AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

CODE GENERATION FROM VDM++ TO TYPESCRIPT

BY

PETER HOLST
20104144

AND

NIKOLAS BRAM
20104278

MASTER'S THESIS
IN
COMPUTER ENGINEERING

SUPERVISOR: PROF. PETER GORM LARSEN
Aarhus University, Department of Engineering

June 6, 2016

Peter Holst	Nikolas Bram	Peter Gorm Larsen
20104144	20104278	Supervisor

Abstract

Increasingly complex computer-based systems provide strong impetus for the engineering community to adopt formal methods, such as the Vienna Development Method (VDM). The VDM constitute a model-oriented approach to the specification, development and verification of computer-based systems. Code generation supports an efficient transition from an abstract model to a software implementation and provides a strong correspondence between the two. While it is evident that formal specifications are useful for safety-critical systems, it is apparent that software development in general can benefit from formal specification techniques. The prevalence of increasingly complex web-based applications might introduce new application areas for the use of formal specifications.

This thesis investigates the possibility of code generation from the specification language VDM++ to the programming language TypeScript primarily used for web development. The thesis presents translation rules and identifies semantics preserving limitations of the translations. The translation rules are implemented in a code generator based on the Overture Code Generation Platform and serves as a proof of concept. A case study is presented in which the translation of a VDM++ model to TypeScript is evaluated. This translation is produced by the proof-of-concept code generator. The result of the evaluation shows that the generated TypeScript version performs in accordance with the VDM++ model, and thereby increases the level of certainty in the correctness of the translation rules presented.

Acknowledgements

We would like to thank our academic supervisor Peter Gorm Larsen for his attention, good advice and invaluable feedback on the report during this thesis project. Additionally, we would like to thank Peter W. V. Tran-Jørgensen for his help setting up and getting started with the Overture Code Generation Platform. Finally, we would like to express our appreciation to family and friends for their patience and support.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the selected subject for this thesis and the motivation behind. The hypothesis and additional goals are presented along with the applied approach to either support or refute the hypothesis. The chapter serves as a general introduction to the thesis and it is therefore relevant to all subsequent chapters.

1.1. Overview

The evolution of technology has led to a rapid development of complex and ubiquitous computer-based systems that influence every aspect of people's life [2, 3]. Increasing demands for dependability serve as great impetus for the engineering community to adopt formal methods in the system development process [4]. Formal methods, such as the Vienna Development Method (VDM) [5], refers to mathematically rigorous techniques used for specification, development and verification of computing systems and software. The VDM constitutes a model-based approach to formal specification by which models are expressed in specification languages. By constructing mathematically models and performing appropriate analysis early detection of specification inconsistencies is possible, before expensive implementation commitments are made [6]. This can contribute to the overall reliability and robustness of a design.

When a certain level of confidence is gained in a model's qualities, a transition from model to implementation is often needed. Since valuable resources have been invested into the modelling of a system, the efforts needed in a realisation phase should be kept at a minimum. An implementation in a high-level programming language can either be made manually using a model as a specification, or it can be derived automatically from the model using code generation. Code generation is a broad term, but generally refers to correctness preserving transformations of code from one representation into another [7]. Therefore, automatic code generation supports an efficient and time-reducing transition from an abstract model to an implementation, and more importantly provides a strong correspondence between the two which minimises the risk of software implementations that deviate from system specifications [8, 9]. However, since formal specifications are based on higher-level abstractions compared to traditional programming languages, code generation from a model to an implementation is a non-trivial task with potential loss of abstraction.

While it is evident that formal specifications are particularly applicable for modelling of safety-critical and mission-critical systems [10, 11], it becomes apparent that software development in general can benefit from the use of suitable formal specification techniques. In recent years, traditional desktop applications have lost ground and are often replaced with web-based applications [12]. A part of the reason can be attributed to the prevalence of better execution environments for web-based applications, such as Google Chrome's V8 JavaScript engine[1] and Microsoft's JScript Engine Chakra[2]. The programming language JavaScript has become the de facto standard for the development of web applications, but was originally intended for casual scripting and not for large applications [13, 14]. To overcome this, new programming languages which enable application-scale web development have emerged, such as TypeScript [15] and Dart[3]. The increasing complexity of web-based applications introduce new application areas where the use of model-based specifications might be beneficial and code generation can serve as a starting point for an implementation.

This thesis project explores the possibility of code generation from the specification language VDM++ to the programming language TypeScript. Being a specification language, VDM++ differs from TypeScript by higher-level abstractions. However, a large subset of VDM++ is directly executable and suited for code generation. Even so, code generation is non-trivial due to language differences. The intent of this thesis is to propose correctness preserving translations from VDM++ constructs to corresponding TypeScript constructs and to identify limitations to these in which the semantics of VDM++ models cannot be reflected in TypeScript source code. Throughout this thesis project, a proof-of-concept code generator has been developed which automates the translation process. The code generator is in this thesis referred to as the *VDM-TS Code Generator*, and outputs from it are presented through examples. Additionally, a case study is presented that is used for evaluation of a VDM++ model.

A characteristic feature of TypeScript is that it compiles to plain JavaScript, before it can be run in an execution environment. Thus, several stages of compilation are involved in this thesis project, and at every stage, there is a potential loss of information. Consequently, JavaScript becomes an implicit target language of the code generator developed during this thesis project. The compilation process is illustrated in Figure 1.1.

The remaining of this chapter is organised as follows. First, in section 1.2, the author's motivation for the chosen subject is presented. Next, section 1.3 describes the hypothesis and additional goals for this thesis project. Section 1.4 defines the scope and is followed by section 1.5 that presents the applied approach for testing the hypothesis and achieving

[1]See `https://developers.google.com/v8/`, accessed 2nd of May 2016.
[2]See `https://github.com/Microsoft/ChakraCore`, accessed 2nd of May 2016.
[3]See `https://www.dartlang.org`, accessed 2nd of May 2016

the goals. Afterwards, related work is briefly mentioned in section 1.6. Finally, reading guidance and the overall structure of the thesis is presented in section 1.7 and section 1.8.

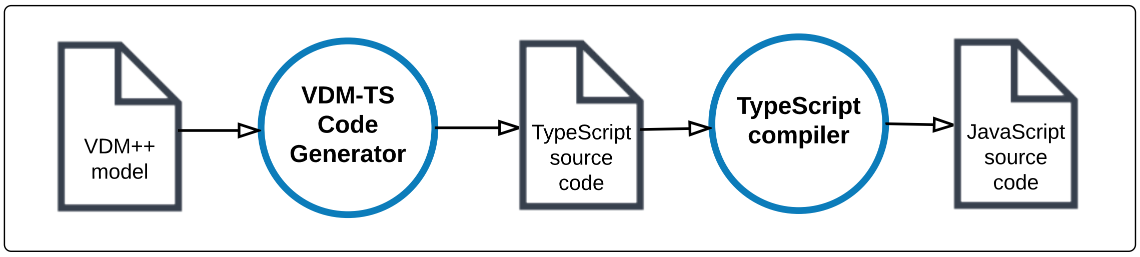


Figure 1.1: Compilation process of this thesis project.

1.2. Motivation

The primary motivation for this thesis project is to improve the tools available for the realisation of model-based specifications. Atwood's law states that: "*Any application that can be written in JavaScript will eventually be written in JavaScript*". Although it might seem as a humorous statement, there is some truth to it. JavaScript is now one of the most widely used programming languages [13] because it has become the "universal virtual machine" allowing JavaScript programs to be run everywhere[4] [17]. However, since JavaScript was never really intended for application-scale development, TypeScript is chosen as target language for the code generation in this thesis project. As the prevalence of web-based applications is expected to grow in the future [17], the increasing complexity of such applications may be hard for developers to comprehend. In order to tackle the increasing complexity, model-based specification techniques can be applied to ensure correspondence between the defined requirements and developed system. Automatic code generation from VDM++ to TypeScript might be beneficial as it will allow VDM++ specifications to be realised on almost every available platform.

Besides the motivation outlined above, the author's personal interest in web development and prior knowledge of JavaScript further motivates the choice of TypeScript as the target language.

1.3. Hypothesis and Goals

The hypothesis of this thesis is that there exists a subset of VDM++, adequate for non-trivial models, that can be translated to TypeScript while preserving the semantics of

[4]JavaScript can both be used for web applications (client- and server side) and for native applications, e.g. on Windows [16].

VDM++.

In this context, a VDM++ model is considered non-trivial when it has qualities of real-life application and contains set-theoretic structures, logic, and operations typically not found in traditional programming languages, such as comprehension, let-bet-such-that and quantified expressions.

In addition to the hypothesis, the following goals have been defined for this thesis project:

Goal 1: Propose translations of VDM++ to TypeScript constructs and identify semantics preserving limitations of these translations. This requires a study of the languages involved.

Goal 2: Develop a proof-of-concept VDM++ to TypeScript code generator based on the findings from Goal 1.

Goal 3: Validate the translation of a non-trivial VDM++ model produced by the proof-of-concept code generator by conducting a case study.

The goals are steps towards supporting or refuting the hypothesis of this thesis.

1.4. Scope

The scope of this thesis project is to propose semantics preserving translations of VDM++ constructs to TypeScript and identify limitations of these. Therefore, it is necessary to invest time in learning the semantics and syntax of the languages involved. The focus in this thesis project is on interesting and challenging construct translations, and not on an exhaustive coverage of VDM++ constructs. The proposed translations will be integrated in a proof-of-concept code generator which will be based on the Overture Code Generation Platform [9]. Part of this thesis project is to evaluate this code generator by conducting a case study in order to validate the correctness of the translations. Concrete use cases of the developed code generator is considered out of the scope of this thesis project.

The reader of this thesis is expected to have a scientific or technological background, such as computer engineering or computer science. Furthermore, the reader should be familiar with the specification language VDM++. Finally, a basic understanding of high-level programming and related paradigms is expected.

1.5. Approach

This section presents the approach that was applied in this thesis. The applied approach for testing the hypothesis and fulfilling the goals of this thesis project can be described as an iterative process. Initially, possible mappings for language constructs that constitute the bare minimum of a VDM++ model were considered. Then, constructs with gradually increasing translation difficulties were treated. Since many translations are similar in nature, translations that can be considered “more of the same” were pushed aside. Finally, VDM++ constructs expected to be particularly challenging for a translation to TypeScript were deliberately selected with the purpose of identifying areas in which semantics cannot be preserved. Translations were continuously integrated in the code generator under development and evaluated for correctness.

The steps involved in the approach are described below.

Language and Literature Study: Gain an understanding of the languages involved in the code generation process necessary to translate constructs from VDM++ to TypeScript while preserving the semantics. Furthermore, acquire knowledge about the Overture Code Generation Platform and the principles behind.

Language Construct Analysis and Design: Compare language constructs, investigate potential translations and identify when the semantics of VDM++ models cannot be preserved in TypeScript.

Implementation: Implement best suited translations in the proof-of-concept code generator. This may require tree transformations of the Intermediate Representation in the Overture Code Generation Platform.

Evaluate: Pass VDM++ models with supported constructs to the code generator under development, and compare the output to the original model manually. Furthermore, run the VDM++ model and its corresponding TypeScript representation in their respective runtime environments[5], and manually compare the execution output.

1.6. Related Work

A lot of work have been carried out in the scene of VDM code generators. Code generation for different VDM dialects is already supported in the Overture Tool suite [8, 6]. This includes the Overture Java Code Generator [9], and other experimental code generators targeting C and C++ [18]. These code generators are built on the Overture Code Generation Platform, which is also the intent of this thesis project. Additionally, code generation from VDM++ to Java and C++ is supported by the commercial VDMTools [19], owned and marketed by CSK Systems Corporation of Japan [8].

1.7. Reading Guide

To ease the reading, this section describes the format and styles used throughout this thesis.

[5]In this thesis project, VDM++ specifications are run by the Overture Interpreter and compiled TypeScript is run both in Google Chrome Version 50 and Node.js v5.5.0

Citation style
Citations in this thesis follow the IEEE citation style. In-text citations have a numbered reference enclosed in square brackets, e.g. [42]. Full details of all references can be found in the bibliography.

Emphasis
Emphasized words are written in *italic*.

Quotations
Quotations from literature or elsewhere is written in *italic* and placed between double quotation marks, such as *"This is a quote"*.

Numbering
Listings, figures, and tables are numbered using the convention $C.N$ where C denote the chapter number and N is the order of the entity.

Code elements and keywords
Elements that are part of VDM++ or TypeScript code are in-text written in `typewriter font`. Code elements that constitute keywords from the respective languages are furthermore written in **`boldface`**. An example is: **`class`** `Alarm`.

Listings
Two listings with different layouts are used throughout this thesis. VDM++ code blocks have a double border on top (see listing 1.1), whereas TypeScript code blocks have a double border at the bottom (see listing 1.2). Additionally, different colours are used for keyword highlighting of the respective languages.

```
-- A VDM++ comment
public foo: real ==> real
foo(x) == return x;
```

Listing 1.1: Example of VDM++ code block.

```
// A TypeScript comment
public foo(x: number): number {
    return x;
}
```

Listing 1.2: Example of a TypeScript code block.

Translation rules
Translation rules are used to concisely describe how constructs in VDM++ can be translated to TypeScript constructs.

Translation 1. Header

Description of a translation.

Limitation boxes
Limitation boxes are used to describe translation limitations, i.e. when the semantics of VDM++ cannot be preserved in the code generated TypeScript source code.

Limitation. Header

Description of a limitation.

1.8. Structure

The structure of this thesis is presented in the swim lane diagram in Figure 1.2. Every chapter is illustrated as a solid box with arrows between them to indicate their dependencies. The text on the left-hand side of the figure describes the main theme of the chapters contained within each swim lane.

Chapter 2: This chapter presents the background, enabling the reader to understand the work carried out in this thesis project. This includes concepts of VDM++, TypeScript and JavaScript as well as theory on the Code Generation Platform used in this thesis project.

Chapter 3: This chapter presents rules and limitations for correctness preserving translations of fundamental constructs from VDM++ to TypeScript. Chapter 3 establish the foundation for chapter 4 and 5.

Chapter 4: This chapter presents a case study used for evaluation of the VDM-TS Code Generator serving as a proof-of-concept by translating a VDM++ model to TypeScript source code.

Chapter 5: This chapter treats selected constructs that are particularly challenging in the translation from VDM++ to TypeScript as a continuation of chapter 3.

Chapter 6: This chapter concludes the thesis. The hypothesis and goals are revisited, and a discussion and evaluation of the thesis project is presented. Additionally, possible directions for future work is presented.

Appendix A: This appendix contains a complete collection of translation rules proposed in this thesis project.

Appendix B: This appendix contains an overview of the VDM-TS Library that provides functionality to reflect VDM++ semantics in the code generated TypeScript.

Appendix C: This appendix contains the entire Alarm model described in chapter 4 and the TypeScript source code produced by the VDM-TS Code Generator.

The source code for the VDM-TS code generator developed during this thesis project can be found at the public git repository: `https://pholst@bitbucket.org/pholst/vdm-ts-cg.git`

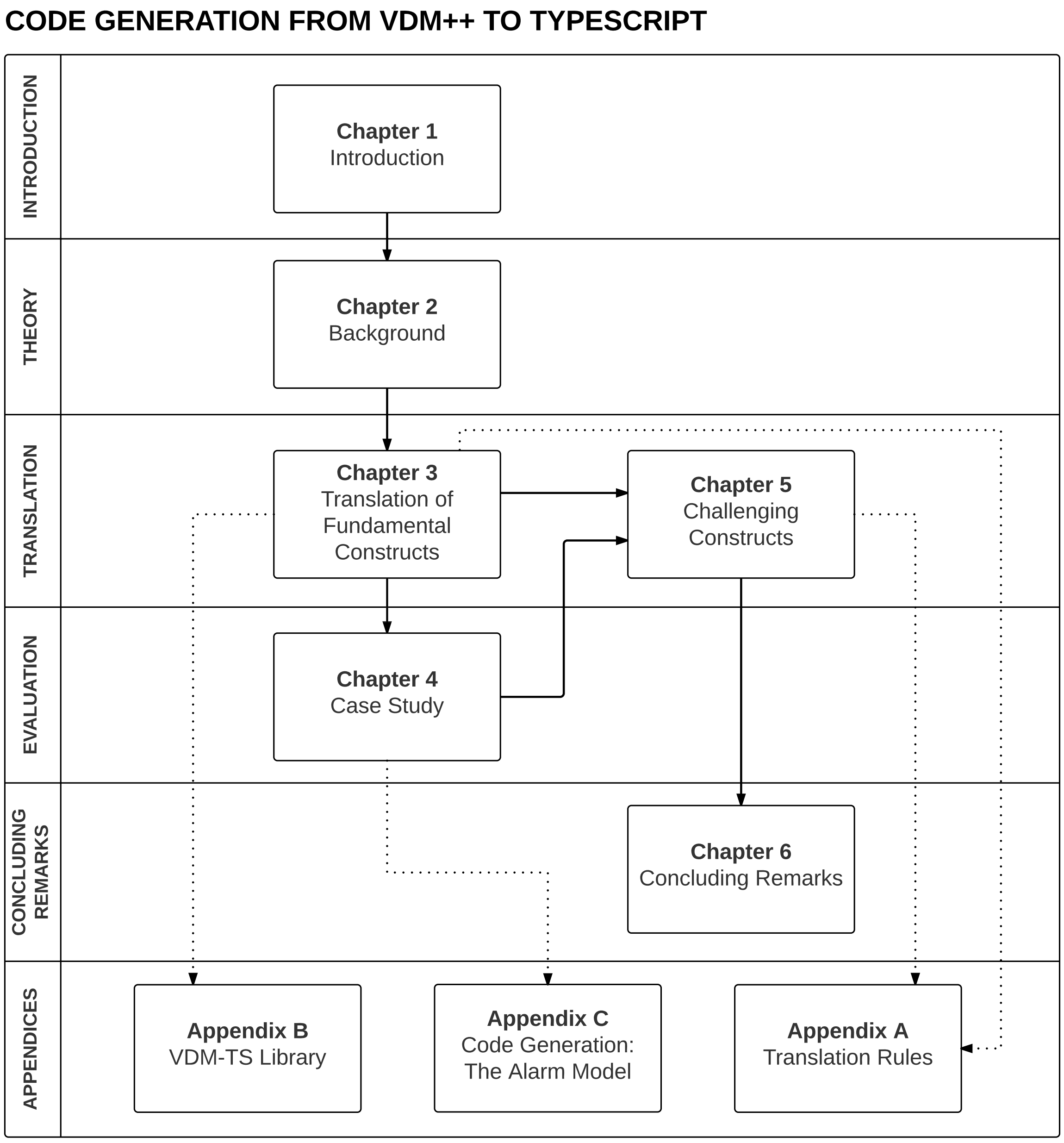

CODE GENERATION FROM VDM++ TO TYPESCRIPT
INTRODUCTION
THEORY
TRANSLATION
EVALUATION
CONCLUDING REMARKS
APPENDICES
Chapter 1 Introduction
Chapter 2 Background
Chapter 3 Translation of Fundamental Constructs
Chapter 5 Challenging Constructs
Chapter 4 Case Study
Chapter 6 Concluding Remarks
Appendix B VDM-TS Library
Appendix C Code Generation: The Alarm Model
Appendix A Translation Rules

Figure 1.2: Thesis structure.

Chapter 2

Background

This chapter presents the background necessary to understand the research conducted in this thesis project. The background constitutes the concepts used in the code generation process as well as the key features and constructs of the languages involved in this process. The knowledge acquired forms the basis for the proposed translations in chapter 3 and chapter 5.

2.1. Introduction

Code generation has proven useful within the field of computer science and computer engineering for reducing trivial work and improving productivity [20]. In general, the term code generation refers to the process of transforming a source language representation to a target language representation while preserving the semantics of the source. This process may happen on different levels. Probably the most common use of code generation is associated with complier theory as a mechanism for producing executable programs from an intermediate representation of source code, e.g. for compiling C++ to machine code [21]. Code generation at a higher level is the generation to source code from a more abstract model representation [22]. This level of code generation can have a positive effect on consistency and quality in software production by letting the human programmer write a model at a higher abstraction level and then automatically generate source code of a lower abstraction level [20]. The code generation of a model representation to source code is the code generation process considered in this thesis.

This chapter presents the background for the VDM-TS Code Generator developed during this thesis project. The code generator is written in Java and is based on the Overture Code Generation Platform (OCGP) [9]. The code generator takes a VDM++ specification as input and translates it to TypeScript source code. Once the TypeScript source code has been generated, it must be compiled to JavaScript before it can be executed by a JavaScript engine. VDM++ models do not have a single point of entry as most modern programming languages e.g. a main method as in Java. This entails that before executable code can be obtained, some handwritten *glue-code* must be added in the generated source code. The amount of handwritten code needed depends on the context in which the generated code is used, and can vary from just instantiating a code generated class, to writing an entire system with the generated code being a small part of that system.

