 */
functions
/* Definition 2 */

end Alpha
```

Values

A value in VDM-SL becomes a read-only auto-property in C#.

<code>values</code>	<code>private static T Alpha { get; }</code>
<code>Alpha = /* E */</code>	<code>= /* E */;</code>

Impure operations

An impure operation in VDM-SL becomes a method in C#.

<code>operations</code>	<code>public static void Alpha()</code>
<code>Alpha: () ==> ()</code>	<code>{</code>
<code>Alpha() == /* Body */</code>	<code>/* Body */</code>
	<code>}</code>
<code>Bravo: int ==> bool</code>	<code>public static bool Bravo(int n)</code>
<code>Bravo(n) == /* Body */</code>	<code>=> /* Body */</code>

Pure operations

A pure operation in VDM-SL becomes a method in C# tagged with [Pure].

<code>operations</code>	<code>[Pure]</code>
<code>pure Alpha: () ==> int</code>	<code>public static int Alpha()</code>
<code>Alpha() == /* Body */</code>	<code>=> /* Body */</code>

Functions

A function in VDM-SL becomes a method in C# tagged with [Pure].

<code>functions</code>	<code>[Pure]</code>
<code>Alpha: int -> bool</code>	<code>public static bool Alpha(int n)</code>
<code>Alpha(n) == /* Body */</code>	<code>=> /* Body */</code>
<code>Bravo[@T]: @T -> bool</code>	<code>public static bool Bravo<T>(T t)</code>
<code>Bravo(t) == /* Body */</code>	<code>=> /* Body */</code>

Implicit functions and operations

An implicit function or operation in VDM-SL becomes a method in C# that throws an instance of `NotImplementedException`.

```
functions                                     [Pure]
  Alpha(n: int) r: int                       public static int Alpha(int n)
  pre /* Precondition */                     {
  post /* Postcondition */                   /* Precondition */
                                              /* Postcondition */
                                              throw new
                                              NotImplementedException();
                                              }
```

States

A state in VDM-SL becomes an auto-property named 'State' in C#. It is marked with the `private` and `static` modifiers.

```
state Alpha of                               private static Alpha State
  /* Fields */                               { get; set; } = /* E */;
  init s == s = /* E */
end
```

Preconditions

A precondition in VDM-SL becomes a call to `Contract.Requires` in C#.

```
pre /* E */                                Contract.Requires(/* E */);
```

Postconditions

A postcondition in VDM-SL becomes a call to `Contract.Ensures` in C#. A reference to the return value becomes a call to `Contract.Result<T>`. A reference to the pre-state becomes a call to `Contract.OldValue`.

```
post /* E */                                Contract.Ensures(/* E */);
  RESULT                                     Contract.Result<TReturn>;
  alpha~                                    Contract.OldValue(alpha);
```

Invariants

An invariant in VDM-SL becomes a call to `Contract.Invariant` in C# within a method named `'ObjectInvariant'` that is tagged with `[ContractInvariantMethod]`. Whether invariants are enabled or temporarily disabled is tracked by a property named `'AreInvariantsEnabled'`, although this feature has not been implemented.