The remainder of this chapter is organised as follows. First, in section 2.2, the Vienna Development Method is briefly introduced and the specification language VDM++ is presented. Next, the OCGP is presented in section 2.3. Afterwards, the programming language JavaScript is presented in section 2.4, as it is the implicit target language of the code generator. Finally, the explicit target language TypeScript is presented in section 2.5.

2.2. Vienna Development Method

The *Vienna Development Method* (VDM) [23] is a model-oriented approach to formal specification used for the development of software and computer systems. The VDM originates from the IBM laboratories in Vienna where it was developed in the 1970's [24]. It allows the specification, modelling and evaluation of systems, which contributes to an increased level of certainty with regards to the validity of the system. VDM models are expressed in a specification language, and currently three dialects of these languages exists. The formal semantics of the *VDM-SL* dialect is standardised by ISO [25, 26]. The *VDM++* dialect is an object-oriented extension of VDM-SL and the *VDM-RT* dialect adds support for distributed real-time systems. VDM models can be analysed and evaluated by the commercial VDMTools [23] or the open-source Overture tool [8].

2.2.1 The VDM++ language

The formal specification language VDM++ offers class-based object-orientation and is therefore structured as a collection of classes that may be organised in hierarchies, e.g. via single or multiple inheritance. VDM++ has a higher abstraction level than ordinary programming languages due to the fact that the modelling language is used to specify only what is relevant of a system. The abstraction is the omission of details that makes the model a *"simplified and idealized description"* [6]. This omission enables implicit construction, where only the function signature is defined and not the function body. Explicit constructs are interesting in relation to code generation, because they are executable and have some behaviour that can be generated in contrast to implicit constructs.

The VDM++ language is a multi-paradigm language that includes both the functional and imperative paradigm. The functional paradigm allows functions to be treated as first-class citizens while the imperative paradigm enables the states of a class to be modified through operations. As an extension of VDM-SL, VDM++ adds the concept of class and objects where classes form the foundation for all VDM++ models as a top-level specification. Classes are comprised of zero or more *member declarations*, which can be divided into multiple groups: definitions of types, values, instance variables, functions and operations. These will together be encapsulated in a scope and define the state and behaviour of the class. The language has lexical scoping with block scopes used when associating names to entities, such as assignment in type definitions. VDM++ has value semantics meaning that all values except the object reference type is passed-by-value. Moreover, the language is statically typed which means that type checking at compile-time. The types on which the checking is performed is founded on mathematical terms and are divided into two categories:

Basic types: These types are atomic values, most of which are known from modern programming languages, e.g. `bool`, `int` and `char`. Whereas the quote type are less known but simply corresponds to an enumerated type with a single value [27].

Compound types: These types are used to represent more complex data structures such as records, functions and collections e.g. sets and mappings, which are also known from modern programming languages. The union type is a special type that defines a union capable of taking several different types. A data structure holding a value of a union type may take the type of any type-member in the set that the union type comprises.

All VDM++ constructs have *value-semantics* [28] entailing that a variable is passed as call-by-value when it is passed as an argument or appear on the right-hand-side of an assignment. The only exception to this is the object reference which is passed as *reference semantics* [28]. VDM++ has the notion of binds which can either be a single binding or a multiple bind. A single binding is a pattern identifier matched to a value of a set or type expression. A multiple bind is essentially the same as a single bind, but with the possibility of matching several patter identifiers to the same set or type [28]. A list of one or more multiple binds are referred to as a *multiple binds*.

2.2.2 Overture

Overture is an Integrated Development Environment (IDE) for working with VDM models. It enables the model to be developed, executed and tested as well as to perform different forms of static analysis on the model [29]. Overture consists of different components all developed in Java, which as a whole form the Overture tool. This includes a code generation platform that enables the translation of VDM models to a specific target language [29].

2.3. The Code Generation Platform

The Overture Code Generation Platform (OCGP) [9] permits Overture to produce implementation specific source code from VDM models. The OCGP architecture facilitates extensions of the code generation capabilities such that new target languages can be supported. The remainder of this section presents the code generation process for the OCGP as well as it's comprising parts.

2.3.1 Overview of the Code Generation process

An overview of the OCGP process is shown in Figure 2.1. The figure illustrates a VDM model passed to the *parser* and *type checker* of Overture, which forms the basis of all operations on the VDM model. The parser produces an *Abstract Syntax Tree (AST)* from the textual representation of the VDM model. The AST is then validated and enriched with additional type information by the type checker. The typed AST is subsequently passed to the OCGP.

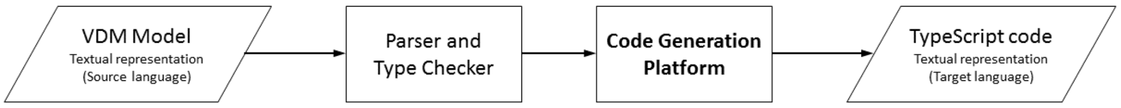


Figure 2.1: Overview of the code generation process in Overture.

From the AST, a common *Intermediate Representation (IR)* is constructed. This IR is afterwards transformed into a simplified IR for a closer resemblance to the target language. The simplified IR is finally passed to a template engine which generates the final source code.

2.3.2 VDM model representation

The AST is used for an internal representation of a VDM model. The AST is created by the ASTCreator tool [30] by using the output from the parser, the AST nodes and the visitors. The generation of AST nodes and visitors is shown in Figure 2.2. Each VDM construct is represented as a AST node and is structured in a bidirectional tree which makes traversals easier. The traversal is made possible by visitors also created by the ASTCreator.

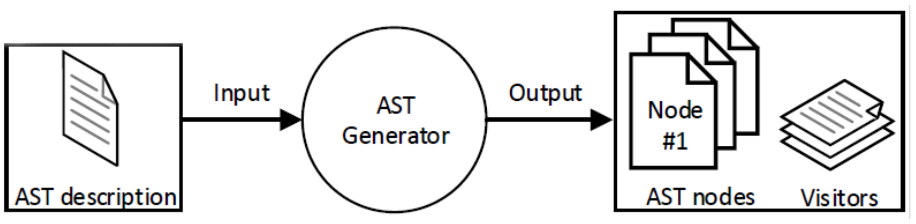


Figure 2.2: The ASTCreator (AST generator) producing AST nodes and visitors. This figure is adapted from [1].

The visitors are implemented using the visitor pattern and they play an important role in the Overture tool, where they are used in the implementation of the type checker and interpreter, among others. The OCGP takes the AST as input and pass it to an IR generator which uses the ASTCreator along with an IR description to generate the IR nodes and visitors as shown in Figure 2.3. These are then applied to generate a simplified representation of the AST [9].

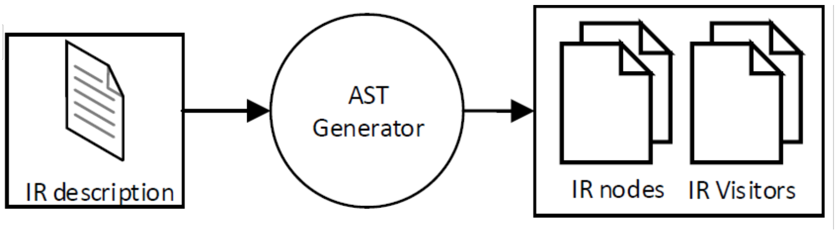


Figure 2.3: IR nodes and visitors generated by the ASTCreator (IR generator). This figure is adapted from [1].

2.3.3 Tree Transformations

Generating source code from VDM constructs are non-trivial if the constructs does not exist in the target language. Code generation across language paradigms is especially challenging, because the constructs present in one paradigm are less likely to be present in other different paradigms. In order to cope with such constructs, the OCGP facilitates the construction of transformations enabling modification of problematic constructs in the IR. This results in a *simplified IR* [9] which are used in the target language backend for source code translation.

2.3.4 Target Language Backend

The translation of the simplified IR to source code is handled by the *target language backend* (or simply backend), which is a separate part of the OCGP architecture. The separation enables reuse of the transformations applied to the IR for other languages with similar constructs, but different syntax, e.g. as Java and C++. Moreover, the backend translates IR to target language source code using template based technology from the Apache Velocity project [31]. Velocity is a Java-based template engine that enables referencing of Java objects through a provided template language. In the backend each IR node is associated with a Velocity template, such that the node is translated according to its related template. Additionally, the generated source code can use a runtime library containing supporting functionality for VDM parts that are not easily translated to the target language. Such a runtime library is also handled in the backend of the OCGP.

2.4. The JavaScript Language

This section describes the basic JavaScript concepts which is directly related to the thesis. JavaScript is a multi-paradigm programming language which is highly inspired by the programming languages Java, Self and Scheme [32]. JavaScript resembles Java in terms of syntax, basic control-flow constructs and memory-safety. However, most notable language design concepts, such as dynamic typing, class-less object-orientation and prototype-based inheritance come from Self. Moreover, the concept of functions in JavaScript being first-class citizens is inspired by the functional programming language Scheme which permits language features such as lambda expressions and higher-order functions.

2.4.1 The ECMAScript Specification

JavaScript has a changeable history due to the intrinsic dynamics and constant evolution of the web. In its early years it became standardised in the ECMAScript specification [33] in an attempt to unify the language across all JavaScript implementers [34]. More precisely, JavaScript is merely a generic name that can refer to any edition of the ECMAScript specification. Most notable editions of the specification are ES3, ES5 and ES2015. The third edition (ES3) is back from 1999 and is known for its many deficiencies. The fifth edition (ES5) resolves many of the ambiguities encountered in ES3[1]. The sixth edition (ES2015) introduces new language features, such as built-in support for classes and modules. However, at the time of writing, still not all features from ES2015 are supported by major implementers [34]. Therefore, this thesis project targets the ES5 edition of JavaScript. In the subsequent sections and chapters, JavaScript is used as a synonym for the 5th edition of the ECMAScript standard.

2.4.2 Execution Environments

JavaScript source code can be executed in many different *execution environments*, also called JavaScript engines. Since JavaScript is the predominant programming language for web applications, the majority of these engines are browsers. Examples of these are Google Chrome's V8 JavaScript engine[2], Microsoft's JScript Engine Cakra[3] and Mozilla's SpiderMonkey JavaScript engine[4]. In addition, JavaScript source code can be executed in environments outside the browser, such as Node.js [35] or RingoJS[5]. This allows server-side applications or desktop applications to be written in JavaScript. Therefore, JavaScript is not just for casual scripting anymore, e.g. animation of web page elements, but can be considered a full-fledged general-purpose programming language.

2.4.3 Types and values

JavaScript is a loosely, dynamically typed language. This means that the type of any variable may change during program execution and that every value has a type at runtime [32]. This differs from static typing in which type analysis is performed prior execution. Additionally, JavaScript has built-in *type coercion* rules which can implicitly convert values of one type into another. Since type coercion is performed on a "as needed" basis, some applications may silently fail with incorrect results if not carefully designed. As an example, one can wonder how an expression like `"1"+2-"3"` can evaluate to `9` [6]. The

[1]`http://wiki.ecmascript.org/doku.php?id=es3.1:history`, accessed 1th of May 2016.

[2]`https://developers.google.com/v8/`, accessed 2nd of May 2016.

[3]`https://github.com/Microsoft/ChakraCore`, accessed 2nd of May 2016.

[4]`https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`, accessed 2nd of May 2016.

[5]`http://ringojs.org/`, accessed 2nd of May 2016.

[6]The problem arises from the fact that the plus operator is both used for numeric addition and string concatenation. This makes sense in a statically typed language such as Java, but less sense in a dynamically

way type coercion works in JavaScript is one of the peculiarities of the language which for long have been subject for criticism [32].

There exist five primitive types in JavaScript which define immutable values: boolean, null, undefined, number and string. All other values are objects. Objects are collections of named properties where each property is either a primitive value or a reference to an object, i.e. an object is a finite mapping from strings to values. A property can also be a function (since functions are objects), and when this is the case, it is referred to as a *method* of the object. Objects in JavaScript are always *passed by reference*. It is important to understand that JavaScript objects are not instances of a class, but instead created either by using an *object initializer* or using a *constructor function* along with the **`new`** operator. Creation using *object literal notation* (an object initializer) is illustrated in listing 2.1.

```javascript
var foo = {                   // foo will reference the new object
    "bar": 3,                 // property named bar
    "baz": function () {      // method named baz
        return true;
    }
};
foo.bar;                      // 3
foo.baz();                    // true
```

Listing 2.1: Simple objects in JavaScript

As opposed to many traditional programming languages, JavaScript does not have *block scope*. Functions are the only way to introduce a new scope (at least in ES5). This is also called *function scope*. Since functions are ordinary objects, they can be passed as arguments to, or returned by, functions. Such higher-order functions are particular useful for operations on arrays, e.g. filtering and apply-to-all (map). In addition, higher-order functions can be used to define closures. A closure is a function having access to its parent scope even though the parent has returned [32]. Closures are possible because JavaScript has function scope and the language is lexically scoped. Closures are commonly used to obtain privacy and modularity in the language, for instance by use of the *module pattern* [32]. Another common pattern in JavaScript is an *immediately invoked function expression* (IIFE), which is a function that is run as soon as it is defined [34]. This is similar to a *lambda abstraction application* in lambda calculus. In JavaScript, IIFE can be used to transform a sequence of statements into an expression.

2.4.4 Prototypal Inheritance

Since JavaScript is a class-less language, the way inheritance is obtained is very different from traditional programming languages with classes. Inheritance in JavaScript is *prototype-based*, which is implemented as a prototype linkage feature allowing objects to inherit properties from other objects. Every object in JavaScript has an internal prototype link which can be set to point to another object, in effect creating a chain of objects. When

typed language. Due to JavaScript's type coercion, the result is `9` because `"1"+2` evaluates to the string `"12"` and `"12"-3` evaluates to the numeric value `9`.

a specific property is absent in an object, JavaScript will look after this property in the prototype chain until it is found, if possible. This model of programming is often called *delegation* or *prototypal inheritance* [32].

JavaScript is in a sense a very expressive language, because various patterns can be used to simulate the way “classical” languages obtain inheritance through classes. JavaScript has the notion of *constructor functions*, which can instantiate objects by using the **`new`** keyword. Every constructor function has an associated prototype, and new instances of a specific constructor function inherits properties from this prototype. Therefore, common functionality between instances should be put on the prototype of a constructor function.

In order to mimic class-based inheritance in JavaScript, it is possible to set up an object hierarchy as illustrated in Figure 2.4. All the entities in the figure represent JavaScript objects, but their shape indicate different roles. The leftmost objects (rectangles) are constructor functions, the objects in the center (ovals) are the prototypes associated to each constructor function, and the rightmost objects (circles) are instances of a constructor function. When **`new`** `B()` is called, a new instance is created using the constructor function `B` as a skeleton, and the instance’s internal prototype link `[[proto]]` is set to the prototype of `B`. Thereby, instances (such as `b1`) inherits properties from `B.prototype`. In this example, `B.prototype` can additionally inherit properties from `A.prototype`, and `A.prototype` can inherit properties from `Object.prototype`, because there is an internal link between them. Thus, if `b1.toString()` is called, then JavaScript will first search for this method in `b1`’s own properties, and since the method is not present in `b1`, the internal link will be followed until `toString()` is found in `Object.prototype`. As a result, `b1` has inherited the `toString` method

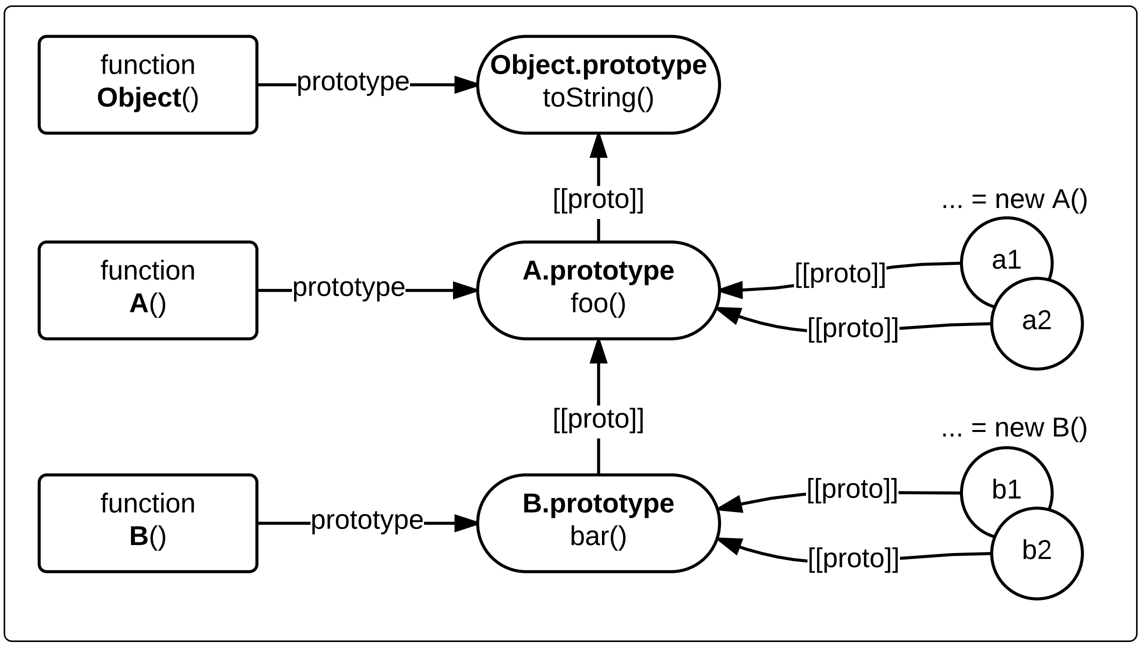


Figure 2.4: Simulation of class-based inheritance in JavaScript.

through *delegation*. It should be noticed how the constructor functions and prototypes mimic class-based inheritance. The example corresponds to a conventional class-hierarchy where `B` is a subclass of `A` and `A` is a subclass of `Object`. In addition, it should be noticed how the internal prototype link constitute *single inheritance* by its construction.

Simulation of class-based inheritance in JavaScript can be fairly intricate and somewhat esoteric. This is presumably why classes are built-in constructs in the ES2015 specification and in languages that compiles to JavaScript, e.g. CoffeeScript[7] and TypeScript [15].

2.5. TypeScript

> *"What if we could strengthen JavaScript with the things that are missing for large-scale application development like static typing, classes, modules... that's what TypeScript is about."* [36].
>
> — Anders Hejlsberg

This section describes TypeScript which is the explicit target language of the code generator developed during this thesis project. TypeScript is an open-source programming language defining a typed superset of JavaScript that compiles to plain JavaScript. Hence, one can basically paste existing JavaScript programs into TypeScript source code. The main purpose of TypeScript is to address the perceived shortcomings of JavaScript for developing and maintaining large-scale applications [15]. TypeScript enriches JavaScript with optional *static typing* and additional language constructs that are aligned with the ES2015 specification. The characteristics of the type system and relevant language construct extensions added to TypeScript is described in the subsequent sections.

2.5.1 The type system

The type system of TypeScript enables static analysis at compile time which help catching representational mismatches caused by silent type coercion in JavaScript. According to the official design goals of TypeScript, the intention is not to be a language in its own right, but to *"Emit clean, idiomatic, recognizable JavaScript code"* [37]. This leads to a set of distinctive properties regarding the type system:

Structural typing: Compatibility of types in TypeScript is defined in terms of structural typing rather than nominal typing known from most industrial mainstream languages such as Java and C#. This means that TypeScript relates types based on their members and not names, which seems as a reasonable fit for JavaScript code where anonymous objects are often used.

Type inference: Type inference is a built-in mechanism in TypeScript that can deduce type information when none is explicitly declared. The type inference is based

[7]`http://coffeescript.org/`, accessed 28th of April 2016.

on a best common type algorithm and essentially minimizes the number of type annotations that need to be provided.

Type erasure: Once compiled, no traces of TypeScript types are left in the emitted JavaScript code. Consequently, no checks of those types can be made at runtime. Instead, dynamic runtime checks have to be done by JavaScript reflection techniques [38].

Gradual typing: Gradual typing allows distinction between statically and dynamically types in a program. The type **`any`** in TypeScript is used to represent a statically-unknown type.

It is particularly interesting that TypeScript type information is erased at compile time. This implies that run-time casts cannot possibly be supported in the language. Casting refers to the process of explicitly converting one type into another and are generally used in places where type information might be lost. Explicitly type conversion can be achieved by *type assertions* in TypeScript, however such conversion is merely static. As a result, no invalid cast exceptions will be thrown at runtime as in other statically typed languages, such as Java and C#.

2.5.2 Classes

In addition to the static type system, TypeScript introduces the notion of classes. Thus, TypeScript looks syntactically familiar to other class-based languages such as Java. Classes in TypeScript can be seen as syntactic sugar for JavaScript programs as they compile to idiomatic JavaScript patterns that simulate classes. In listing 2.2, an example of a simple hierarchy comprised of two classes is shown. When these classes are compiled to JavaScript, this hierarchy will be represented as in Figure 2.4.

```
class A {
    public foo(): number {
        return 24;
    }
}
class B extends A {
    public bar(): number {
        return 42;
    }
}
var b1 = new B();
b1.foo();     // 24
```

Listing 2.2: Simple class hierarchy in TypeScript.

2.5.3 Modules

Modules in TypeScript are useful large-scale applications in terms of modularity and maintainability as they provide code reuse and isolation of components. Modules are divided into two types: internal and external modules, but are referred to as *namespaces* and *modules*, respectively. Namespaces are internal modules that enables modularity by separating components and enforce encapsulation at runtime. This is achieved by using the *module pattern* in the emitted JavaScript as it encapsulates properties using closure variables. External modules or simply *modules* are defining their own scope within a separate file and uses directives to declare dependencies at file level. Modules are dependent on a *module loader* (such as CommonJS build-in to Node.js [35] or RequireJS[8] for in-web browser use) for locating and execution dependencies before the module itself is executed.

[8]See `http://requirejs.org/`, accessed 10th of May 2016.

Chapter 3

Translation of Fundamental Constructs

This chapter presents rules and limitations for correctness preserving translations of fundamental constructs from VDM++ to TypeScript. Alternative ways to represent VDM++ constructs are elaborated based on the gained understanding of language concepts and features presented in chapter 2. The rationale behind the best suited translations is presented, and these translations are implemented in the VDM-TS Code Generator. This code generator, and thereby the translations, are evaluated by a case study in chapter 4. The set of translation rules is extended in chapter 5, where particularly challenging translations of VDM++ constructs are presented.

3.1. Introduction

Programming language constructs are the building blocks by which a computer program is constructed or as S. K. Bensal describes it: *"A language construct is a syntactically allowable part of a program that may be formed from one or more lexical tokens in accordance with the rules of a programming language."* [39]. These constructs describe the control structure of the program, i.e. the syntactic form in the language used to express the flow of control. The process of translating a source language into a target language is accomplished by having a translation for each language construct. The translation is used to convert the language construct into one or more constructs in the target language that obtain a semantically equivalent representation.

The focus of this chapter is the translation of VDM++ to TypeScript. Constructs in VDM++ and TypeScript are diverse, which makes translation challenging because fewer possibilities for translating the source language exists. Since both VDM++ and TypeScript include the imperative paradigm, the translation of statement constructs is straightforward and have in most cases just a syntactical difference. Expressions on the other hand are less trivial to translate.

The remainder of this chapter is organised as follows. First, in section 3.2, translations of basic data types in VDM++ are presented. As classes constitute a top-level specification, translation of these will be addressed in section 3.3. Then, in section 3.4, translations of compound data types is presented. Afterwards, section 3.5 presents how value semantics of VDM++ can be obtained in TypeScript. Finally, in section 3.6, translation of selected VDM++ expressions are presented.

The translation will be presented in the following way. First, general considerations and alternative translations are discussed. Then, the best suited translation is summarised in a translation rule. This is followed by an example of the translation. Finally, eventual limitations are presented related to the translation rule. Some of the translations are similar in nature and are therefore not presented in this chapter. A collection of all translation rules can be found in appendix A.

3.2. Basic Data Types

The basic data types in VDM++ are mapped to their most suitable corresponding primitive data types in TypeScript as shown in Table 3.1. Quote types are special constructs that has no counterpart in TypeScript. Thus, it is translated to a class that implements the singleton design pattern [40], allowing only one instance of every quote class. These classes are placed in a global namespace enabling access for all program entities. The token type is translated to a class with a single instance member. Translation of quote and token types are summarised in translations rule 1 and 2, respectively. In VDM++, strings are represented as a **`seq of char`**. Hence, this sequence is mapped to the TypeScript string type.

VDM++	TypeScript
`bool`	**`boolean`**
`real`	**`number`**
`rat`	**`number`**
`int`	**`number`**
`nat`	**`number`**
`nat1`	**`number`**
`char`	**`string`**
Quote	**`class`**
Token	**`class`**
`seq of char`	**`string`**

Table 3.1: VDM++ basic data types and their TypeScript counterparts.

Translation 1. Quote Types

The quote type in VDM++ is code generated to a class that implements the singleton pattern.

Translation 2. Token Types

Let t be a token type in VDM++ wrapping an arbitrary expression e. Then t is code generated to a class derived from the base class `Token` containing an instance member that holds the meaning of e.

3.3. Classes

Classes are a fundamental construct in VDM++ as a top-level specification is composed of one or more classes. Since TypeScript is a class-based object-oriented language, a one-to-one translation an obvious choice.

Translation 3. Classes

Classes in VDM++ become the equivalent class construct in TypeScript.

Classes in both languages may contain instance and static members, and the accessibility of these is determined by the notion of access modifiers. Albeit seemingly similar, a VDM++ class include a more exotic set of members. The most interesting ones are considered the following subsections.

3.3.1 Type definitions

Type definitions introduce a name which can be used as a synonym for the type, also referred to as a *named type*. TypeScript supports *type aliases* allowing any kind of type to be named. While types in VDM++ belong to the class defining them, it is not possible to declare type aliases within the body of a class in TypeScript. Type aliases belong per default to the global namespace, which will cause a duplicate identifier error in the generated code when two VDM++ classes use the same name to define a type. Three approaches are considered to overcome this issue.

1. Let the code generator perform resolution and substitution of named types.
2. Decorate type names with additional information about class affiliation.
3. Introduce a namespace associated to each class exporting its named types.

The first approach eliminates the need of type definitions but compromises readability in the generated code. The second approach involves deciding upon a name mangling strategy to be implemented in the code generator, e.g. prepending the name of defining class to the type name. The third approach is basically a variant of the second but is favourable as it utilises *namespaces* which is one of TypeScript's built-in structuring mechanisms.

Hence, named types from VDM++ can be preserved in TypeScript. In addition, this approach will generally produce less code compared to the other approaches. Listing 3.2. The translation of type definitions is summarised in translation rule 4 below.

Translation 4. Type definitions

Type definitions become exported *type aliases* in the associated namespace of a class.

```
class A
types
    Amount = real
end A
```

Listing 3.1: Type definition within VDM++ class.

```
class A { }
namespace A {
    export type Amount = number;
}
```

Listing 3.2: Type alias associated with class through namespaces in TypeScript.

Limitation. Recursive type definitions

Recursively-defined data types cannot directly be translated since type aliases in TypeScript are prohibited to circularly reference themselves. This restriction implies that recursive types in general is not supported by the code generator. However, some support of recursive types is possible because TypeScript classes in their internal structure can reference themselves in effect producing infinite nesting of types [41]. This is relevant because classes are used to represent VDM++ record types. See subsection 3.4.2.

3.3.2 Value definitions

Value definitions define constant values. Two feasible translations are considered. One possibility is to represent VDM++ values as static instance members of a TypeScript class as proposed in [6]. This allows a value to be referenced without instantiating a class. However, TypeScript does not have the notion of constant class members. Hence, a static instance member can be changed at any time. Therefore, this is not an ideal representation of VDM++ value definitions. Another possible translation is to represent values within the namespace of the defining class. This allows the **`const`** keyword to be used, which statically ensures that a value is constant. To ensure a value remains constant at runtime, the `Object.freeze`[1] functionality from JavaScript is applied making the value immutable. This translation is a better alternative and is therefore summarised below in translation rule 5.

Translation 5. Value definitions

Values become an exported *const declaration*, applied with `Object.freeze`, residing in the associated namespace of a class.

An example of the code generated value definition is shown in Listing 3.3 and Listing 3.4, where it can be seen that the value in TypeScript is no longer protected by an access modifier.

```
class A
values
    private foo : bool = true
end A
```

Listing 3.3: A value definition in VDM++.

```
class A { }
namespace A {
    export const foo: boolean = Object.freeze(true);
}
```

Listing 3.4: Constant value defined within the associated namespace of the class.

Limitation. Access control

Declaring the value within the namespace makes it public accessible even if it is protected by an access modifier in VDM++.

3.3.3 Instance variable definitions

Instance variables in VDM++ are directly translated to instance members of a TypeScript class. It is necessary to explicitly state all access modifiers on the translated instance members, because TypeScript and VDM++ have conflicting default behaviour with regards to these modifiers. TypeScript instance members are by default public, whereas VDM++ by default has private instance variables. This leads to translation rule 6.

[1]See `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze`, accessed 5th of April 2016.

Translation 6. Instance variables

Instance variables become *instance members* of a TypeScript class with explicit access modifiers similar to VDM++ either declaring public, protected or private accessibility.

Listing 3.5 and listing 3.6 shows a translation example of instance variables.

```
class A
instance variables
    public bar : int := 42;
end A
```

Listing 3.5: VDM++ instance variable definition.

```
class A {
    public bar: number = 42;
}
```

Listing 3.6: Instance member of a TypeScript class.

Limitation. Static Instance Variables

Instance variables in VDM++ can be declared **`static`**. This behaviour cannot be reflected in TypeScript.

3.3.4 Function and operation definitions

VDM++ distinguish between function and operation definitions whereas TypeScript only has the notion of methods. Hence, both function and operation definitions are translated to TypeScript methods. Methods that represent function definitions are however declared as **`static`** class members of the defining class, which makes them accessible as a class attribute. Declaring an operation with the same name as the class will make the operation act as the class constructor. The constructor must be translated to a non-static constructor method in TypeScript permitting class instances to be created. Access modifiers for both function and operation definitions must be handled as described above in subsection 3.3.3. This leads to translation rule 7.

Translation 7. Function and operation definitions

Both VDM++ functions and operations are code generated as methods of a TypeScript class. Methods representing functions are declared with the `static` keyword. An operation with an identical name to the class is translated to a constructor in the TypeScript class.

The translation of both a function and operation from is shown in listing 3.7 and listing 3.8.

```
class A
functions
    foo : () -> bool
    foo() == true;
operations
    bar : int ==> int
    bar(x) == return x;
end A
```

Listing 3.7: Functions and operations in VDM++.

```
class A {
    private static foo(){
        return true;
    }
    private bar(x: number){
        return x;
    }
}
```

Listing 3.8: Static method and constructor method in TypeScript.

Limitation. Purity

Purity of VDM++ *functions* and *pure operations* cannot be reflected. While some research has been conducted for purity analysis in JavaScript [42], there is no support in the language for detection or enforcement of purity.

Limitation. Overloading

Functions and operations can be overloaded in VDM++. This behaviour cannot be reflected in TypeScript. The reason is that JavaScript does not have the notion of method overloading. JavaScript objects can only have one property (e.g. method) with the same name.

3.4. Compound Data Types

Compound data types are built-in types in VDM++ with the primary purpose to act as a container for other kinds of data. Most modern programming languages have support for compound data types such as collections (sets, lists, maps) either as built-in or library constructs. On the contrary, TypeScript has a very limited set of built-in data types. Despite the fact that millions of external libraries exist, it proves challenging to find libraries that provide exactly the needed functionality with respect to the VDM++ compound data types. Thus, for TypeScript to support these types, there is a need to implement an external TypeScript library. This library will be referred to as the *VDM-TS Library*. An overview of the functionality of the library can be found in appendix B.

3.4.1 The VDM-TS Library

The purpose of the VDM-TS Library is to provide functionality that can reflect the semantics of the compound data types in VDM++. This entails that the VDM-TS Library shall handle equality of these types. In addition, the library must define a cloning function in order to obtain the effects of value semantics in VDM++. Value and reference semantics is described in more detail in section 3.5.

Cloning in JavaScript is in principle fairly easy because properties of one object can be iterated and put onto a freshly created one. However, some properties can be non-enumerable and special attention has to be given in order to avoid erroneously copying of properties which should be left in the prototype (or intermediate prototypes) of an object. It becomes even more heavy-handed when dealing with nested data structures, which will require a recursive clone function to be defined.

Equality of non-primitive objects can also be fairly intricate because comparison of objects is based on its reference. Hence, comparison of two distinct objects that are structurally the same evaluates to false. Since object equality is important in various applications such as JavaScript test frameworks, many libraries provide functionality for structural comparison of deeply nested objects. Unfortunately, none of these deep equality library algorithms seem suitable for comparing a construct such as the VDM++ set collection where the order of elements is non-essential.

To cope with these challenges, the VDM-TS Library is inspired by the way class instances in Java have `equals` and `clone` methods either inherited from the root object or overwritten at class declaration [43]. Hence, all representations of compound data types within the VDM-TS Library can be cloned and tested for equality, in addition to other operations that may belong to a specific type. It shall be noted that the VDM-TS Library is referenced by an underscore, which is deliberately chosen because VDM++ does not allow a leading underscore in identifiers. Thus, potential name ambiguities are prevented in the translation.

3.4.2 Record Types

Objects in TypeScript and records in VDM++ have an interestingly strong resemblance. For each *property* in a TypeScript object, the key is a name and the associated value can be of any type. Hence, an object property corresponds to a field in a VDM++ record, making objects and records conceptually equivalent. VDM++ records are tagged with an identifier and therefore is TypeScript *interfaces* or *aliased object type literals* candidates for record type translation. However, both interfaces and object type literals are subject for TypeScript type erasure. As a consequence, no type information is present at run-time. The translation will work out for a subset of VDM++, but this loss of type information is undesirable. By representing the VDM++ record by a TypeScript class, it is still possible to retrieve type information once TypeScript is compiled to JavaScript. This leads to the translation rule 8.

Translation 8. Record types

Let R be a VDM++ record (composite) type with fields $f_1, \ldots, f_n$ with corresponding types $t_1, \ldots, t_n$. Then R is code generated to a TypeScript class with identical name and with instance members $f_1, \ldots, f_n$ annotated with $t_1, \ldots, t_n$. All classes that represents a record type derives from base class `Record`.

```
class A
types
    Foo ::  bar : real
            baz : bool
end A
```

Listing 3.9: A record type in VDM++.

```
class A { }
namespace A {
    export class Foo extends Record {
        constructor(public bar: number, public baz: boolean) {
            super();
        }
    }
}
```

Listing 3.10: A record class residing in the namespace associated with the class.

3.4.3 Set Types

The set type in VDM++ represents a collection of unordered values of the same type. There is no similar built-in type in TypeScript. Hence, a `Set<T>` class is present in the VDM-TS Library in order to mimic the semantics of VDM++ sets. The type parameter `T` is used to statically ensure that the set is comprised of the same type. All operators that can be applied on VDM++ set types are available in the VDM-TS Library. A full overview of operators can be found in appendix section B.2. The translation of set types is summarised in translation rule 9.

Translation 9. Set types

Let S be a VDM++ set type comprised of arbitrary complex elements $e_1, \ldots, e_n$ of same type t. Then S is translated to the generic class `Set<T>` where the type parameter `T` becomes t. Values are created by making instances of the class and passing $e_1, \ldots, e_n$ as arguments to the constructor. Explicit type annotation of `T` is only needed when t is a union type.

Values of set types can be constructed in three ways; either by explicitly enumerating the elements, by applying operators on existing sets or by a comprehension from another set. Examples of the two first ways of creating set values are shown in the code snippets below. The third way, *set comprehension expressions*, is given special attention in subsection 3.6.3.

```
values
    a : set of int = {1, 2, 3};
    b : set of int = {3, ... , 5};
    c = {{1, 2}, {3, 4}};
    d : bool = 2 not in set b;
    e : set of int = dunion { a, b, {5, 6} };
    f = {1, true, 2, false, 3};
    g : set of (int|bool) = a union {true, false};
    h : bool = d = e
```

Listing 3.11: Examples of set types and values in VDM++.

```
...
    a : Set<number> = new Set(1, 2, 3);
    b : Set<number> = _.range(3, 5);
    c : Set<Set<number>> = new Set(new Set(1,2), new Set(3, 4));
    d : boolean = !b.inset(2);
    e : Set<number> = a.union(b.union(new Set(5, 6)));
    f = new Set<boolean|number>(1, true, 2, false, 3);
    g : Set<number|boolean> = a.union(new Set(true, false));
    h : boolean = _.equals(d, e);
```

Listing 3.12: Corresponding translation of set types and values in TypeScript.

3.4.4 Function Types

Being functional languages, both VDM++ and TypeScript allow functions to be passed as arguments to, or returned from other functions. Both languages offer a *function type*, however, the way function types are declared in TypeScript differ from VDM++, because function types in TypeScript require parameter names. This difference is exemplified in listing 3.13 and listing 3.14.

```
real * real -> bool
```

Listing 3.13: Function type in VDM++.

```
(a: number, b: number) => boolean
```

Listing 3.14: Function type in TypeScript.

Since parameter names are required in TypeScript, it is unclear how function types shall be code generated. It is possible to leave out explicit function type annotations, and let the TypeScript compiler provide this type information using the built-in type inference mechanism. This is however not always possible. For example, since VDM++ type definitions are represented as type aliases in the TypeScript translation, there is no value from which the function type can be inferred. Therefore, parameter names have to be explicitly defined in some situations in order to produce valid TypeScript. To circumvent this, arbitrary unique parameter names in a function type can be generated. This is possible because parameter names just are meant to help readability. As long as the parameter types line up, the type is considered valid for the function [44]. An issue caused by this approach is that otherwise meaningful names will be hidden from a developers' perspective when using tooling support such as auto-completion. While function types can be automatically inferred in some situations, generating unique parameter names is a better choice for consistency. This leads to the translation rule 10 below.

Translation 10. Function types

Let F be a VDM++ *function type* from (tuple) type A to type B. Then F is code generated as a TypeScript function type where $A_1, \ldots, A_n$ become parameters with unique parameter names $x_1, \ldots, x_n$ and B becomes the return type.

Different usage of VDM++ function types and the corresponding TypeScript version is shown in listing 3.15 and listing 3.16, respectively.

```
class A
types
    public F = real * real -> bool

instance variables
    foo : F := lambda a:real, b:real & a < b

functions
    public bar: real -> (real -> real)
    bar(a) == lambda b:real & a + b
end A
```

Listing 3.15: Example of function types in VDM++.

```
class A {
    private foo: A.F = (a: number, b: number) => a < b;

    static bar(a: number): (x1: number) => number) {
        return (b: number) => a + b
    }
}
namespace A {
    export type F = (x1: number, x2: number) => boolean;
}
```

Listing 3.16: Corresponding example of function types in TypeScript.

3.4.5 Union Types

As mentioned in subsection 2.2.1, VDM++ includes a union type, which is a special compound data type for holding multiple types. Two feasible translations are considered. The first is the translation to the `Object` type, which is a strategy used by the Overture Java Code Generator [45]. The `Object` type is the root type in TypeScript and permit a value to be of any type. This is not a good representation of the union type. The other translation is to make use of TypeScripts build-in union type, which permit a value to take one of multiple fixed types as declared in the union type. However, the VDM++ and TypeScript union types are not completely equivalent. The TypeScript union allows only access of non-disjoint properties of a union type.

When the TypeScript union is used in the context of a method, it can cause an unsound, non-deterministic state in the type system, because *"a union type U is assignable to a type T if each type in U is assignable to T."* [41]. Hence, the union type cannot be narrowed and assigned to a more specific type unless additional measures are taken. These additional measures involve the use of *type assertions* i.e. explicitly stating the exact type of the value. This gives the type system the required information as shown in listing 3.17.

```
foo(x: number|boolean): number {
    return x as number;
}
```

Listing 3.17: A method taking a union type as argument and returning a number.

However, type assertions are not enough as they are discarded at runtime due to type erasure. In order to ensure equivalent semantics, code generation of *type guards*[2] are required. The type guards surround all values of a union type, ensuring that the type system has information about the exact type of the value within the type guard. This eliminate type errors and narrows the union type to one specific type, allowing exceptions to be thrown if the type of the value is incorrectly accessed at runtime. The use of type guards may involve either **`typeof`** and **`instanceof`**. For more complex types, such as `Set<`**`number`**`>` where **`typeof`** and **`instanceof`** are not enough, *user-defined type guards* are constructed by using *is expressions* as described in section 5.4. Code generation of the union type is summarised in translation rule 11.