```
inv s == /* E */  
  
[ContractInvariantMethod]  
private void ObjectInvariant()  
{  
    Contract.Invariant(  
        !AreInvariantsEnabled  
        || /* E */);  
}
```

Booleans

<code>bool</code>	<code>bool</code>
<code>true</code>	<code>true</code>
<code>false</code>	<code>false</code>
<code>not a</code>	<code>!a</code>
<code>a and b</code>	<code>a && b</code>
<code>a or b</code>	<code>a b</code>
<code>a => b</code>	<code>!a b</code>
<code>a <=> b</code>	<code>!(a ^ b)</code>
<code>a = b</code>	<code>a == b</code>
<code>a <> b</code>	<code>a != b</code>

Characters

<code>char</code>	<code>char</code>
<code>'A'</code>	<code>'A'</code>
<code>a = b</code>	<code>a == b</code>
<code>a <> b</code>	<code>a != b</code>

Numbers

The `Modulo` and `IntPower` extension methods are defined in the runtime library.

<code>nat1</code>	<code>int</code>
<code>nat</code>	<code>int</code>
<code>int</code>	<code>int</code>
<code>rat</code>	<code>decimal</code>
<code>real</code>	<code>decimal</code>
<code>42</code>	<code>42</code>
<code>13.37</code>	<code>13.37m</code>
<code>-a</code>	<code>-a</code>
<code>abs a</code>	<code>Math.Abs(a)</code>
<code>floor a</code>	<code>(int) Math.Floor(a)</code>
<code>a + b</code>	<code>a + b</code>
<code>a - b</code>	<code>a - b</code>
<code>a * b</code>	<code>a * b</code>
<code>a / b</code>	<code>a / b</code>
<code>a div b</code>	<code>a / b</code>
<code>a rem b</code>	<code>a % b</code>
<code>a mod b</code>	<code>a.Modulo(b)</code>
<code>a ** b</code>	<code>a.IntPower(b)</code>
<code>a < b</code>	<code>a < b</code>
<code>a <= b</code>	<code>a <= b</code>
<code>a > b</code>	<code>a > b</code>
<code>a >= b</code>	<code>a >= b</code>
<code>a = b</code>	<code>a == b</code>
<code>a <> b</code>	<code>a != b</code>

Quote types

A quote type in VDM-SL becomes an enum member of the `Quote` enum in C#.

<code><Alpha></code>	<code>Quote.Alpha</code>
----------------------------	--------------------------

Tokens

The `Token` type is defined in the runtime library.

<code>token</code>	<code>Token</code>
<code>mk_token(a)</code>	<code>Token.Create(a)</code>
<code>a = b</code>	<code>a == b</code>
<code>a <> b</code>	<code>a != b</code>

Sets

The ToHashSet and PowerSet extension methods are defined in the runtime library.

<pre> set of T { } { a, b, c } { a, ..., b } card a dunion a dinter a power a a in set b a in set dom b a in set rng b a not in set b a not in set dom b a not in set rng b a union b a := a union { b } a := a union b a inter b a := a inter b a \ b a := a \ { b } a := a \ b a subset b a psubset b a = b a = { } a <> b a <> { } </pre>	<pre> HashSet<T> new HashSet<T>() new HashSet<T> { a, b, c } Enumerable.Range(a, b - a + 1) a.Count a.Cast<IEnumerable<T>>() .Aggregate(Enumerable.Union) .ToHashSet() a.Cast<IEnumerable<T>>() .Aggregate(Enumerable.Intersect) .ToHashSet() a.PowerSet() b.Contains(a) b.ContainsKey(a) b.ContainsValue(a) !b.Contains(a) !b.ContainsKey(a) !b.ContainsValue(a) a.Union(b).ToHashSet() a.Add(b); a.UnionWith(b); a.Intersect(b).ToHashSet() a.IntersectWith(b); a.Except(b).ToHashSet() a.Remove(b); a.ExceptWith(b); a.IsSubsetOf(b) a.IsProperSubsetOf(b) a.SetEquals(b) !a.Any() !a.SetEquals(b) a.Any() </pre>
--	--

Set comprehensions

A set comprehension in VDM-SL becomes a LINQ query in C#.

<pre> { c a in set s & p } </pre>	<pre> (from a in s where p select c).ToHashSet() </pre>
<pre> { c a, b in set s } </pre>	<pre> (from a in s from b in s select c).ToHashSet() </pre>

Sequences

The sequence modification operator (++) has not been translated.

<code>seq of T</code>	<code>List<T></code>
<code>seq1 of T</code>	<code>List<T></code>
<code>[]</code>	<code>new List<T>()</code>
<code>[a, b, c]</code>	<code>new List<T> { a, b, c }</code>
<code>hd a</code>	<code>a.First()</code>
<code>tl a</code>	<code>a.Skip(1).ToList()</code>
<code>len a</code>	<code>a.Count</code>
<code>elems a</code>	<code>a.ToHashSet()</code>
<code>inds a</code>	<code>Enumerable.Range(1, a.Count)</code>
	<code>.ToHashSet()</code>
<code>reverse a</code>	<code>a.Reverse().ToList()</code>
<code>conc a</code>	<code>a.Cast<IEnumerable<T>>()</code>
	<code>.Aggregate(Enumerable.Concat)</code>
	<code>.ToList()</code>
<code>a ^ b</code>	<code>a.Concat(b).ToList()</code>
<code>a := a ^ [b]</code>	<code>a.Add(b);</code>
<code>a := a ^ b</code>	<code>a.AddRange(b);</code>
<code>a := [b] ^ a</code>	<code>a.Insert(b, 0);</code>
<code>a := b ^ a</code>	<code>a.InsertRange(b, 0);</code>
<code>a(i)</code>	<code>a[i - 1]</code>
<code>a(i, ..., j)</code>	<code>a.Skip(i - 1)</code>
	<code>.Take(j - i + 1)</code>
	<code>.ToList()</code>
<code>a = b</code>	<code>a.SequenceEqual(b)</code>