Translation 11. Union types

The VDM++ union type comprised of the types $t_1, ..., t_n$ is code generated to the built-in union type in TypeScript.

The translated union type is used in different contexts and it is therefore always necessary to wrap the union type value in type guards. Listing 3.18 shows an operation `foo` taking a union type as input and returning a **`int`**. This simple example is code generated as shown in listing 3.19 with a type guard for **`number`** and one for **`boolean`**. In the case a boolean value is passed to `foo`, an exception is thrown because the return type must be a **`number`**.

```
...
operations
    foo : int|bool ==> int
    foo(x) == return x
...
```

[2]Type guards are expression patterns causing types of variables to be narrowed to more specific types.

Listing 3.18: Operation taking a union type as argument.

```
foo(x: number|boolean) : number {
    if(typeof x === "number") {
        return x;
    } else if(typeof x === "boolean") {
        throw "Error: Cannot convert boolean to number";
    }
}
```

Listing 3.19: Method that takes a union type as argument and returns a number.

3.5. Value and Reference Semantics

Both VDM++ and TypeScript have value and reference semantics as described in section 2.4 and subsection 2.2.1. Since almost every value in JavaScript are objects, reference semantics is especially apparent in TypeScript. Therefore, representation of VDM++ value types in TypeScript requires special attention. The simplest way to obtain the effect of value semantics in TypeScript is to provide a `clone` function which is applied on objects, when they are passed as arguments to methods or they appear on the right-hand-side of an assignment statement.

Translation 12. Obtaining Value Semantics

Apply the VDM-TS Library `clone` function on TypeScript objects that represent a value type in VDM++ when it:

- is passed as an argument to a function.
- appears on the right-hand-side of an assignment.

Object Reference Type

The object reference type is denoted by a class name and a value can be regarded as a reference to an object. The object reference type is represented in TypeScript as classes. Instances of a class can be seen as values of this type. Since JavaScript has a sequential execution of program flow, the order in which classes are declared are important in contrast to VDM++. This is problematic if classes are dependent on each other. Two possible solutions for handling class dependencies is considered. The first is to code generate all VDM++ classes into one large TypeScript file and ensure that the classes are inserted in the right order. This approach is fairly easy, but decrease modularity and cannot be used to resolve mutually recursive dependencies. The alternative approach encompasses utilisation of a *module loader* for dependency management, such as CommonJS [35] and RequireJS[3]. This allows classes to be contained in its own file. Additionally, module loaders allow TypeScript programs to have circular references. The first approach is implemented in the VDM-TS Code Generator due to its simplicity. However, the latter approach is a more viable option.

Translation 13. Class Dependencies Caused by Object References

Each code generated TypeScript class is contained in its own file specifying its dependencies using the TypeScript module import declaration `import`.

3.6. Expressions

This section presents translations of different VDM++ expressions. There are no similar expressions in TypeScript and therefore careful deliberation is needed to represent the semantics of these VDM++ expressions.

3.6.1 Let Expressions

Let expressions in VDM++ are useful for creating local bindings not seen elsewhere. As mentioned in section 2.4, one of the distinctive features of JavaScript is that it has *function scope* only. TypeScript enriches JavaScript with a **`let`** keyword that can be used to define variables that abide by *block scope*. However, while this keyword is suitable for translation of *let statements* in VDM++, it cannot be used for translation of *let expressions*. Instead, the semantics of *let expressions* can be obtained by using the common design pattern IIFE mentioned in section 2.4. The translation rule for *let expressions* is summarised in translation rule 14.

Translation 14. Let Expressions

Let l be a *let expression* in VDM++ that denotes an expression e involving the pattern identifiers $i_1, \ldots, i_n$ of the patterns $p_1, \ldots, p_n$ matched against the corresponding expressions $e_1, \ldots, e_n$. Then l is translated as an immediately-invoked function expression where the arguments become $e_1, \ldots, e_n$, the formal parameters become $i_1, \ldots, i_n$ and the body of the function becomes e.

[3]See `http://requirejs.org/`

```
let a:int = 1, b:int = 2 in a+b+4
```

Listing 3.20: Simple let expression in VDM++.

```
((a: number, b: number) => a+b+4)(1, 2)
```

Listing 3.21: Immediately invoked function expression in TypeScript.

3.6.2 If Expressions

Conditional constructs in VDM++ can both be expressed in terms of statements and expressions. On the contrary, the primary conditional construct in TypeScript is *if statements* because of the language's imperative nature. One way to transform *if statements* into expressions in TypeScript is to wrap it in an IIFE and return from both branches. However, TypeScript cannot automatically infer the return type of a function when multiple return statements have no common type. Thus, the OCGP has to explicitly annotate the return type of a function when the result is a union type. Albeit being a perfectly viable option, an alternative translation is the *ternary operator* which can be used to express inline conditionals in TypeScript. This is a better translation due to its succinctness and because TypeScript can infer a union type of such expression. This leads to translation rule 15.

Translation 15. If Expressions

If expressions become an inline conditional expression using the ternary operator.

Simple examples of *if expressions* can be seen in the code snippets below.

```
if x < 1 then true else 42
if x > 3 then 42 else (if x < 0 then 41 else true)
```

Listing 3.22: Examples of *if expressions* in VDM++.

```
x < 1 ? true : 42
x > 3 ? 42 : (x < 0 ? 41 : true)
```

Listing 3.23: Corresponding TypeScript translation of listing 3.22.

3.6.3 Set Comprehension Expression

As mentioned in subsection 3.4.3 above, sets can be constructed by a comprehension from other collections. The *set comprehension expression* constructs a set by evaluating an expression on all bindings for which a predicate is true. A binding matches a pattern to a value and is either a *set bind* or *type bind*. However, set binds are of most interest in this thesis because bindings over infinite types cannot be executed by a VDM++ interpreter[4].

It is clear that translation of set comprehension expressions must be based on the `Set<T>` class from the VDM-TS Library. Set comprehension can in functional programming be seen as filter and map operation applied (in that order) to an existing set. Because the `Set<T>` class is built on top of JavaScript arrays and all arrays inherit the higher-order map and filter functions from the `Array` prototype, translation of set comprehension is straightforward in the case of a single *set bind*.

Translation of set comprehension involving multiple set binds is less obvious and some consideration is needed because alternative approaches exist. One possibility is to introduce a function for computing the cartesian product of multiple sets within the VDM-TS Library. Invocation of this function will yield a set of ordered n-tuples and permit the same line of thought as with a single set bind by applying a filter and map function, with the only difference that each tuple must be destructured such that every tuple element is bound to the correct identifier within the functions passed to filter and map.

Alternatively, translation of set comprehension with multiple set binds can be made by the code generator itself. The code generator can emit nested loop constructs that will iterate all set binds and add the evaluated expression to a resulting set if the predicate is `true`. The benefit of this approach is that the comprehended set can be constructed “in one go”, whereas the library approach constructs the entire cartesian product and afterwards reduces and manipulates it by filter and map functions. The library approach is however a viable option having an advantage in its simplicity and its smaller code generated file size. Nevertheless, the latter approach is chosen and summarised in translation rule 16.

Translation 16. Set Comprehension Expression

Single set bind: Let S be a *set comprehension expression* in VDM++ with a single set bind. Then S is code generated to a set of the single bind applied with the filter and map functions.

Multiple set binds: Let S be a *set comprehension expression* in VDM++ with multiple binds. Then S is code generated to an IFFE wrapping nested loop constructs iterating all multiple set binds and a condition with the predicate for adding the evaluated expression to the resulting set.

Listing 3.24 shows two examples of set comprehensions with a single bind and multiple binds (with a simple identifier pattern), respectively. The corresponding generated TypeScript code can be seen in Listing 3.25.

[4]This requires searching of infinite sets such as the natural numbers.

```
{ x * 2 | x in set {1, 2, 3, 4} & x mod 2 == 0 }
{ x | x in set {1,...,100}, y in set {13, 27} & x mod y = 0 }
```

Listing 3.24: VDM++ set comprehension.

```
new Set(1, 2, 3, 4).filter(x => x % 2 == 0).map(x => x * 2)
(() => {
    var result = new Set<number>();
    _.range(1, 100).forEach(x =>
        new Set(13, 27).forEach(y => {
            if (x % y === 0) { result.add(x); }
        }));
    return result;
})();
```

Listing 3.25: TypeScript set comprehension.

Notice, the latter example with multiple binds is wrapped in an IIFE just like let expressions in order to produce a value and avoid hoisting of the result variable. Also, notice that no while or for-loops is are used for iteration. Instead, the higher-order `forEach` function inherited from the `Array` prototype are used to enumerate the elements from the sets. The `forEach` function always returns void[5] and is in general used to execute side-effects as in these examples. This essentially combines an imperative and functional style of programming, which turns out to be very powerful as it eliminates the need to declare indexing and value variables.

3.6.4 Let-be-such-that Expression

In addition to simple let expressions, VDM++ provides *let-be-such-that expressions* that can be used to abstract away non-essential choices of an element from a set. These expressions consist of a multiple set bind of one or more patterns. Hence, the same approach as for set comprehensions described in subsection 3.6.3 is suited for the translation. However, a let-be-such-that expression is not well-formed in case that no applicable bindings can be found, thus the code generator has to emit a conditional check that throws a JavaScript run-time exception in order to preserve the semantics of a VDM++ interpreter. The translation is summarised in translation rule 17.

Translation 17. Let-be-such-that Expressions

Let L be a *let-be-such-that expression* in VDM++ with a multiple set bind. Then L is code generated to an IFFE wrapping nested loop constructs iterating the multiple set bind and a condition of the `be st` predicate.

[5]The void type in TypeScript can either be undefined or null. JavaScript functions without an explicit return value always returns undefined.

An example of a let-be-such-that expression and its translation is shown in listing 3.26 and listing 3.27, respectively. Notice that the `forEach` function is used for convenience. As there is no way to stop or break such function once a member satisfying the predicate is found, a for-loop might serve as a better translation, but this is only in terms of a slight increase in performance.

```
let x, y in set {1, 2, 3} be st x > y in x
```

Listing 3.26: A let-be-such-that expression in VDM++.

```
(() => {
    var result;
    new Set(1, 2, 3).forEach(x =>
        new Set(1, 2, 3).forEach(y => {
            if (x > y) { result = x; }
        }));
    if (!result) { throw 'Let be st found no applicable bindings'; };
    return result;
})();
```

Listing 3.27: The translated let-be-such-that expression in TypeScript.

3.6.5 Quantified Expressions

VDM++ provides three forms of quantified expressions:

- Universal: $(\forall x \in S)(P(x))$
- Existential: $(\exists x \in S)(P(x))$
- Unique existential: $(\exists! x \in S)(P(x))$

Where S is a VDM++ set and P is a predicate. The simplest expression is the *unique existential quantification* because it consists only of a single bind. This expression can be translated as a filtered set on a given predicate for which the cardinality is exactly one.

The *universal* and *existential* quantifiers may contain multiple set binds and give rise to the same considerations as described in subsection 3.6.3. Albeit similar in nature, translation of the universal and existential quantifiers can be done in a purely functional fashion. The `Set<T>` class inherits the higher-order function `every` which takes a callback function (predicate) as argument and returns true if the callback returns true for all elements from the set. Similarly, a `some` function exists that returns true if any callback on an element evaluates to true. By evaluating a sequence of these functions the semantics of multiple bound quantifiers can be reflected. The translation is described in translation rule 18.

Translation 18. Quantified Expressions

Unique existential: Let bd be a set bind and e be an boolean expression involving the pattern identifier of bd. Then the *Unique existential expression* is code generated to the set from bd applied with a filter of the boolean expression e for which the cardinality is exactly to one.

Existential: Let mbi's be multiple binds and e be an boolean expression involving the pattern identifier of mbi. Then the *Existential expression* is code generated to an IFFE wrapping nested loop constructs iterating all mbi and applying the `some` function with e to each mbi.

Universal: Let mbi's be multiple binds and e be an boolean expression involving the pattern identifier of mbi. Then the *Universal expression* is code generated to an IFFE wrapping nested loop constructs iterating all mbi and applying the `every` function with e to each mbi.

```
exists1 x in set {1, 2, 3} & x > 2
forall x, y in set {2, 4, 6}, z in set {7, 8, 9}
    & x mod 2 = 0 and y < z
```

Listing 3.28: Examples of quantified expressions in VDM++.

```
new Set(1, 2, 3).filter(x => x > 2).card() === 1
new Set(2, 4, 6).every(x =>
    new Set(2, 4, 6).every(y =>
        new Set(7, 8, 9).every(z =>
            (x % 2 % === 0) && (y < z) )))
```

Listing 3.29: Corresponding TypeScript translation of listing 3.28.

Chapter 4

Case Study

This chapter presents a case study used for evaluation of the VDM-TS Code Generator, and thereby the translations described in chapter 3. The case study involves a VDM++ model and its translated TypeScript version. The findings from this chapter is used in chapter 6 for either supporting or refuting the hypothesis.

4.1. Introduction

A case study is an analysis of an object that is studied by one or more methods in a natural setting [46]. Software testing comprise a set of methods that enable an investigation of source code. According to Dijkstra et al.: "*Program testing can be used to show the presence of bugs, but never to show their absence*" [47]. Since testing only shows the presence of errors it is useful to define principles that a testing process should adhere to, in order to catch as many errors as possible. In general, testing is first performed in a controlled environment on units of the software to ensure test isolation and to retain control of the execution. Then, the test environment is changed to a more natural setting allowing tests to be performed on the entire program.

Throughout the development of the VDM-TS Code Generator, different kinds of tests have been performed to ensure a reliable outcome. This chapter presents a case study of the VDM-TS Code Generator. The case study is used for evaluation and validation of the code generator, and thereby the translations proposed in chapter 3. The intent of the case study is to provide an increased level of certainty with regards to semantics preserving translations.

The remainder of this chapter is organised as follows. First, in section 4.2, the testing principles applied in the case study is presented. Next, the VDM++ Alarm model is presented in section 4.3, which is the chosen subject for translation. This is followed by an examination of the interesting constructs in the translation in section 4.4. Finally, section 4.5 describes the tests performed for evaluation of the model translation. The entire translation of the Alarm model can be found in appendix C.

4.2. Validation and Verification Principles Applied

Certain principles should be adhered to when examine and testing code in order to obtain a more reliable result. This section summarises the principles applied during this case study.

Manual and automatic testing: An effective testing process must include both manual and automatic test procedures. Manual tests are good at depth as they reflect on the domain and data structure. Automatic tests are good at breadth as they are executed and verified automatically by computer programs [48].

Unit testing: Unit testing is used for testing units separately, e.g. methods or objects, and should focus on the functionality of that unit. Unit testing can be combined with Test-Driven Development (TDD) for increased code coverage and regression testing [49, 50].

Test environment: A testing process should involve tests performed in different environments, including the working environment in which the software is expected to run.

Inspection: A verification process may include software reviews and inspections. Inspections do not involve the execution of the software, but relies on a review of the source code to find errors [51].

4.3. The Alarm Model

The VDM++ Alarm model presented in this section is borrowed from the book "Validated Designs for Object-oriented Systems" [6, pp.391-396] and can be found on the Overture website [8]. The model is based on an industrial plant for which it models the management of alarms. It is used to determine which expert should be called for a specific alarm and additionally ensure clarity of the expert duty roster. The model is expressed in VDM++ (VDM-10 language version). The model includes a single entry point: `new Test1().Run()`.

This particular model has been chosen for the case study, because it is compact and covers enough constructs in the VDM++ language to be considered a non-trivial model in this thesis. The Alarm model is comprised of four classes containing types, values, instance variables as well as operations and functions. Additionally, the model contains non-executable Design-by-Contract elements, but these are intentionally left out in the translation.

The following VDM++ constructs from the Alarm model are covered by this case study:

- Classes
  - Type definitions
  - Value definitions
  - Instance variable definitions
  - Function definitions
  - Operation definitions
    - Constructors
- Expressions
  - Enumerations (**`set`**, **`seq`**, **`map`**)
  - Set comprehension
  - Quantifiers (**`forall`**, **`exists`**)
  - Let-be-such-that
- Type operators (on **`set`**, **`map`**)
- Statements
  - Let statement
  - Block statement
  - Assignment
- Types
  - Basic types (**`bool`**, **`nat`**)
  - Quote types
  - Token type
  - Record types
  - Collection types (**`set`**, **`seq`**, **`map`**)
  - Union types
  - Object reference types

The following VDM++ constructs from the Alarm model are intentionally left out:

- Invariants
- Preconditions
- Postconditions

Two minor changes have been made to the Alarm model. The let-be-such-that statement in the `Plant` class is replaced with a let-bet-such-that expression. The tuple type in the `Test1` class is replaced with a record type. These changes have been made in order to cover more challenging constructs in the case study.

Structure of the model

The alarm model is comprised of four classes. Three of these constitute the actual model and one of them acts as entry point for testing. The structure of the model is shown in the UML class diagram in Figure 4.1. The functionality of each class is described below.

Expert: When an alarm is triggered, an expert with the required qualification is called to the scene. The expert is represented by the `Expert` class which contains a set of qualifications.

Alarm: The `Alarm` class represents an alarm with a description and a qualification needed to handle the alarm.

Plant: The `Plant` class represents the entire system logic. It holds the current schedule which is a mapping from periods to a set of experts. In addition, the `Plant` class maintains a set of triggered alarms. The `Plant` class is responsible for finding the number of experts on duty for a given period.

Test1: The `Test1` class contains both test data and testing logic.

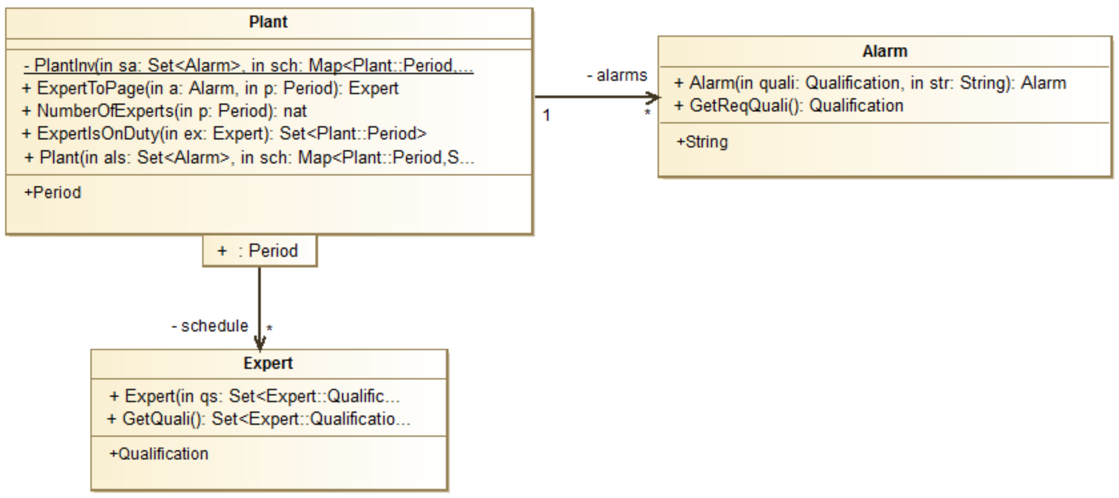

Plant
- PlantInv(in sa: Set<Alarm>, in sch: Map<Plant::Period,...
+ ExpertToPage(in a: Alarm, in p: Period): Expert
+ NumberOfExperts(in p: Period): nat
+ ExpertIsOnDuty(in ex: Expert): Set<Plant::Period>
+ Plant(in als: Set<Alarm>, in sch: Map<Plant::Period,S...
+Period
+ : Period
- alarms
1
*
Alarm
+ Alarm(in quali: Qualification, in str: String): Alarm
+ GetReqQuali(): Qualification
+String
- schedule
*
Expert
+ Expert(in qs: Set<Expert::Qualific...
+ GetQuali(): Set<Expert::Qualificatio...
+Qualification

Figure 4.1: Class diagram of the Alarm model.

4.4. Code generation to TypeScript

This section documents the output of the code generator, i.e. a translation of the VDM++ Alarm model to TypeScript. The Alarm model translation will be presented class-by-class following the structure described above. The entire model is translated and can be found in appendix C, but only the interesting parts are presented in this section.