<code>a = []</code>	<code>!a.Any()</code>
<code>a <> b</code>	<code>!a.SequenceEqual(b)</code>
<code>a <> []</code>	<code>a.Any()</code>

Sequence comprehensions

A sequence comprehension in VDM-SL becomes a LINQ query in C#.

<code>[c a in set s & p]</code>	<code>(from a in s</code>
	<code>where p</code>
	<code>select c).ToList()</code>
<code>[c a, b in set s]</code>	<code>(from a in s</code>
	<code>from b in s</code>
	<code>select c).ToList()</code>

Mappings

The `ToDictionary` (with no arguments), `OverrideBy` and `DictionaryEquals` extension methods are defined in the runtime library. The mapping composition (`comp`) and iteration (`**`) operators have not been translated.

<code>map T to U</code>	<code>Dictionary<T, U></code>
<code>inmap T to U</code>	<code>Dictionary<T, U></code>
<code>{ -> }</code>	<code>new Dictionary<T, U>()</code>
<code>{ a -> b, }</code>	<code>new Dictionary<T, U> { [a] = b }</code>
<code>dom a</code>	<code>a.Keys.ToHashSet()</code>
<code>rng a</code>	<code>a.Values.ToHashSet()</code>
<code>inverse a</code>	<code>a.ToDictionary(_ => _.Value, _ => _.Key)</code>
<code>merge a</code>	<code>a.Cast<IEnumerable< KeyValuePair<T, U>>()&br/>.Aggregate(Enumerable.Concat) .ToDictionary() a.Concat(b).ToDictionary() a.Add(b, c); a.OverrideBy(b).ToDictionary() a[b] = c; a.Where(_ => s.Contains(_.Key)) .ToDictionary() a.Where(_ => !s.Contains(_.Key)) .ToDictionary() a.Remove(b); a.Where(_ => s.Contains(_.Value)) .ToDictionary() a.Where(_ => !s.Contains(_.Value)) .ToDictionary() a[i] a.DictionaryEquals(b) !a.DictionaryEquals(b)</code>
<code>a munion b</code>	
<code>a := a munion { b -> c }</code>	
<code>a ++ b</code>	
<code>a := a ++ { b -> c }</code>	
<code>s <: a</code>	
<code>s <=: a</code>	
<code>a := { b } <=: a</code>	
<code>a :> s</code>	
<code>a :-> s</code>	
<code>a(i)</code>	
<code>a = b</code>	
<code>a <> b</code>	

Mapping comprehensions

A mapping comprehension in VDM-SL becomes a LINQ query in C#.

<code>{ c -> d a in set s & p }</code>	<code>(from a in s where p select new KeyValuePair<T, U>(c, d)) .ToDictionary()</code>
<code>{ c -> d a, b in set s }</code>	<code>(from a in s from b in s select new KeyValuePair<T, U>(c, d)) .ToDictionary()</code>

Strings

<code>seq of char</code>	<code>string</code>
<code>seq1 of char</code>	<code>string</code>
<code>" "</code>	<code>" "</code>
<code>"ABC"</code>	<code>"ABC"</code>
<code>len a</code>	<code>a.Length</code>
<code>a ^ b</code>	<code>a + b</code>
<code>a = b</code>	<code>string.Equals(a, b)</code>
<code>a <> b</code>	<code>!string.Equals(a, b)</code>

Tuples

<code>T1 * T2 * T3</code>	<code>Tuple<T1, T2, T3></code>
<code>mk_(a, b, c)</code>	<code>Tuple.Create(a, b, c)</code>
<code>a.#1</code>	<code>a.Item1</code>
<code>a = b</code>	<code>a == b</code>
<code>a <> b</code>	<code>a != b</code>

Composite type definitions

A composite type definition becomes a data class in C# with auto-property members corresponding to the fields of the composite type.

<code>Alpha :: a: int</code>	<code>public sealed class Alpha</code>
<code> b: bool</code>	<code>: ICopyable<Alpha>,</code>
	<code> IEquatable<Alpha></code>
	<code>{</code>
	<code> public int A { get; set; }</code>
	<code> public bool B { get; set; }</code>
	<code> // ...</code>
	<code>}</code>

Records

<code>mk_Alpha(a, b)</code>	<code>new Alpha(a, b)</code>
<code>r.a</code>	<code>r.A</code>
<code>a = b</code>	<code>a.Equals(b)</code>
<code>a <> b</code>	<code>!a.Equals(b)</code>
<code>is_Alpha(a)</code>	<code>a is Alpha</code>

Unions

<code>T1 T2</code>	<code>object</code>
<code>T1 T2 T3</code>	<code>object</code>

Optional types

<code>ni l</code>	<code>nul l</code>
<code>[bool]</code>	<code>bool ?</code>
<code>[char]</code>	<code>char?</code>
<code>[int]</code>	<code>int?</code>
<code>[nat]</code>	<code>int?</code>
<code>[nat1]</code>	<code>int?</code>
<code>[rat]</code>	<code>decimal ?</code>
<code>[real]</code>	<code>decimal ?</code>
<code>[<Al pha>]</code>	<code>Quote?</code>

Function types

<code>() -> T1</code>	<code>Func<T1></code>
<code>T1 -> T2</code>	<code>Func<T1, T2></code>
<code>T1 * T2 -> T3</code>	<code>Func<T1, T2, T3></code>
<code>lambda x: int & a</code>	<code>(x: int) => a</code>

Type aliases

A type alias becomes a data class in C# with a read-only auto-property named 'Value' that holds the value of the aliased type.

<code>Al pha = int</code>	<pre>publ i c seal ed cl ass Al pha : I Copyabl e<Al pha>, I Equatabl e<Al pha> { publ i c int Val ue { get; } // ... }</pre>
---------------------------	---

Inferred type invariants

The `IsInjective` method is defined in the runtime library.