4.4.1 The `Expert` class

Listing 4.1 shows the VDM++ `Expert` class which contains **`types`**, **`operations`** and **`instance variables`**. The translated `Expert` class, shown in listing 4.2, is represented as a TypeScript class with the translated types residing in a namespace associated with the class. Listing 4.3 shows the TypeScript quote `Elec` used by the `Qualification` as a part of its union type. Note that the TypeScript `quali` variable has a **`private`** access modifier which is the default behaviour in VDM++.

```
class Expert
    instance variables
        quali : set of Qualification;
    types
        public Qualification = <Mech> | <Chem> | <Bio> | <Elec>;
    operations
        public Expert: set of Qualification ==> Expert
        Expert(qs) == quali := qs;
        ...
end Expert
```

Listing 4.1: The `Expert` class.

```
class Expert {
    private quali: Set<Expert.Qualification> = undefined;
    constructor (qs: Set<Expert.Qualification>) {
        this.quali = qs;
    }
    ...
}
namespace Expert {
    export type Qualification =
        Quotes.Bio|Quotes.Chem|Quotes.Elec|Quotes.Mech;
}
```

Listing 4.2: The translated `Expert` class.

```
namespace Quotes {
    export class Elec extends Quote {
        private static instance = new Elec();
        constructor() {
            super();
            if (Elec.instance) {
                throw Error("Use getInstance()");
        }
        Elec.instance = this;
    }
    public static getInstance() {
        return this.instance;
    }
    ...
```

Listing 4.3: A generated quote class implementing the singleton pattern.

4.4.2 The `Alarm` class

Listing 4.4 shows two definitions in the `Alarm` class; the named type `String` and the operation `GetReqQuali`. The translated `GetReqQuali` operation has a return type `Expert.Qualification` which is a fully qualified named to the type in `Expert` class.

```
...
types
    public String = seq of char;

operations
    pure public GetReqQuali: () ==> Expert`Qualification
    GetReqQuali() ==
        return reqQuali;
...
```

Listing 4.4: Snippets from the VDM++ `Alarm` class.

```
...
    public GetReqQuali() : Expert.Qualification {
        return this.reqQuali
    }
}
namespace Alarm {
    export type String = string;
}
```

Listing 4.5: Snippets from the translated `Alarm` class.

4.4.3 The `Plant` class

This subsection presents the `Plant` class which is responsible for the logic of the Alarm model. The class is split into smaller parts in order to present the code generation in a clear manner.

4.4.3.1 Types and instance variables

The translation of both types- and instance variable definitions are shown below in listing 4.6 and listing 4.7. The `Period` type is represented in TypeScript by the `Token` class contained within the VDM-TS Library.

```
class Plant
instance variables
    schedule : map Period to set of Expert;
    ...
types
    public Period = token;
```

Listing 4.6: Types and instance variables of the VDM++ `Plant` class.

```
class Plant {
    private schedule: Map<Plant.Period, Set<Expert>> = undefined;
    ...
}
namespace Plant {
    export type Period = Token;
}
```

Listing 4.7: Types and instance variables of the translated `Plant` class.

4.4.3.2 Functions

The `Plant` class has a single function as shown in listing 4.8 and listing 4.9. The `PlantInv` function body contains multiple *quantified expressions* including both the *existential* and *universal* quantifiers. The function returns a conjunction of two quantified expressions.

```
1  ...
2  functions
3      PlantInv: set of Alarm * map Period to set of Expert -> bool
4      PlantInv(sa,sch) ==
5      (forall p in set dom sch & sch(p) <> {}) and
6      (forall a in set sa &
7          forall p in set dom sch &
8              exists expert in set sch(p) &
9                  a.GetReqQuali() in set expert.GetQuali());
10 ...
```

Listing 4.8: Function from the VDM++ `Plant` class.

It is worth mentioning that the two listings 4.8 and 4.9 are aligned from line 5 to 9, such that each line in the listings correspond to one another.

```
1  ...
2  static PlantInv(sa: Set<Alarm>,
3                  sch: Map<Plant.Period,Set<Expert>>) : boolean {
4      return
5          sch.dom().every(p => !_.equals(sch.apply(p), new Set())) &&
6          sa.every(a =>
7              sch.dom().every(p =>
8                  sch.apply(p).some(expert =>
9                      expert.GetQuali().inset(a.GetReqQuali()))));
10 }
11 ...
```

Listing 4.9: Function from the translated `Plant` class.

4.4.3.3 Operations

The `Plant` class contains four operations. The `NumberOfExperts` and `Plant` operations are omitted. The translations of the two more interesting operations `ExpertToPage` and `ExpertIsOnDuty` are presented below.

The `ExpertToPage` operation is shown in listing 4.10 and listing 4.11. The operation includes a *let-be-such-that expression* used to select a non-essential expert for which the *predicate* evaluates to **`true`**. The translation rule of the *let-be-such-that expression* can be found in subsection 3.6.4. Recall that the code generator has to construct a check for an applicable binding and throw a run-time exception if this is not the case (see line 11 in listing 4.11).

```
...
    public ExpertToPage: Alarm * Period ==> Expert
    ExpertToPage(a, p) ==
        return let expert in set schedule(p) be st
            a.GetReqQuali() in set expert.GetQuali()
        in
            expert
...
```

Listing 4.10: Operations from the VDM++ `Plant` class.

```
 1  ...
 2      public ExpertToPage(a: Alarm, p: Plant.Period) : Expert {
 3          return (() => {
 4              var result;
 5              this.schedule.apply(p).forEach(expert => {
 6                  if(expert.GetQuali().inset(a.GetReqQuali())) {
 7                      res = expert;
 8                  }
 9              });
10              if (!result) {
11                  throw 'Let be st found no applicable bindings';
12              };
13              return result;
14          })();
15      }
16  ...
```

Listing 4.11: Operations from the translated `Plant` class.

The `ExpertIsOnDuty` operation encompass a *set comprehension expression* shown in listing 4.12. The code generated TypeScript method is shown in listing 4.13.

```
...
    public ExpertIsOnDuty: Expert ==> set of Period
    ExpertIsOnDuty(ex) ==
        return {p | p in set dom schedule & ex in set schedule(p)};
...
```

Listing 4.12: Operations from the VDM++ Plant class.

```
...
    public ExpertIsOnDuty(ex: Expert) : Set<Plant.Period> {
        return (() => {
            var res = new Set<Plant.Period>();
            this.schedule.dom().forEach(p => {
                if(this.schedule.apply(p).inset(ex)) {
                    res.add(p);
                }
            });
            return res;
        })();
    }
}
```

Listing 4.13: Operations from the translated Plant class.

4.4.4 The Test1 class

This subsection presents the final class of the Alarm model which is responsible for testing the actual model. Similar to the Plant class, this class is presented in smaller parts.

4.4.4.1 Types and values

Listing 4.14 and listing 4.15 shows the VDM++ types and values of the Test1 class and their code generated TypeScript counterparts. The record type PE is translated to a fixed class implementation in TypeScript as described in subsection 3.4.2. Notice that both the record type and the Token value p1 are residing in the namespace associated with the Test1 class.

```
...
types
    public PE :: periods : set of Plant`Period
                 expert  : Expert
values
    p1: Plant`Period = mk_token("Monday day");
...
```

Listing 4.14: Types and values from the translated Test1 class.

```
...
namespace Test1 {
    export class PE extends Record {
        constructor(public periods: Set<Token>, public expert: Expert){
            super();
        }
    }
    export const p1: Plant.Period = new Token("Monday day");
    ...
}
```

Listing 4.15: Types and values from the translated `Test1` class.

4.4.4.2 Instance variables

The instance variables of `Test1` class, shown in listing 4.16 and the translated TypeScript shown in listing 4.17, lists three distinct instance variables. Each are of the *object reference type* as they hold a value of a new class instance. Quote types in TypeScript are represented as classes implementing the singleton pattern. Consequently, the quote values must be accessed through its `getInstance` method as shown for both the `a1` and `ex1` instance variable in Listing 4.17.

```
class Test1
instance variables
    a1    : Alarm   := new Alarm(<Mech>,"Mechanical fault");
    ...
    ex1   : Expert  := new Expert({<Mech>,<Bio>});
    ...
    plant: Plant    := new Plant({a1},{p1 |-> {ex1,ex4},
                                        p2 |-> {ex2,ex3}});
```

Listing 4.16: Instance variables from the translated `Test1` class.

```
class Test1 {
    private a1: Alarm =
        new Alarm(Quotes.Mech.getInstance(), "Mechanical fault");
    ...
    private ex1: Expert =
        new Expert(new Set<Quotes.Bio|Quotes.Mech>(
            Quotes.Mech.getInstance(),Quotes.Bio.getInstance()));
    ...
    private plant: Plant =
        new Plant(new Set(this.a1), new Map(
            new Maplet(Test1.p1, new Set(this.ex1,this.ex4)),
            new Maplet(Test1.p2, new Set(this.ex2,this.ex3))));
...
```

Listing 4.17: Instance variables from the translated `Test1` class.

4.4.4.3 Operations

The `Run` operation is shown in listing 4.18 and listing 4.19. It consists of a *let statement* which introduces local bindings that hold the result of operation invocations on the plant object reference and returns a value of a record type.

```
...
operations
    public Run: () ==> PE
    Run() ==
        let periods = plant.ExpertIsOnDuty(ex1),
            expert  = plant.ExpertToPage(a1,p1)
        in
            return mk_PE(periods,expert);
end Test1
```

Listing 4.18: Operation from the translated `Test1` class.

```
...
    public Run() : Test1.PE {
        const periods : Set<Token> =
            this.plant.ExpertIsOnDuty(this.ex1);
        const expert : Expert =
            this.plant.ExpertToPage(this.a1, Test1.p1);
        return new Test1.PE(periods, expert);
    }
}
```

Listing 4.19: Operation from the translated `Test1` class.

4.5. Test and Evaluation

Several tests have been performed in order to evaluate and validate the correctness of the code generated TypeScript. This section presents the tests which adhere to the testing principles described in section 4.2. These tests are not only concerned with the case study but also the development of the VDM-TS Code Generator.

4.5.1 Inspection

Section 4.4 above enables inspection and comparison of the VDM++ model and the code generated TypeScript. A *code review* is one example of an inspection technique, which is used in this thesis project. This encompass reviewing selected code samples in order to find errors and inconsistencies in the code [52]. Code reviews are normally performed on developed source code rather than on generated code, but in order to ensure correctness, the static process of inspecting the generated output of the Alarm model has been conducted in this case study. In addition, the translated constructs from chapter 3 and 5 has been inspected during development of the VDM-TS Code Generator through small sample models.

The utilisation of the code review technique has resulted in a number of corrections to the translations during development and thereby reducing the presence of errors in the code generation of the Alarm model. However, inspections cannot be used for validation of correctness alone and therefore additional tests should be carried out.

4.5.2 Automated testing

During the development of the code generator, automated testing has been applied to ensure the validity of the developed functionality. The VDM-TS Library, described in section 3.4.1, provide essential, not built-in functionality to the code generated TypeScript. It is important that this functionality comply with the semantics of VDM++ defined in the language manual [28]. The VDM-TS Library has been subject for unit testing using the Jasmine testing framework [53], where the functionality of each component is automatically executed and compared to the expected result.

The unit tests have been written using the *Arrange-Act-Assert* unit testing pattern [54] for a clear separation of what is setup, what is object under test and what is the expected result. One of the 49 test cases for the `Map` and `Set` operators is shown below in Listing 4.20.

```
it("Should union sets of different types", function() {
    // Arrange
    var set1 = new Set<number>(1, 2, 3);
    var set2 = new Set<boolean>(true, false);
    var result = new Set<boolean|number>(1, 2, 3, true, false);
    // Act
    var unionSet = set1.union(set2);
    // Assert
    expect(unionSet.equals(result)).toBe(true);
});
```

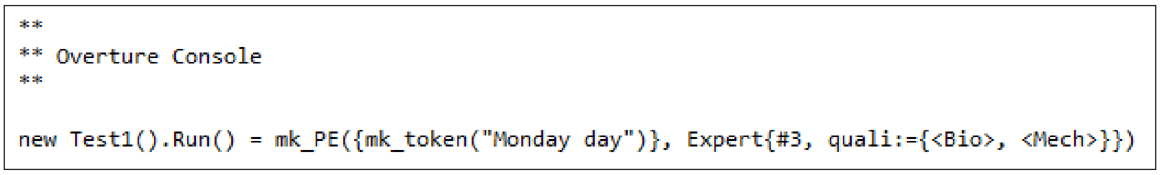
Listing 4.20: Unit test of the `Set` union operation.

4.5.3 Execution test

An execution test is performed as a manual test on the VDM-TS Code Generator using the Alarm model. The generated TypeScript code is compiled and executed and the actual result is compared to the expected result from the execution of the VDM++ Alarm model using the Overture interpreter. Throughout the thesis project, execution tests have additionally been performed on smaller VDM++ samples[1] in relation to each translated construct. This improves the reliability of the construct translations and thereby increases the level of certainty of correctness for the VDM-TS Code Generator as a whole.

[1]Some of these samples are shown for each translated construct in chapter 3.

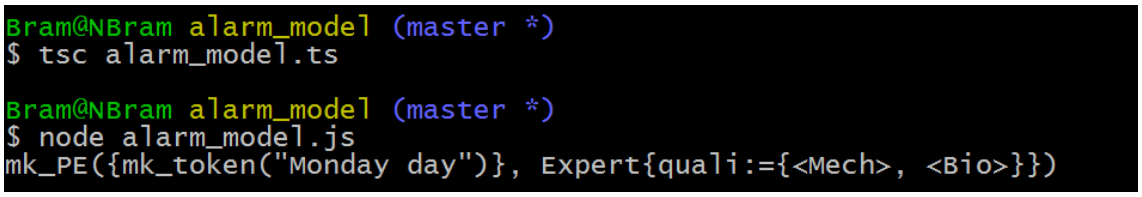
The result of the execution test performed on the Alarm model is shown in the figures below. Before the TypeScript source code is executed, it is compiled to JavaScript using the TypeScript compiler. Figure 4.2 shows the VDM++ output from the Overture interpreter [8]. Figure 4.3 shows the execution of the compiled JavaScript source code in the Node.js runtime environment version 5.5.0 [35]. Additionally, fig. 4.4 shows the JavaScript object from the compiled TypeScript run in the Chrome browser version 50 (V8 JavaScript engine[2]).



```
**
** Overture Console
**

new Test1().Run() = mk_PE({mk_token("Monday day")}, Expert{#3, quali:={<Bio>, <Mech>}})
```

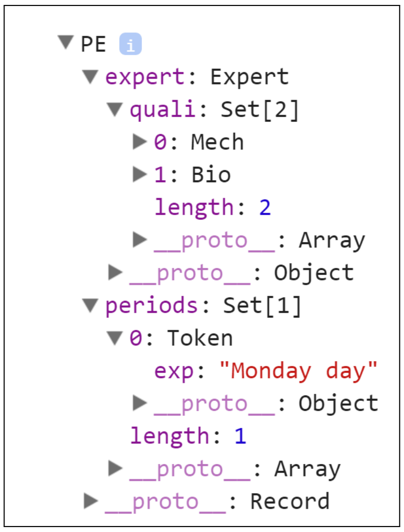
Figure 4.2: Execution of the Alarm model in the Overture tool.



```
Bram@NBram alarm_model (master *)
$ tsc alarm_model.ts

Bram@NBram alarm_model (master *)
$ node alarm_model.js
mk_PE({mk_token("Monday day")}, Expert{quali:={<Mech>, <Bio>}})
```

Figure 4.3: Execution of the code generated Alarm model in Node.js.



```
▼PE i
  ▼expert: Expert
    ▼quali: Set[2]
      ▶0: Mech
      ▶1: Bio
       length: 2
      ▶__proto__: Array
    ▶__proto__: Object
  ▼periods: Set[1]
    ▼0: Token
       exp: "Monday day"
      ▶__proto__: Object
     length: 1
    ▶__proto__: Array
  ▶__proto__: Record
```

Figure 4.4: The JavaScript object from the execution of the code generated Alarm model in the Chrome browser (V8 JavaScript engine).

The result of the execution test shows that the code generator produces a valid TypeScript output and that this output, when executed, performs in accordance with the VDM++

[2]See https://developers.google.com/v8/, accessed 2nd of May 2016.

Alarm model. This confirms that the generated code complies with the semantics of VDM++ for this example.

Chapter 5

Challenging Constructs

This chapter is concerned with translations of selected VDM++ constructs that are particularly challenging to represent in TypeScript due to language differences. The translations build on the background introduced in chapter 2 and further extends the set of translation rules derived in chapter 3. This chapter is preceded by chapter 6 in which the findings from the thesis project is synthesised and evaluated. The findings from this chapter contribute to a general discussion on TypeScript as target language in chapter 6.

5.1. Introduction

Difficulties in translation from one language to another stem from the fact that languages can be different in spirit, semantics and syntax. In programming languages, the two major ways data is structured can be said to derive from branches of mathematics and from taxonomy [55]. The mathematical way organises data as cartesian products, disjoint sums and function spaces. The taxonomically way organises data in hierarchies where data anywhere in a hierarchy can inherit attributes from data at a higher level.

In VDM++, data is structured both in a mathematical and taxonomical way. The core structure of VDM++ is based on a mathematical foundation with constructs such as records, unions and functions. A distinguishing feature of VDM++ is the extension of classes, which enables modelling of object-oriented systems and allows data to be taxonomically organised through multiple inheritance. Many modern programming languages, including TypeScript, only supports single inheritance. So, reflecting the semantics of multiple inheritance from VDM++ in TypeScript is a challenging task.

The remainder of this chapter treats VDM++ constructs that prove challenging for translation to TypeScript. In section 5.2, translation of class-based inheritance in VDM++ is considered with special attention given to multiple inheritance. Since TypeScript has type erasure, translation of VDM++ constructs that require run-time type information is particularly challenging. Thus, *class membership expressions* and the more general *is expressions* are addressed in section 5.3 and section 5.4, respectively.

It should be noted that the translations are presented in the same way as in chapter 3, i.e. first general considerations are presented, followed by a translation rule, examples and eventual limitations. Additionally, it should be noted that some translation rules in this chapter are proposals only[1].

5.2. Inheritance

Inheritance in object-oriented programming is either class-based or prototype-based. More precisely, inheritance can be defined as: *"Inheritance in OOP may be based either on the concepts of subclass (in those OOP languages with classes) or on default delegation of responsibility (in those OOP languages based on prototypes)"* [56]. At first, this distinction might seem to have little importance from a translation point of view. After all, both VDM++ and TypeScript are class-based languages. However, while it is possible to translate single inheritance by completely abstracting away how inheritance is obtained in JavaScript, it is not the case for multiple inheritance. As mentioned in chapter 2, the way prototypes are used in JavaScript constitute single inheritance. For this reason, multiple inheritance in TypeScript cannot possibly be a native language construct. Nonetheless, with knowledge of how prototypal inheritance works in JavaScript, it is possible, at least to some extent, to mimic the semantics of multiple inheritance in VDM++. The following subsections describe translation of inheritance in more detail.

5.2.1 Single Inheritance

Inheritance in VDM++ and TypeScript is conceptually identical because both languages support class-based inheritance. The native support for single inheritance in TypeScript entails that translation of this functionality in principle is trivial. However, some differences in the languages impose limitations to the translation. These limitations are described immediately after the translation rule and the example below.

Translation 19. Single Inheritance

Let S be a subclass in VDM++ that inherits from base class B. Then S is translated to a TypeScript class with a corresponding name that denote B in its **extends** clause.

```
class A
    values
        public foo = 2;
    operations
        public bar : () ==> int
```

[1]The proposed translations of *multiple inheritance* and *is expressions* for compound data types are not implemented in the VDM-TS Code Generator.

```
        bar() == return 4;
end A

class B is subclass of A
    operations
        public baz : () ==> int
        baz() == return foo + bar();
end B
```

Listing 5.1: Base class and subclass in VDM++.

```
// base class A
class A {
    public bar(): number { return 4; }
}
namespace A { export const foo: number = 2; }

// subclass B inherits foo and bar from A
class B extends A {
    public baz(): number { return B.foo + this.bar(); }
}
new B().baz()   // 6
```

Listing 5.2: Base class and subclass in TypeScript.

Limitation. Overwriting private class members

Private class members cannot be overridden in TypeScript. This implies that only overridden class members in VDM++ declared as **`public`** or **`protected`** can be supported by the code generator.

Limitation. Name qualification

In VDM++, *name qualification* (**`** -sign) can be used to refer to any class member in a hierarchy of classes. There is no similar functionality in TypeScript. The **`super`** keyword can be used to access only a direct superclass, and this is limited to accessibility of methods, not instance variables. Since value and type definitions in VDM++ are reflected in TypeScript as members of a namespace, they can be accessed in a similar fashion to name qualification by using TypeScript dot notation.