```
a: nat
a: nat1
a: token
a: seq of char
a: seq1 of char
a: set of nat

a: seq of nat1
a: seq1 of token

a: map nat to nat

a: inmap nat1 to bool

a: nat * bool
a: nat1 * rat * nat1

a: nat | bool
a: <Alpha> | <Bravo>

a >= 0
a > 0
a != null
a != null
a != null && a.Any()
a != null
&& Contract.ForAll (a, _ => _ >= 0)
a != null
&& Contract.ForAll (a, _ => _ > 0)
a != null && a.Any()
&& Contract.ForAll (a,
    _ => _ != null)
a != null
&& Contract.ForAll (a.Keys,
    _ => _ >= 0)
&& Contract.ForAll (a.Values,
    _ => _ >= 0)
a != null && a.IsInjective()
&& Contract.ForAll (a.Keys,
    _ => _ > 0)
a != null && a.Item1 >= 0
a != null && a.Item1 > 0
&& a.Item3 > 0
(a is int && a >= 0) || a is bool
a == Quote.Alpha
|| a == Quote.Bravo
```

Universal quantifications

```
forall a in set s & p (from a in s
    select p).All(_ => _)
forall a, b in set s & p (from a in s
    from b in s
    select p).All(_ => _)
```

Existential quantifications

<code>exists a in set s & p</code>	<code>(from a in s</code> <code> select p).Any(_ => _)</code>
<code>exists a, b in set s & p</code>	<code>(from a in s</code> <code> from b in s</code> <code> select p).Any(_ => _)</code>

Unique existential quantifications

<code>exists1 a in set s & p</code>	<code>(from a in s</code> <code> select p).Count(_ => _) == 1</code>
<code>exists1 a, b in set s & p</code>	<code>(from a in s</code> <code> from b in s</code> <code> select p).Count(_ => _) == 1</code>

Iota expressions

<code>iota a in set s & p</code>	<code>s.Single(a => p)</code>
--------------------------------------	----------------------------------

If expressions

<code>if a then b else c</code>	<code>a ? b : c</code>
---------------------------------	------------------------

Let expressions

A `Let`-expression in VDM-SL becomes an immediately-invoked function expression (IIFE) in C#. The `Let` method is defined in the runtime library.

<code>let a = x, b = y in c</code>	<code>Let(() =></code> <code> {</code> <code> var a = x;</code> <code> var b = y;</code> <code> return c;</code> <code> })</code>
------------------------------------	--

Let-be-such-that expressions

A `let be st-expression` in VDM-SL becomes an immediately-invoked function expression (IIFE) in C#. The `Let` method is defined in the runtime library.