5.2.2 Multiple Inheritance

Multiple inheritance in VDM++ allows classes to have more than one superclass. Despite TypeScript has no support for multiple inheritance, it is interesting to investigate whether the VDM-TS Code Generator can do better than just throw unsupported exceptions. One naive approach is to transform a class hierarchy with multiple inheritance into a single multi-level inheritance hierarchy. For instance, if `C` extends both `A` and `B`, then `C` might become a subclass of `B` and `B` become a subclass of `A`. This entails that `C` can inherit implementations from both `A` and `B`. However, this approach breaks the subtype relationship established by multiple inheritance; while `C` will correctly be subclass of both `A` and `B`, `B` has now become a subclass of `A`. This introduced relationship clearly conflicts with the semantics of multiple inheritance in VDM++. Moreover, it is unclear whether `A` shall be subclass of `B` or vice visa.

The VDM++ to Java Code Generator in VDMTools [23] handles multiple inheritance by substitution of interfaces. Like TypeScript, Java only supports single inheritance. The approach is that a derived class extends one class, while implementing the rest as interfaces. This is not a perfect resemblance of multiple inheritance because no actual implementation is inherited by implementing an interface. However, Java interfaces establish a is-a relationship of class types similar to inheritance[2]. Therefore, this approach reflects the subtyping semantics of multiple inheritance in VDM++, as opposed to the naive approach described above. TypeScript include a notion of interfaces, but these are distinct from Java interfaces in that they are purely a compile-time construct, hence having no run-time representation. Consequently, no **`instanceof`** tests involving interfaces can be made, which poses limitations to *class membership expressions* (see section 5.3).

In Common Lisp Object System (CLOS) the concept of *mixins* can be used as an alternative to achieve some form of multiple inheritance [57]. According to Flatt et al. [58], "*A mixin function maps a class to an extended class by adding or overriding fields and methods.*". This essentially means that a class can be extended to gain access to methods from another class. How accessibility of these methods is obtained is very language dependent (i.e. how the mixin function is implemented). This concept is also called *mixin-based inheritance*. In TypeScript, mixins are not a native language construct. Similar in CLOS, mixin-based inheritance is merely a programming style. It turns out that TypeScript is powerful enough to embrace this style of programming, because mixins can be mimicked by copying functions from one object to another. To resemble multiple inheritance by mixins, the idea is that a class inherits from one plain class and multiple *mixin classes*. This approach shares characteristics with the replacement of Java interfaces, however the distinct feature of mixins is that they allow actual implementations to be inherited from multiple superclasses.

In order to model mixins in TypeScript, it is required to have insights into how prototypal inheritance works in JavaScript. As described in chapter 2, objects in JavaScript that are created by a *constructor function*, inherit properties from the prototype of this function. In TypeScript, this corresponds to a class having an associated prototype, from which instances of the class can inherit properties (e.g. methods). Hence, instances of a class share the same properties defined on the class prototype. Since JavaScript is a dynamically

[2]http://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php, accessed 15th of May 2016.

typed language, properties of the prototype can be added and deleted at run-time. Thus, properties of one class can be included in another class by copying properties from the prototypes. Listing 5.3 shows the TypeScript mixin function that does the heavy lifting. It copies prototype properties from an arbitrary number of mixin classes (superclasses) into the prototype of a derived class (subclass). Notice how this imposes a linearisation scheme, i.e. name conflicts, for example caused by the Diamond Problem [59], are resolved by overriding already defined properties. Thus, precedence is determined by the order in which the mixin classes are applied.

```
function applyMixins(derived: any, bases: any[]) {
    bases.forEach(base => {
        Object.getOwnPropertyNames(base.prototype).forEach(name => {
            derived.prototype[name] = base.prototype[name];
        });
    });
}
```

Listing 5.3: Mixin function in TypeScript.

The combination of traditional and mixin-based inheritance can be used to provide support for multiple inheritance in a limited form in TypeScript. One limitation is that mixins do not establish a is-a relationship as multiple inheritance in VDM++. In addition, there is a semantically difference in how VDM++ and the outlined mixin-based strategy handles name ambiguities. The idea of prototypes is that common functionality can be shared between instances, and this encompass typically only methods. From a VDM++ level, this means that all superclasses may define functions and operations implementations, but only one superclass may define instance variables, types and values, in order to be subject for code generation. Despite the presence of these limitations, mixin-based inheritance is the best possible resemblance of VDM++ multiple inheritance in TypeScript. This leads to the translation rule 20 below.

Translation 20. Multiple Inheritance

Let D be a subclass of the base classes $B_1, \ldots, B_n$.
Then D is translated to a TypeScript class with a corresponding name that extends B_1 and implements $B_2, \ldots, B_n$. The `applyMixins` function from the VDM-TS Library is invoked immediately after D has been defined.

For the example below, suppose `A` is a class defining an operation named `square`, and `B` is an class defining an operation named `cube`. The operations take an **`int`** as argument and returns its squared and cubed value, respectively.

```
class C is subclass of A, B
    operations
        public sum : () ==> int
        sum() == return square(4) + cube(2);
end C
```

Listing 5.4: Example of multiple inheritance in VDM++.

```
class A {
    public square(x: number): number { return x*x; }
}

class B {
    public cube(x: number): number { return x*x*x; }
}

class C extends A implements B {
    public sum(): number  {
        return this.square(4) + this.cube(2);
    }
    public cube: (x: number) => number;
}
_.applyMixins(C, [B])

new C().sum(); // 24
```

Listing 5.5: Mixed-based inheritance in TypeScript.

Limitation. Cascading single inheritance limitations

Limitations imposed by single inheritance also applies to multiple inheritance.

Limitation. Superclass definitions

Multiple superclasses may define actual implementations for operations and functions, but only one superclass may define instance variables, types, and values.

5.3. Class Membership Expressions

Once hierarchies are established with inheritance, it can be useful to test the relationship between *object references* and classes. VDM++ includes four *class membership* expressions for this purpose. While the **instanceof** operator in TypeScript nicely reflects the semantics of the *isofclass* expression in VDM++, there are no similar expressions in TypeScript to the *sameclass*, *isofbaseclass* and *samebaseclass* expressions. Since these expressions rely on run-time type information, their translation is limited to reflection capabilities in JavaScript.

As described in chapter 2, class-based object-orientation is simulated in JavaScript by *constructor functions* and *prototypes*. A key observation is that instances created by a *specific constructor* function inherit properties from the same prototype. Thus, the semantics of the *sameclass* expression can be obtained by testing equality between prototypes from which two instances inherit. To see this, it might be beneficial to consult figure 2.4 in chapter 2. From the figure it can also be observed, that the prototype of a *constructor function* acting as a base class (function `A`), inherits from `Object.prototype`. Therefore, by testing equality between these prototypes and testing whether the *object reference* in question is an instance of the constructor function acting as a base class, the semantics of *isofbaseclass* expressions can be obtained. Reflecting the semantics of the *samebassclass* expression is more elaborate, but the basic idea is to find the constructor function that acts as a base class of one of the *object references* applied, and afterwards test whether this constructor function also acts as a base class for the other object reference.

Translation 21. Membership Expressions

The four class membership expressions, *isofclass*, *sameclass*, *isofbaseclass* and *samebaseclass*, become TypeScript functions with corresponding names residing in the VDM-TS Library.

Suppose `B` and `C` are both subclasses of `A`. Let `a` be an instance of `A`, `b` an instance of `B` and `c` an instance of `C`. Then:

```
isofclass(A, b)  -- true
sameclass(b, b)  -- true
isofbaseclass(A, a) -- true
samebaseclass(b, c) -- true
```

Listing 5.6: Class membership expressions in VDM++.

```
b instanceof A // true
Object.getPrototypeOf(b) === Object.getPrototypeOf(b)    // true
Object.getPrototypeOf(A.prototype) === Object.prototype
    && a instanceof A        // true
_.samebaseclass(b, c)    // true
```

Listing 5.7: Semantics of class membership expressions in TypeScript.

The functionality for reflecting the *class membership expressions* is implemented as functions in the VDM-TS Library. However, the actual implementations is shown for the first three examples in Listing 5.7.

Limitation. Class hierarchies with multiple inheritance

The semantics of *class membership expressions* cannot be reflected in TypeScript for class hierarchies with multiple inheritance. This is because multiple inheritance is replaced with mixins during code generation and mixins do not establish subtype relationship.

5.4. Is Expressions

Beside class membership expressions, VDM++ includes additional reflection capabilities. *Is expressions* in VDM++ are used to test whether an expression is of a specific type. Is expressions are particularly interesting due to their usage in *user-defined type guards* as described in subsection 3.4.5, but also proves challenging for translation due to type erasure in TypeScript. Hence, translation must rely entirely on built-in or added reflection capabilities in JavaScript, which in ES5 are not the best.

For the VDM++ types represented as primitive data types in TypeScript (string, number, boolean), the **`typeof`** operator will suffice for translation. Furthermore, the **`instanceof`** operator will work out for some of the VDM++ types represented as TypeScript classes. However, type inspection of many of the compound data types are non-trivial because of their nested nature. For instance, the set collection is in the TypeScript translation represented as a generic class `Set<T>`. While it is possible to test whether an expression is instance of `Set`, there is no way to determine the type parameter `T` at run-time because generics are erased at compile time[3].

One approach to cope with this issue is to encode type information into objects as metadata. For example, the type of the elements from a set can be encoded as a string and assigned to a special property of the object representing a set expression. The benefit of this approach is that a function within the VDM-TS Library can be implemented to reflect the semantics of *is expressions*. The main drawback is that an encoding strategy has to be decided for nested data types such as **`set of set of int`**, which has to be parsed by the function residing in the library.

Another approach is to let the VDM-TS Code Generator generate constructs in the translation that constitute the effect of *is expressions*. Continuing the set example from above,

[3]Generics are even in some strongly-typed languages merely compile-time syntactic sugar, e.g. Java. See `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`, accessed 25th of April 2016.

the code generator can emit code that checks if an object is instance of `Set` and whether all elements from the set is of the same type. Since set types can be arbitrary complex, the *is expression* can within the code generator be constructed recursively. This idea can also be transferred to support *is expressions* of other compound data types. The liability of this approach is that it will generate code that can be hard to read and adds overhead to the translation. It is however the easiest way to reflect the semantics of *is expressions* because no additional metadata on types is required. This leads to translation rule 22.

Translation 22. Is expressions

Basic type: Let i be either a boolean, numeric, character or token *basic is expression* with a value e. The i is code generated to an expression involving **`typeof`** or **`instanceof`** of e depending on the type of the is expression.

General: Let i be a *general is expression* with a value e and a type t. Then i is code generated to an expression of nested IFFEs constructed recursively from e for checking every sub-types of t.

Examples of VDM++ *is expressions* and their corresponding translation are shown in listing 5.8 and listing 5.9. The translation of the example on line 5 in listing 5.8 might look intimidating, however it does exactly for set types as outlined above in a recursive fashion and are additionally wrapped in IIFE's in order to transform a sequence of statements into an expression.

```
1 is_int(3)
2 is_bool(bool)
3 is_token(mk_token(42))
4 is_(17, int|token)
5 is_({{1, 2, 3}, {4, 5, 6}}, set of set of int)
```

Listing 5.8: Examples of VDM++ is expressions.

```
typeof 3 === 'number'
typeof true === 'boolean'
new Token(false) instanceof Token
(u => typeof u === 'number' || u instanceof Token)(17)

// is set of set of number exp
(s => {
    if (s instanceof Set) {
        return s.every(e =>
            // is set of number exp
            (s => {
                if (s instanceof Set) {
                    return s.every(e =>
                        // is number exp
```

```
                                typeof e === 'number'
                            );
                        }
                        return false;
                    })(e)
                );
        }
        return false;
})(new Set(new Set(1, 2, 3), new Set(4, 5, 6)))
```

Listing 5.9: Corresponding TypeScript translation for listing 5.8.

Chapter 6

Concluding Remarks

This chapter concludes the thesis and it is therefore based on all the preceding chapters. The main findings are presented, discussed and put into perspective. The hypothesis and goals defined in chapter 1 are revisited and evaluated. Additionally, the entire thesis project is reflected upon and potential directions for future work are presented.

6.1. Introduction

Once a project comes to an end, it is important to evaluate on the achievements in order to provide answers to the original questions. Discussing the various issues encountered during the project can help reaching a final judgment. Additionally, reflection on the work carried can be used to extract knowledge from the experiences. Thus, one can benefit from lessons learned and avoid making similar mistakes in future projects.

This chapter is concerned with this kind of reflection in the context of this thesis project, and it is organised as follows. First, a general discussion of selected topics on code generation from VDM++ to TypeScript is presented in section 6.2. Next, reflection on the work carried out during this thesis project is presented in section 6.3. Then, the hypothesis is evaluated in section 6.4, followed by an evaluation of the thesis goals in section 6.5. Afterwards, in section 6.6, possible directions of future work are proposed and lastly, final remarks are presented in section 6.7.

6.2. Discussion

At first sight, VDM++ and TypeScript share many similarities. Both languages offer a combination of class-based object-orientation and functional programming, and include many of the traditional language constructs from these paradigms, such as conditionals, while loops, mutable variables, assignment statements, lambda abstractions and higher-order functions. However, during this thesis project it has become apparent that there are fundamental differences between the languages. Below a general discussion of Type-

Script as target language is presented along with an important design consideration in the VDM-TS Code Generator is presented. Additionally, the capabilities of the code generator are compared with two existing VDM++ to Java code generators.

6.2.1 TypeScript Code Generation

Code generation from VDM++ to TypeScript is not a straightforward process. This is primarily caused by the type system of TypeScript and its distinctive feature of type erasure. As described in chapter 2, the static type information introduced by TypeScript is completely erased when it is compiled to JavaScript. This complicates the process of mapping from VDM++ to TypeScript constructs, because special considerations have to be given in order to determine how TypeScript type information can be preserved in JavaScript. For example, VDM++ as well as TypeScript offer a tuple type, however, JavaScript does not have the notion of such a type. Once the TypeScript tuple type is compiled to JavaScript, it is represented as an `Array`. For this reason, the semantics of *is expressions* in VDM++ are non-trivial to reflect in TypeScript, as described in section 5.4. The decoupling of compile and run-time types is problematic, because run-time casts cannot be supported by TypeScript. Since JavaScript is a dynamically typed language, no type checks are made of a function's return type. This entails that VDM++ conversion errors that happen at run-time are particularly difficult to reflect, and numerous explicit checks have to be code generated. This is especially relevant for narrowing of union types, as described in subsection 3.4.5.

The paragraph above describes the main elements encountered during this thesis project that impede translation from VDM++ to TypeScript, and additionally serves as a premise for a relevant question. It has become apparent that code generation from VDM++ to TypeScript cannot be attained without knowledge of JavaScript. Thus, this raises the question whether the extra layer of compilation that TypeScript impose is actually worthwhile or code generation directly to JavaScript is a more viable option? The most notable features of TypeScript are classes, modules and obviously static typing. However, both classes and modules are included in the upcoming[1] ES2015 version of JavaScript which in principle obviates the need for TypeScript with respect to these constructs. Despite the fact that JavaScript in the future will converge towards a TypeScript looking syntax, there are, at the time of writing, no signs of static typing entering future ECMAScript specifications [61]. Hence, TypeScript is still a better choice than JavaScript for reflecting the typing of VDM++.

6.2.2 Code Generator or Library Calls

One of the most frequent design considerations to be taken during this thesis project is whether functionality shall be put in a library or directly emitted by the code generator. Both approaches are viable options for reflecting the semantics of VDM++, but a library might be favourable as it encourages code reusability and can provide similar syntax to VDM++. One of the lessons learned is that design decisions with respect to library calls

[1]The ES2015 specification has been released but no JavaScript engine has completed the implementation at the time of writing [60].

should be consistent. For example, in the VDM-TS Code Generator, the semantics of **`dunion`** is obtained by IR transformations, and the result is emitted code that continuously apply the **`union`** operator. Hence, the operand to **`dunion`** (a set of set values) must be available as IR nodes in the code generator. However, since the **`power`** operator is represented as a library call, the resulting powerset is first known at runtime. Therefore, distributed union of a powerset is not possible in the current VDM-TS Code Generator, albeit the solution is simple, for instance by representing both the **`power`** and **`dunion`** operators as library calls.

6.2.3 Comparing TypeScript with Java Code Generation

TypeScript can be seen as an attempt to make JavaScript look familiar to developers coming from traditional class-based languages, such as Java. However, TypeScript's underlying programming model is still JavaScript, and it is clear that there is a world of difference between JavaScript and Java; dynamic typing, prototype-based inheritance and higher-order functions are unknown concepts in Java. Nevertheless, it is interesting to see how TypeScript and Java compare from a VDM++ code generation perspective. The capabilities of the Overture Java Code Generator [9] and the VDMTools VDM++ to Java Code Generator [19] will be used for comparison.

Since TypeScript embraces class-based object-orientation, many VDM++ types are represented as classes (e.g. records, quotes, tokens) in the VDM-TS Code Generator, similar to the two Java code generators. The VDM-TS Library is inspired by the Overture Java Code Generator run-time library and includes the VDM++ notion of equality and clone functionality for value semantics. VDM++ collection types (**`set`**, **`map`**, **`seq`**) are additionally represented as classes in all the code generators. However, these types are in both Java code generators built on library classes from the `java.util` package (`HashSet`, `HashMap`, `ArrayList`), whereas they all are built on `Array` in the VDM-TS Code Generator. The data structure possibilities in Java are more comprehensive and optimised for performance.

The functional nature of TypeScript is where the language shines compared to Java. Functional constructs from VDM++, e.g. lambda expressions, local function definitions and higher-order functions, maps nicely to TypeScript. The VDMTools Java Code Generator does not support these constructs. The Overture Java Code Generator simulates lambda expressions by generating classes that represent the VDM++ function type, but it is not a native language construct[2] as in TypeScript. Many of the expression translations in the VDM-TS Code Generator benefit from TypeScript being a functional language. For example, *let expressions* are represented in TypeScript by an immediately-invoked function expression (IIFE). On the contrary, *let expressions* will in Java be translated in an imperative manner involving statements and by introducing a block scoped variable. For this reason, the Overture Java Code Generator does not support translation of let expressions in value definitions.

[2]Native lambda expressions are new in Java 8 which future versions of the Overture Java Code Generator might benefit from.

Both TypeScript and Java are class-based languages, but neither of them support multiple inheritance. In the Overture Java Code Genetator multiple inheritance is simply not supported. In the VDMTools Code Generator, multiple inheritance is substituted with interfaces. The VDM-TS Code Generator incorporates the concept of mixins which is possible because methods can be added to classes at runtime in TypeScript. Mixins allow subclasses to inherit actual implementations from multiple base classes, as opposed to the Java interface approach. However, interfaces in Java permit multiple inheritance of static fields (which represent VDM++ values), and additionally establish a subtype relationship which cannot be reflected in TypeScript. In general, it is easier to reflect runtime semantics of VDM++ in Java, because static type information is not erased at compile time.

6.3. Reflection on the Thesis Project

This master thesis project has been an unique opportunity to work independently and gain deeper knowledge of the selected field of study. While the process itself has been enriching, reflection on work carried out enhances its meaning. Reflecting on the things that went well, and in particular things that proved challenging during the thesis project, is a great way to foster the learning outcome. Being aware of personal qualities and areas for improvement is something one can benefit from not only now, but also in the future career.

One of the greater challenges during this thesis project was when to stop analysing possible ways to represent VDM++ constructs in TypeScript, before implementing the translation in the code generator. The difficulty arose from the fact that many times a new VDM++ construct was considered, the proposed translation introduced limitations to previously determined translations. In other words, the design choices made earlier influenced the possibilities of subsequent translations. Consequently, the design choices that was made had to be reiterated in order to find the best trade-offs, which delayed the implementation of each construct. Being able to proceed systematically and thoroughly understand the impact of choices is an essential competence when doing research. The time spent on analysis is the reason why bugs where found in the Overture Interpreter[3] and Overture Java Code Generator[4] during this thesis project. However, since almost every project has a limited time frame, including this thesis project, it is important not to fall into “analysis paralysis” that prevents progression of work. This is also relevant to software development outside academia. No matter how much analysis is made some problems will not reveal themselves until the product is being realised in code or actually is up and running. While a lot can be carried out to make educated decisions, it is in general an engineering discipline to be able to determine when to move forward from analysis. Analytical skills are valuable assets, but the authors should in the future avoid getting stuck because not every little detail is nailed down, and instead focus on the most

[3]For example, call by value semantics was not respected in all cases: `https://github.com/overturetool/overture/issues/544`, accessed 15th of May 2016.

[4]For example, a *let expression* within a *set enumeration* in *value definitions* made it fail with an “unexpected error". Reported by private communication with Peter W. V. Tran-Jørgensen, 12th of May, 2016.

critical design considerations.