```
let a in set s in c           Let(() =>
                              {
                                var _ = (from a in s
                                           select new { a })
                                           .First()
                                {
                                  var a = _.a;
                                  return c;
                                }
                              })
let a, b in set s be st p     Let(() =>
in c                           {
                                var _ = (from a in s
                                           from b in s
                                           where p
                                           select new { a, b })
                                           .First()
                                {
                                  var a = _.a;
                                  var b = _.b;
                                  return c;
                                }
                              })
```

If statements

```
if a then b                  if (a) { b; }
if a then b else c           if (a) { b; } else { c; }
if a then b else if c then d if (a) { b; } else if (c) { d; }
```

Indexed for-loops

```
for i = a to b do s          for (var i = a; i <= b; i++)
                              { s; }
for i = a to b by c do s     for (var i = a; i <= b; i += c)
                              { s; }
```

Set for-loops

```
for all a in set b do s      foreach (var a in b) { s; }
```

Sequence for-loops

```
for a in b do s             foreach (var a in b) { s; }
```

While-loops

```
while a do s                while (a) { s; }
```

Assignments

```
a := b                      a = b;           // Value types  
c := d                      c = d.Copy();    // Reference types
```

Multiple assignments

The `Atomic` method is defined in the runtime library.

```
atomic (a := x; b := y)     Atomic(() =>  
                             {  
                               a = x;  
                               b = y;  
                             })
```

B Runtime library

This appendix describes the utility classes and methods in the runtime library of the transcompiler. A few of them have not been implemented in the prototype.

`static class VdmSI ToCsUtil i t i e s`

The static `VdmSI ToCsUtil i t i e s` class contains utility methods and extension methods.

<code>ToHashSet()</code>	Converts this instance of <code>I Enumerabl e<T></code> to an instance of <code>HashSet<T></code> .
<code>PowerSet()</code>	Computes the power set of this instance of <code>HashSet<T></code> .
<code>ToDi cti onary()</code>	Converts this instance of <code>I Enumerabl e<KeyVal uePai r<T, U>></code> to an instance of <code>Di cti onary<T, U></code> .
<code>I sl nj ecti ve()</code>	Determines whether this instance of <code>Di cti onary<T, U></code> is injective.
<code>Di cti onaryEqual s(that)</code>	Determines whether this instance of <code>Di cti onary<T, U></code> is structurally equal to <code>that</code> .
<code>Overri deBy(that)</code>	Adds the key-value pairs in <code>that</code> to this instance of <code>Di cti onary<T, U></code> by overriding the existing key-value pairs.
<code>I ntPower(exponent)</code>	Raises this number of the power of <code>exponent</code> . This method complements the built-in <code>Math.Pow</code> method, which only works for <code>doubl e</code> values and not for <code>deci mal</code> .

Continues on the next page ...

Continued from the previous page ...

`Modulo(divisor)`

Computes the modulus between this number and `divisor`.

`Let(() => { ... })`

Executes the given delegate, cf. immediately-invoked function expressions (IIFEs).

`Atomic(() => { ... })`

Disables invariant checking by setting the `AreInvariantsEnabled` property to false, executes the given delegate, re-enables invariant checking by setting the `AreInvariantsEnabled` property to true and asserts that all invariants hold.

sealed class Token

The `Token` data class represents an instance of a token.

`Value`

Holds the underlying token expression.

`Equals(that)`

Determines whether this instance of `Token` is structurally equal to `that`.

`Create(expression)`

Creates a new instance of `Token` with `expression` stored in `Value` as the underlying expression.