Structuring this thesis proved to be a challenging task. In order to minimise the number of forward references and ease readability, a lot of effort was put into determining the presentation order of translations. However, this was not easy due to dependencies between translations. For example, translation of *type definitions* and its imposed limitations was chosen to be presented before translation of the *record type*, but the record type alleviates the limitation of type definitions. To avoid a forward reference, translation of the record type could have been introduced earlier, but it seemed counterintuitive to postpone the translation of classes and member definitions as they constitute a top-level specification. Writing this thesis has been a lesson in the importance of presentation considerations such that a flow from one subject to another is established, ultimately helping the reader to understand the message from the author.

Another challenge, related to the presentation challenge above, was to avoid imprecision in terminology throughout this thesis. For example, VDM++ has functions and operations that belong to a class definition, whereas TypeScript has functions and methods, but functions can be defined outside classes. Once TypeScript is compiled to JavaScript, there are technically only functions, however functions are often referred to as methods when used in a pattern that simulates classes. Hence, there are some overloading and confusion of terms, which is undesirable, in order to properly convey information. The ability to precisely convey information is an essential part of researching, because it lays the foundation for other research to build upon. In addition, having spent more than four months on a field of study, presence of the author's implicit assumptions is likely, which can have a negative effect on precision. Being able to identify situations prone to ambiguities and implicit assumptions is the first step towards improving communication skills in the future.

6.4. Revisiting the Hypothesis

This thesis project was set out to investigate the possibility of code generation from VDM++ to TypeScript. The investigation was conducted by first analysing how the semantics of VDM++ constructs can be represented by constructs in TypeScript and by identifying when the semantics cannot be reflected. Then, the best suited translations were implemented in the VDM-TS Code Generator in order to automate the translation process. The translations integrated in the VDM-TS Code Generator were continuously verified by small sample models, and later evaluated as a whole by conducting a case study. Due to a limited time frame, not all proposed translations where implemented in the code generator. Instead, focus in the final stages of the thesis project was on particularly challenging constructs in the search for boundaries between VDM++ and TypeScript.

The hypothesis of the thesis project was:

There exists a subset of VDM++, adequate for non-trivial models, that can be translated to TypeScript while preserving the semantics of VDM++.

The hypothesis is supported by the results obtained in the conducted case study, but also by validation of small sample models. Despite its simplicity, the Alarm model contains high-level expressions not present in TypeScript, and evaluation of the TypeScript representation indicates semantics preservation of VDM++. Therefore, the translation serves as a proof of existence. However, VDM++ and TypeScript are clearly dissimilar in spirit and with respect to syntax. Many VDM++ constructs require a lot of effort to be represented in TypeScript, and some of these impose restrictions to VDM++ models, if they shall be realised in TypeScript with automatic code generation. Moreover, this thesis does not cover all aspects of the VDM++ language. Many of its distinguishing features, such as Design-by-Contract elements, concurrency and pattern matching, remain unaddressed. In order to facilitate attainment of a more comprehensive code generator from VDM++ to TypeScript, further research is required.

6.5. Evaluation on the Achievement of Goals

In addition to the hypothesis, three goals were set. This section revisits these goals and evaluates whether they have been achieved.

6.5.1 Revisiting the goals

The goals were presented in chapter 1, and are shown again below:

Goal 1: Propose translations of VDM++ to TypeScript constructs and identify semantics preserving limitations of these translations. This requires a study of the languages involved.

Goal 2: Develop a proof-of-concept VDM++ to TypeScript code generator based on the findings from Goal 1.

Goal 3: Validate the translation of a non-trivial VDM++ model produced by the proof-of-concept code generator by conducting a case study.

6.5.2 Evaluation of goals

The goals are evaluated separately by briefly explaining why each goal is considered to be achieved and which parts of this thesis project the goal is related to.

Goal 1: This goal is *achieved.* The goal is considered achieved because concrete translation rules along with potential semantics preserving limitations have been presented in this thesis. Translation of fundamental VDM++ constructs is presented in a progressive manner in chapter 3, starting with constructs that constitute a top-level specification. Translation of particularly challenging constructs is presented in chapter 5. Additionally, appendix A contains trivial translations and translations that are similar in nature. Learning the semantics and syntax of VDM++, TypeScript and JavaScript has been a necessity for identifying possible translations. The fundamental language concepts and constructs is presented in chapter 2. This has allowed careful deliberation for translation of each VDM++ construct, before being summarised in a translation rule. The proposed translations in this thesis do not cover all VDM++ constructs, however, the goal is still considered achieved as this was not part of the thesis scope.

Goal 2: This goal is *partly achieved*. During the thesis project the VDM-TS Code Generator has been developed based on the Overture Code Generation Platform introduced in chapter 2. The goal is achieved in the sense that the code generator implements translations from chapter 3 necessary to conduct the case study described in chapter 4. The goal is considered partly achieved because some of the proposed translations presented in chapter 3 and chapter 5 have not been implemented in the code generator due to time limitations. Integration of these translations is subject for future work.

Goal 3: This goal is *achieved*. The case study described in chapter 4 has been used to validate the translation of the VDM++ Alarm model to TypeScript. The validation was based on inspection and execution tests in multiple environment, and the result demonstrated that the TypeScript version performed in accordance with the VDM++ model. Hence, the case study provides an increased level of certainty with regards to semantics preserving translations. Whether this goal is achieved depends on the definition of a non-trivial model. In chapter 1, a non-trivial model was defined as one having qualities of real-life application and containing set-theoretic structures, logic and operations typically not found in traditional programming languages. By this definition the Alarm model is non-trivial, even though design-by-contract elements were intentionally left out in the case study, because it is an example model that is inspired by a real-life system and it contains high-level expressions, such as set comprehension, let-be-such-that and quantified expressions. Therefore, the goal is considered achieved. However, it can be discussed whether non-trivial was initially too vaguely defined, because some of the distinctive features of VDM++ was not covered by the case study, such as Design-by-Contract elements.

6.6. Future Work

The achievements of this thesis demonstrates that a subset of VDM++ can be translated to TypeScript. Further research is required to facilitate code generation of a larger subset and improvements of the VDM-TS Code Generator. This section describes possible directions of the future work envisaged.

Adding support for additional VDM++ constructs: The proposed mixin-based approach to multiple inheritance described in subsection 5.2.2 is not implemented in the VDM-TS Code Generator, however it should be straightforward to do so. Is expressions for compound data types is neither supported, but a translation is proposed for the set type in section 5.4, and support for other compound data types should be achievable in a similar fashion. Some translations have been considered "more of the same" and therefore not treated in this thesis. An example is the sequence type which is currently not supported by the code generator. An implementation of this type should be similar to the implementation of the set type (represented as `Set<T>`) and thereby remain subject for future work.

Design-by-Contract: Since VDM++ is an extension of VDM-SL, it is also a Design-by-Contract language that can provide guarantees of the correctness in a specification. These guarantees should be reflected when a VDM++ specification is realised in a high-level programming language. Thus, contract-based elements from VDM++, such as *invariants*, *pre-* and *post-conditions*, should be considered in the code generation process. However, TypeScript has no native Design-by-Contract support and no built-in assertion functionality[5] that can help ensure the consistency of the generated VDM++. Some JavaScript libraries offer support for the Design-by-Contract methodology [6] which may serve as viable candidates for the code generation of the contract-based elements. An investigation of these libraries is required to determine their usefulness in a VDM++ and TypeScript context. Failing to identify a suitable library will entail the need for a manual implementation of the Design-by-Contract functionality in the VDM-TS Code Generator. This includes an assertion function in the VDM-TS Library to assist the translation of contract-based elements from VDM++ to TypeScript.

Optimise performance: The source code generated by the VDM-TS Code Generator can potentially be used in a wide variety of TypeScript applications and it might therefore be necessary to consider performance. The performance aspect is outside the scope of this thesis project, but it would be beneficial to revisit the implementations in the VDM-TS Code Generator and the VDM-TS Library for optimisation purposes. For example, the VDM-TS Code Generator uses an aggressive cloning strategy to obtain the value-semantics of VDM++. Cloning is in many cases not necessary in the current implementation. The VDMTools VDM++ to Java Code Generator utilises a reference counter mechanism in order to minimise the number of clone function invocations. This approach can be adopted in the VDM-TS Code Generator.

[5]Some execution environments offer an assert feature, but it is not part of the ECMAScript standard and consequently the functionality is widely different `https://developer.mozilla.org/en-US/docs/Web/API/console/assert`, accessed 18th of May 2016.

[6]Design-by-Contract JavaScript libraries:
Cerny.js: `http://www.cerny-online.com/cerny.js/`, accessed 18th of May 2016,
jsContract: `http://kinsey.no/projects/jsContract/`, accessed 18th of May 2016,
AspectJS: `http://www.aspectjs.com/`, accessed 18th of May 2016.

Integration in the Overture tool suite: The VDM-TS Code Generator is based on the Overture Code Generation Platform but is otherwise decoupled from the Overture tool. Integrating the code generator in the Overture tool allows testing and usage by a larger audience which hopefully can provide valuable feedback and help detection of bugs.

6.7. Final Remarks

The hypothesis was, that a subset of VDM++ can be translated to TypeScript while preserving the semantics of VDM++. The hypothesis has been supported by the results obtained in this thesis project. However, the work carried out does not cover all aspects of VDM++. Thus, further research is required in order to determine whether more diverse VDM++ specifications can be code generated to TypeScript.

In addition to the hypothesis, three goals were partly achieved. The achievement of these goals mean, that the authors have conducted a study of VDM++ and TypeScript, proposed and integrated translations in a proof-of-concept code generator, and tested, evaluated and verified these translations by conducting a case study.

To conclude this thesis, it is the authors hope that the obtained results provide a foundation for further research in order to broaden the application areas for model-based specification. Additionally, it is the hope that the proposed translation rules can serve as an inspiration for code generation from VDM++ to other target languages.

Bibliography

[1] P. W. V. Jørgensen, K. Lausdahl, and P. G. Larsen, "An Architectural Evolution of the Overture Tool," in *The Overture 2013 workshop*, August 2013. [cited at p. viii, 12, 13]

[2] J. Krumm, *Ubiquitous Computing Fundamentals*. Chapman & Hall/CRC, 1st ed., 2009. [cited at p. 1]

[3] The Royal Academy of Engineering, "The Challenges of Complex IT Projects," tech. rep., Royal Academy of Engineering, 2004. [cited at p. 1]

[4] G.-C. Roman, C. Julien, and Q. Huang, "Formal specification and design of mobile systems," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, (Washington, DC, USA), pp. 160–, IEEE Computer Society, 2002. [cited at p. 1]

[5] H.Bekić, D.Bjørner, W.Henhapl, C.B.Jones, and P.Lucas, "A Formal Definition of a PL/I Subset," Tech. Rep. 25.139, IBM Laboratory, Vienna, December 1974. [cited at p. 1]

[6] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object–oriented Systems*. Springer, New York, 2005. [cited at p. 1, 5, 10, 24, 42]

[7] C. F. Ngolah and Y. Wang, "Exploring java code generation based on formal specifications in rtpa," in *Electrical and Computer Engineering, 2004. Canadian Conference on*, vol. 3, pp. 1533–1536, IEEE, 2004. [cited at p. 1]

[8] Overture-Core-Team, "Overture Web site." http://www.overturetool.org, 2007. [cited at p. 1, 5, 10, 42, 53]

[9] P. W. V. Jørgensen, M. Larsen, and L. D. Couto, "A Code Generation Platform for VDM," in *Proceedings of the 12th Overture Workshop, Newcastle University, 21 June, 2014* (N. Battle and J. Fitzgerald, eds.), School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446, January 2015. [cited at p. 1, 4, 5, 9, 11, 12, 13, 67]

[10] L. M. Barroca and J. A. McDermid, "Formal methods: Use and relevance for the development of safety-critical systems," *The Computer Journal*, vol. 35, no. 6, pp. 579–599, 1992. [cited at p. 2]

[11] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software Engineering Journal*, vol. 8, pp. 189–209, Jul 1993. [cited at p. 2]

[12] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web.," in *OSDI*, vol. 8, pp. 339–354, 2008. [cited at p. 2]

[13] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, (Berlin, Heidelberg), pp. 238–255, Springer-Verlag, 2009. [cited at p. 2, 3]

[14] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris, "Safe & efficient gradual typing for typescript," in *ACM SIGPLAN Notices*, vol. 50, pp. 167–180, ACM, 2015. [cited at p. 2]

[15] "The TypeScript website," April 2016. `http://www.typescriptlang.org/`. [cited at p. 2, 17]

[16] S. Fenton, *Pro TypeScript: Application-Scale JavaScript Development*. Berkely, CA, USA: Apress, 1st ed., 2014. [cited at p. 3]

[17] D. Crockford, "Syntaxation," October 2015. `https://www.gotoconf.com/`. [cited at p. 3]

[18] Miran Hasanagic, Peter Gorm Larsen, Marcel Groothuis, Despina Davoudani, Adrian Pop, Kenneth Lausdahl, Victor Bandur, "Design Principles for Code Generators," tech. rep., INTRO-CPD, 2015. [cited at p. 5]

[19] CSK, "VDMTools homepage." *http://www.vdmtools.jp/en/*, 2007. [cited at p. 5, 67]

[20] J. Herrington, *Code Generation in Action*. Greenwich, CT, USA: Manning Publications Co., 2003. [cited at p. 9]

[21] T. g. Mogensen, *Introduction to Compiler Design*. Springer, 2011. [cited at p. 9]

[22] J. Kramer, "Is Abstraction the Key to Computing?," *Communications of the ACM*, vol. 50, no. 4, pp. 37–42, 2007. [cited at p. 9]

[23] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in Support for Formal Modeling in VDM," *ACM Sigplan Notices*, vol. 43, pp. 3–11, February 2008. [cited at p. 10, 58]

[24] H.Bekić, D.Bjørner, W.Henhapl, C.B.Jones, and P.Lucas, "A Formal Definition of a PL/I Subset," Tech. Rep. 25.139, IBM Laboratory, Vienna, December 1974. [cited at p. 10]

[25] P. G. Larsen, B. S. Hansen, H. Brunn, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, *et al.*, "Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language," December 1996. [cited at p. 10]

[26] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *SIGSOFT Softw. Eng. Notes*, vol. 35, pp. 1–6, January 2010. [cited at p. 10]

[27] P. G. Larsen, K. Lausdahl, N. Battle, J. Fitzgerald, S. Wolff, S. Sahara, M. Verhoef, P. W. V. Tran-Jørgensen, and T. Oda, "The VDM-10 Language Manual," Tech. Rep. TR-2010-06, The Overture Open Source Initiative, April 2010. [cited at p. 11]

[28] P. G. Larsen, C. Thule, K. Lausdahl, V. Bardur, C. Gamble, K. Pierce, E. Brosse, A. Sadovykh, A. Bagnato, and L. D. Couto, "Integrated Tool Chain for Model-based Design of Cyber-Physical Systems," in *Submitted for FM 2016*, (Cyprus), FME, November 2016. [cited at p. 11, 52]

[29] P. G. Larsen, K. Lausdahl, P. Jørgensen, J. Coleman, S. Wolff, and N. Battle, "Overture VDM-10 Tool Support: User Guide," Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org, April 2013. [cited at p. 11]

[30] L. D. Couto, P. W. V. Tran-Jørgensen, and K. Lausdahl, "Principles for Reuse in Formal Language Tools," in *31st ACM Symposium on Applied Computing*, April 2016. [cited at p. 12]

[31] "The Apache Velocity website," 2015. `http://velocity.apache.org/`. [cited at p. 13]

[32] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. [cited at p. 13, 14, 15, 16]

[33] E. International, "The ECMA International website," March 2016. `http://www.ecma-international.org/`. [cited at p. 14]

[34] "Mozilla Developer Network," March 2016. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. [cited at p. 14, 15]

[35] "Node.js," April 2016. `https://nodejs.org/en/`. [cited at p. 14, 19, 35, 53]

[36] A. Hejlsberg, "Introducing TypeScript," October 2012. `https://channel9.msdn.com/posts/Anders-Hejlsberg-Introducing-TypeScript`. [cited at p. 17]

[37] J. T. Ryan Cavanaugh, "TypeScript Language Wiki," 2016. `https://github.com/Microsoft/TypeScript-wiki/blob/master/TypeScript-Design-Goals.md`. [cited at p. 17]

[38] N. Rozentals, *Mastering Typescript*. Community Experience Distilled, Packt Publishing, 2015. [cited at p. 18]

[39] S. K. Bansal, *Dictionary of it Terms*. APH Publishing Corporation, 2009. [cited at p. 21]

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. [cited at p. 22]

[41] "TypeScript Language Specification," April 2016. `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#3.10`. [cited at p. 24, 33]

[42] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter, "Detecting function purity in javascript," in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pp. 101–110, IEEE, 2015. [cited at p. 27]

[43] C. S. Hostmann, *Object-Oriented Design and Patterns (2. ed.)*. Wiley, 2005. [cited at p. 29]

[44] "The TypeScript Handbook," April 2016. `http://http://www.typescriptlang.org/docs/tutorial.html`. [cited at p. 31]

[45] T. V. T. Group, "The vdm++ to c++ code generator," tech. rep., CSK Systems, January 2008. [cited at p. 32]

[46] D. E. Perry, S. E. Sim, and S. M. Easterbrook, "Case studies for software engineers," in *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pp. 736–738, 2004. [cited at p. 41]

[47] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds., *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972. [cited at p. 41]

[48] B. Meyer, "Seven principles of software testing," *Computer*, vol. 41, no. 8, pp. 99–101, 2008. [cited at p. 42]

[49] H. B. Christensen, *Flexible, Reliable Software: Using Patterns and Agile Development*. Chapman & Hall/CRC, 1st ed., 2010. [cited at p. 42]

[50] K. Beck, *Test-driven Development: By Example*. A Kent Beck signature book, Addison-Wesley, 2003. [cited at p. 42]

[51] I. Sommerville, *Software Engineering*. International Computer Science Series, Pearson, 2011. [cited at p. 42]

[52] J. Greene and A. Stellman, *Applied Software Project Management*. O'Reilly, first ed., 2005. [cited at p. 51]

[53] Pivotal-Labs, "The Jasmine Test Framework," March 2016. `http://jasmine.github.io/`. [cited at p. 52]

[54] J. Grigg, "The Arrange-Act-Assert pattern for unit testing," July 2012. `http://c2.com/cgi/wiki?ArrangeActAssert`. [cited at p. 52]

[55] L. Cardelli, “A semantics of multiple inheritance,” *Information and Computation*, vol. 76, no. 2, pp. 138–164, 1988. [cited at p. 55]

[56] S. Danforth and C. Tomlinson, “Type theories and object-oriented programmimg,” *ACM Computing Surveys (CSUR)*, vol. 20, no. 1, pp. 29–72, 1988. [cited at p. 56]

[57] G. Bracha and W. Cook, “Mixin-based inheritance,” *ACM Sigplan Notices*, vol. 25, no. 10, pp. 303–311, 1990. [cited at p. 58]

[58] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and mixins,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 171–183, ACM, 1998. [cited at p. 58]

[59] D. Malayeri and J. Aldrich, “CZ: Multiple inheritance without diamonds,” in *OOPSLA ’09*, October 2009. [cited at p. 59]

[60] J. Zaytsev, “ECMAScript sompatibility tabel,” March 2016. `http://kangax.github.io/compat-table/es6/`. [cited at p. 66]

[61] ECMA-International, “ECMAScript 2017 Language Specification,” May 2016. `https://tc39.github.io/ecma262/`. [cited at p. 66]

Appendices

Appendix A

Translation Rules

This appendix contains the complete collection of translation rules proposed in this thesis project. These translation rules are presented by grouping related translations into sections.

A.1. Basic Data Types

Translation 1. Boolean type

The boolean type in VDM++ is code generated to its counterpart in TypeScript.

Translation 2. Numeric types

The five numeric types in VDM++ are mapped to the only number type in TypeScript.

Translation 3. Character type

The character type in VDM++ are represented as a TypeScript strings with length one.

Translation 4. Quote Types

The quote type in VDM++ is code generated to a class that implements the singleton pattern.

Translation 5. Token Types

Let t be a token type in VDM++ wrapping an arbitrary expression e. Then t is code generated to a class derived from the base class `Token` containing an instance member that holds the meaning of e.

Translation 6. Sequence of characters

The **`seq of char`** type in VDM++ are code generated to the string type in TypeScript.

A.2. Classes

Translation 7. Classes

Classes in VDM++ become the equivalent class construct in TypeScript.

Translation 8. Type definitions

Type definitions become exported *type aliases* in the associated namespace of a class.

Translation 9. Value definitions

Values become an exported *const declaration*, applied with `Object.freeze`, residing in the associated namespace of a class.

Translation 10. Instance variables

Instance variables become *instance members* of a TypeScript class with explicit access modifiers similar to VDM++ either declaring public, protected or private accessibility.

Translation 11. Function and operation definitions

Both VDM++ functions and operations are code generated as methods of a TypeScript class. Methods representing functions are declared with the **`static`** keyword. An operation with an identical name to the class is translated to a constructor in the TypeScript class.

Translation 12. Single Inheritance

Let S be a subclass in VDM++ that inherits from base class B. Then S is translated to a TypeScript class with a corresponding name that denote B in its **`extends`** clause.

Translation 13. Multiple Inheritance

Let D be a subclass of the base classes $B_1, \ldots, B_n$.
Then D is translated to a TypeScript class with a corresponding name that extends B_1 and implements $B_2, \ldots, B_n$. The `applyMixins` function from the VDM-TS Library is invoked immediately after D has been defined.

A.3. Compound Date Types

Translation 14. Record types

Let R be a VDM++ record (composite) type with fields $f_1, \ldots, f_n$ with corresponding types $t_1, \ldots, t_n$. Then R is code generated to a TypeScript class with identical name and with instance members $f_1, \ldots, f_n$ annotated with $t_1, \ldots, t_n$. All classes that represents a record type derives from base class `Record`.

Translation 15. Set types

Let S be a VDM++ set type comprised of arbitrary complex elements $e_1, \ldots, e_n$ of same type t. Then S is translated to the generic class `Set<T>` where the type parameter `T` becomes t. Values are created by making instances of the class and passing $e_1, \ldots, e_n$ as arguments to the constructor. Explicit type annotation of `T` is only needed when t is a union type.

Translation 16. Map types

The translated map type in TypeScript is a Map class containing an array for holding key-value pairs represented as a Maplet classes with a key and value property.

Translation 17. Tuple type

The tuple (product) type in VDM++ is code generated to its counterpart tuple type in TypeScript.

Translation 18. Function types

Let F be a VDM++ *function type* from (tuple) type A to type B. Then F is code generated as a TypeScript function type where $A_1, \ldots, A_n$ become parameters with unique parameter names $x_1, \ldots, x_n$ and B becomes the return type.

Translation 19. Union types

The VDM++ union type comprised of the types $t_1, ..., t_n$ is code generated to the built-in union type in TypeScript.

A.4. Value and Reference Semantics

Translation 20. Obtaining Value Semantics

Apply the VDM-TS Library `clone` function on TypeScript objects that represent a value type in VDM++ when it:

- is passed as an argument to a function.
- appears on the right-hand-side of an assignment.

Translation 21. Class Dependencies Caused by Object References

Each code generated TypeScript class is contained in its own file specifying its dependencies using the TypeScript module import declaration `import`.

A.5. Expressions

Translation 22. Let Expressions

Let l be a *let expression* in VDM++ that denotes an expression e involving the pattern identifiers $i_1, \dots, i_n$ of the patterns $p_1, \dots, p_n$ matched against the corresponding expressions $e_1, \dots, e_n$. Then l is translated as an immediately-invoked function expression where the arguments become $e_1, \dots, e_n$, the formal parameters become $i_1, \dots, i_n$ and the body of the function becomes e.

Translation 23. If Expressions

If expressions become an inline conditional expression using the ternary operator.

Translation 24. Set Comprehension Expression

Single set bind: Let S be a *set comprehension expression* in VDM++ with a single set bind. Then S is code generated to a set of the single bind applied with the filter and map functions.

Multiple set binds: Let S be a *set comprehension expression* in VDM++ with multiple binds. Then S is code generated to an IFFE wrapping nested loop constructs iterating all multiple set binds and a condition with the predicate for adding the evaluated expression to the resulting set.

Translation 25. Let-be-such-that Expressions

Let L be a *let-be-such-that expression* in VDM++ with a multiple set bind. Then L is code generated to an IFFE wrapping nested loop constructs iterating the multiple set bind and a condition of the `be st` predicate.

Translation 26. Quantified Expressions

Unique existential: Let bd be a set bind and e be an boolean expression involving the pattern identifier of bd. Then the *Unique existential expression* is code generated to the set from bd applied with a filter of the boolean expression e for which the cardinality is exactly to one.

Existential: Let mbi's be multiple binds and e be an boolean expression involving the pattern identifier of mbi. Then the *Existential expression* is code generated to an IFFE wrapping nested loop constructs iterating all mbi and applying the `some` function with e to each mbi.

Universal: Let mbi's be multiple binds and e be an boolean expression involving the pattern identifier of mbi. Then the *Universal expression* is code generated to an IFFE wrapping nested loop constructs iterating all mbi and applying the `every` function with e to each mbi.

Translation 27. Membership Expressions

The four class membership expressions, *isofclass*, *sameclass*, *isofbaseclass* and *same-baseclass*, become TypeScript functions with corresponding names residing in the VDM-TS Library.

Translation 28. Is expressions

Basic type: Let i be either a boolean, numeric, character or token *basic is expression* with a value e. The i is code generated to an expression involving **`typeof`** or **`instanceof`** of e depending on the type of the is expression.

General: Let i be a *general is expression* with a value e and a type t. Then i is code generated to an expression of nested IFFEs constructed recursively from e for checking every sub-types of t.

Translation 29. Apply expression

The VDM++ apply expression is code generated to a method or function invocation in TypeScript.

Translation 30. New expression

Let n be a **`new`** expression in VDM++ having a classname c and the parameters $p_1, \ldots, p_n$. Then n is code generated to a **`new`** operator, instantiating the class c by calling c's constructor with the parameters $p_1, \ldots, p_n$.

A.6. Statements

Translation 31. Assignment statement

The VDM++ assignment statement is code generated to its counterpart in TypeScript.

Translation 32. Let and Define statements

Let l be a simple let or define statement in VDM that creates a scope around a statement s where $i_1, \ldots, i_n$ identifiers are bound to $e_1, \ldots, e_n$ expressions.
Then l is code generated as a **`const`** declaration where $i_1, \ldots, i_n$ identifiers are bound to $e_1, \ldots, e_n$ expressions, followed by the statement s and wrapped in curly brackets.

Translation 33. Conditional statements

Let i be an if-statement in VDM where e is a boolean expression, $s1$ and $s2$ are statements.
Then i is code generated as an if-statement where e is the boolean condition expression, $s1$ is the statement evaluated if e evaluates to true and $s2$ is and optional statement being evaluated if e evaluates to false.

Translation 34. While-loop statemants

Let w be a while-loop statement in VDM where e is boolean expression and s is a statement.
Then w is code generated as a while-loop statement in TypeScript where e is boolean expression and s is a statement.

Translation 35. Index for-loop statements

Let f be an index for-loop in VDM++where i is an identifier, $e1$, $e2$ and $e3$ are integer expressions and s is a statement.
Then f is code generated as a for-loop statement in TypeScript where i is an identifier, $e1$, $e2$ and $e3$ are integer expressions and s is a statement.

Translation 36. Set and sequence for-loop statements

Let f be a set or sequence for-loop statement in VDM++where e is a pattern bound to subsequent values from S, S being either a set or sequence expression. s is a statement being evaluated in the environment extended with e.
Then f is code generated as a for-of statement in TypeScript where e is a variable assigned to subsequent values from S, S being a iterable object either a set or sequence. s is the statement being evaluated.

Translation 37. Call statement

Let c be a call statement inVDM++ with parameters $p_1, \ldots, p_n$. Then c is code generated to a TypeScript method invocation with the parameters $p_1, \ldots, p_n$.

Translation 38. Return statement

Let r be a return statement in VDM++ returning an expression e. Then r is code generated to the equivalent return statement in TypeScript returning e.

Appendix B

The VDM-TS Library

This Appendix presents the VDM-TS Library which provide functionality the can help reflect the semantics of VDM++ in the VDM-TS Code Generator. An overview of the functionality implemented in the VDM-TS Library is given in section B.1. Additionally, the code generation of the Set and Map operations provided by the VDM-TS Library is presented in section B.2 and section B.3.

B.1. VDM-TS Library Overview

This section contains an overview, shown below in Table B.1, of the VDM++ functionality and semantics reflected by the VDM-TS Library.

VDM-TS Library	**Reflected VDM++ Functionality**
Functions	
`equals`	Type equality
`clone`	Pass-by-value semantics
`range`	Set range expression
`isofclass`	Class Membership
`isofbaseclass`	Base Class Membership
`sameclass`	Same Class Membership
`samebaseclass`	Same Base Class Membership
Classes	
`Quote` base class	Quote type
`Token` base class	Token type
`Record` base class	Record type
`Set<T>` class	Set type
`Map<K,V>` class	Map type
`Maplet<K,V>` class	Maplet

Table B.1: VDM-TS Library overview

In addition, the VDM-TS Library includes a `pretty` function that can be used to obtain a textual representation of VDM++ values.

B.2. Set Type Operations

This section provide an overview of the `Set` operations contained in the VDM-TS Library.

Name	**VDM++**	**TypeScript**
Membership	`e` **`in set`** `s1`	`s1.inset(e)`
Not membership	`e` **`not in set`** `s1`	`!s1.inset(e)`
Union	`s1` **`union`** `s2`	`s1.union(s2)`
Intersection	`s1` **`inter`** `s2`	`s1.inter(s2)`
Difference	`s1 \ s2`	`s1.diff(s2)`
Subset	`s1` **`subset`** `s2`	`s1.subset(s2)`
Proper subset	`s1` **`psubset`** `s2`	`s1.psubset(s2)`
Equality	`s1 = s2`	`s1.equals(s2)`
Inequality	`s1 <> s2`	`!s1.equals(s2)`
Cardinality	**`card`** `s1`	`s1.card()`
Distributed union	**`dunion`** `ss`	`s1.union(s2)...union(sn)`
Distributed intersection	**`dinter`** `ss`	`s1.inter(s2)...inter(sn)`
Finite power set	**`power`** `s1`	`s1.power()`

Table B.2: `Set` operations provided by VDM-TS Library

B.3. Map Type Operations

This section provide an overview of the `Map` operations contained in the VDM-TS Library.

Name	VDM++	TypeScript
Domain	**`dom`** `m`	`m.dom()`
Range	**`rng`** `m`	`m.rng()`
Merge	`m1` **`munion`** `m2`	`m1.munion(m2)`
Override	`m1 ++ m2`	`m1.override(m2)`
Distributed merge	**`merge`** `ms`	`m1.merge(m2).merge(mn)`
Domain restrict to	`s <: m`	`m.domRestrictTo(s)`
Domain restrict by	`s <-: m`	`m.domRestrictBy(s)`
Range restrict to	`m :> s`	`m.rngRestrictTo(s)`
Range restrict by	`m :-> s`	`m.rngRestrictBy(s)`
Map apply	`m(d)`	`m.apply(d)`
Map composition	`m1` **`comp`** `m2`	`m1.comp(m2)`
Map iteration	`m ** n`	`n = 0: m.identity()`
		`n = 1: m`
		`n > 1: m.comp(m.comp(...))`
Equality	`m1 = m2`	`m1.equals(m2)`
Inequality	`m1 <> m2`	`!m1.equals(m2)`
Map inverse	**`inverse`** `m`	`m.inverse()`

Table B.3: `Map` operations provided by VDM-TS Library

Appendix C

Code Generation: The Alarm Model

This Appendix contains the entire code generation of the Alarm model from VDM++ to TypeScript. The code generation is presented following a class-by-class approach, where each section presents a VDM++ class followed by the code generated TypeScript class. In order for the code generated TypeScript to compile and run the VDM-TS library must be included.

C.1. The `Expert` class

```
class Expert

instance variables
    quali : set of Qualification;

types
    public Qualification = <Mech> | <Chem> | <Bio> | <Elec>;

operations
    public Expert: set of Qualification ==> Expert
    Expert(qs) == quali := qs;

pure public GetQuali: () ==> set of Qualification
    GetQuali() == return quali;

end Expert
```

```
class Expert {
    private quali: Set<Expert.Qualification> = undefined;
    constructor (qs: Set<Expert.Qualification>) {
        this.quali = qs;
    }
    public GetQuali() : Set<Expert.Qualification> {
        return this.quali
    }
}
namespace Expert {
    export type Qualification =
        Quotes.Bio|Quotes.Chem|Quotes.Elec|Quotes.Mech;
}
```

Quotes

The quotes declared in the `Quotes` namespace are directly related to the code generation of the `Expert` class as it define a type `Qualification` being a union type of multiple quote types.

```
namespace Quotes {
    export class Elec extends Quote {
        private static instance = new Elec();
        constructor() {
            super();
            if (Elec.instance) {
                throw Error("Use getInstance()");
            }
            Elec.instance = this;
        }
        public static getInstance() {
            return this.instance;
        }
        public toString() {
            return '<Elec>';
        }
        public equals(quote: Quote) {
            return quote instanceof Elec;
        }
    }
    export class Bio extends Quote {
        private static instance = new Bio();
        constructor() {
            super();
            if (Bio.instance) {
                throw Error("Use getInstance()");
            }
            Bio.instance = this;
        }
        public static getInstance() {
            return this.instance;
        }
        public toString() {
            return '<Bio>';
        }
        public equals(quote: Quote) {
            return quote instanceof Bio;
        }
    }
    export class Mech extends Quote {
        private static instance = new Mech();
        constructor() {
            super();
            if (Mech.instance) {
                throw Error("Use getInstance()");
            }
            Mech.instance = this;
        }
        public static getInstance() {
            return this.instance;
        }
        public toString() {
            return '<Mech>';
        }
        public equals(quote: Quote) {
            return quote instanceof Mech;
        }
    }
    export class Chem extends Quote {
        private static instance = new Chem();
        constructor() {
            super();
            if (Chem.instance) {
                throw Error("Use getInstance()");
            }
            Chem.instance = this;
        }
        public static getInstance() {
            return this.instance;
        }
        public toString() {
            return '<Chem>';
        }
        public equals(quote: Quote) {
            return quote instanceof Chem;
        }
    }
}
```

C.2. The **Alarm** class

```
class Alarm
types
    public String = seq of char;

instance variables
    descr    : String;
    reqQuali : Expert`Qualification;
```

```

operations

    public Alarm: Expert`Qualification * String ==> Alarm
    Alarm(quali,str) ==
        ( descr := str;
          reqQuali := quali
        );

    pure public GetReqQuali: () ==> Expert`Qualification
    GetReqQuali() ==
       return reqQuali;

end Alarm
```

```
class Alarm {
    private descr: string = undefined;
    private reqQuali: Expert.Qualification = undefined;
    constructor (quali: Expert.Qualification, str: string) {
        this.descr = str;
        this.reqQuali = quali;
    }
    public GetReqQuali() : Expert.Qualification {
        return this.reqQuali
    }
}
namespace Alarm {
    export type String = string;
}
```

C.3. The `Plant` class

```
class Plant

instance variables
    alarms   : set of Alarm;
    schedule : map Period to set of Expert;
    inv PlantInv(alarms,schedule);

functions

    PlantInv: set of Alarm * map Period to set of Expert -> bool
    PlantInv(sa,sch) ==
    (forall p in set dom sch & sch(p) <> {}) and
        (forall a in set sa &
            forall p in set dom sch &
            exists expert in set sch(p) &
```

```
                    a.GetReqQuali() in set expert.GetQuali());

types
    public Period = token;

operations

    public ExpertToPage: Alarm * Period ==> Expert
    ExpertToPage(a, p) ==
        return let expert in set schedule(p) be st
            a.GetReqQuali() in set expert.GetQuali()
        in
            expert
    pre a in set alarms and
        p in set dom schedule
    post let expert = RESULT
    in
        expert in set schedule(p) and
        a.GetReqQuali() in set expert.GetQuali();

    public NumberOfExperts: Period ==> nat
    NumberOfExperts(p) ==
        return card schedule(p)
    pre p in set dom schedule;

    public ExpertIsOnDuty: Expert ==> set of Period
    ExpertIsOnDuty(ex) ==
        return {p | p in set dom schedule &
            ex in set schedule(p)};

    public Plant: set of Alarm * map Period to set of Expert ==> Plant
    Plant(als,sch) ==
        ( alarms := als;
          schedule := sch
        )
    pre PlantInv(als,sch);

end Plant
```

```
class Plant {
    private alarms: Set<Alarm> = undefined;
    private schedule: Map<Token,Set<Expert>> = undefined;
    static PlantInv(sa: Set<Alarm>, sch: Map<Token,Set<Expert>>) {
        return sch.dom().every(p =>
            !_.equals(sch.apply(p), new Set()))
            && sa.every(a =>
                sch.dom().every(p =>
                    sch.apply(p).some(expert =>
                        expert.GetQuali().inset(a.GetReqQuali()))))
    }

    public ExpertToPage(a: Alarm, p: Token) : Expert {
        return (() => {
            var res;
            this.schedule.apply(p).forEach(expert => {
                if(expert.GetQuali().inset(a.GetReqQuali())) {
                    res = expert;
                }
            });
            if (!res) {
                throw 'Let be st found no applicable bindings';
            };
            return res;
        })();
    }
    public NumberOfExperts(p: Token) : number {
        return this.schedule.apply(p).card()
    }
    public ExpertIsOnDuty(ex: Expert) : Set<Token> {
        return (() => {
            var res = new Set<Token>();
            this.schedule.dom().forEach(p => {
                if(this.schedule.apply(p).inset(ex)) {
                    res.add(p);
                }
            });
            return res;
        })();
    }
    constructor (als: Set<Alarm>, sch: Map<Token,Set<Expert>>) {
        this.alarms = als;
        this.schedule = sch;
    }
}
namespace Plant {
    export type Period = Token;
}
```

C.4. The `Test1` class

```
class Test1

instance variables
    a1   : Alarm   := new Alarm(<Mech>,"Mechanical fault");
    a2   : Alarm   := new Alarm(<Chem>,"Tank overflow");
    ex1  : Expert  := new Expert({<Mech>,<Bio>});
    ex2  : Expert  := new Expert({<Elec>});
    ex3  : Expert  := new Expert({<Chem>,<Bio>,<Mech>});
    ex4  : Expert  := new Expert({<Elec>,<Chem>});
    plant: Plant   := new Plant({a1},{p1 |-> {ex1,ex4},
                                      p2 |-> {ex2,ex3}});

```

```
types
    public PE :: periods : set of Plant`Period expert : Expert

values
    p1: Plant`Period = mk_token("Monday day");
    p2: Plant`Period = mk_token("Monday night");
    p3: Plant`Period = mk_token("Tuesday day");
    p4: Plant`Period = mk_token("Tuesday night");

operations
    public Run: () ==> PE
    Run() ==
        let periods = plant.ExpertIsOnDuty(ex1),
            expert  = plant.ExpertToPage(a1,p1)
        in
            return mk_PE(periods,expert);
end Test1
```

```
class Test1 {
    private a1: Alarm =
            new Alarm(Quotes.Mech.getInstance(), "Mechanical fault");
    private a2: Alarm =
            new Alarm(Quotes.Chem.getInstance(), "Tank overflow");
    private ex1: Expert =
            new Expert(new Set<Quotes.Bio|Quotes.Mech>(
              Quotes.Mech.getInstance(),Quotes.Bio.getInstance()));
    private ex2: Expert =
            new Expert(new Set(Quotes.Elec.getInstance()));
    private ex3: Expert =
            new Expert(new Set<Quotes.Bio|Quotes.Chem|Quotes.Mech>(
              Quotes.Chem.getInstance(),
              Quotes.Bio.getInstance(),
              Quotes.Mech.getInstance()));
    private ex4: Expert =
            new Expert(new Set<Quotes.Chem|Quotes.Elec>(
              Quotes.Elec.getInstance(),Quotes.Chem.getInstance()));
    private plant: Plant =
            new Plant(new Set(this.a1), New Map(
                new Maplet(Test1.p1, new Set(this.ex1,this.ex4)),
                new Maplet(Test1.p2, new Set(this.ex2,this.ex3))));

    public Run() : Test1.PE {
        const periods : Set<Token> =
            _.clone(this.plant.ExpertIsOnDuty(this.ex1))
        const expert : Expert =
            this.plant.ExpertToPage(this.a1, Test1.p1)
        return new Test1.PE(periods, expert);
    }
}
namespace Test1 {
    export class PE extends Record {
        constructor(public periods: Set<Token>, public expert: Expert){
            super();
        }
    }
    export const p1: Token = new Token("Monday day");
    export const p2: Token = new Token("Monday night");
    export const p3: Token = new Token("Tuesday day");
    export const p4: Token = new Token("Tuesday night");
}
```