AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

A VDM++ TO RUST CODE GENERATOR FOR OVERTURE

BY

LASSE BRØSTED PEDERSEN

10769

MASTER'S THESIS

IN

COMPUTER ENGINEERING

SUPERVISOR: LECT. STEFAN HALLERSTEDE

CO-SUPERVISOR: MIRAN HASANAGIĆ

Aarhus University School of Engineering

January 4, 2016

Lasse Brøsted Pedersen	Stefan Hallerstede
10769	Supervisor

Abstract

Increasingly complex mission-critical computer-based systems are being developed and utilised for real-time tasks. To ensure correct behaviour of the software for these systems, formal methods such as VDM may be applied during the development process. Using the VDM notation for software specifications, it is possible to automate the process of deriving an implementation of a given specification, resulting in increased productivity and possibly reducing the amount of errors introduced during the process.
This thesis presents rules for automated translation from a large subset of VDM++ to efficient source code in the Rust programming language. The translation rules are described, and implemented in a code generator using the code generation platform of the Overture tool. The translation has been validated by evaluating the translation of a scheduling system for monitoring alarms, and a model of locomotion and gait algorithms for a planetary rover.

Acknowledgements

I would like to thank my academic supervisor, Stefan Hallerstede, for great support, good discussions during the project, and invaluable feedback on the report. In addition I would like to thank my co-supervisor Miran Hasanagić, for support during development of the code generator, as well as feedback and suggestions during the project. I would also like to thank Peter W. V. Tran-Jørgensen for providing general VDM code generation expertise and rapid support when developing for the Overture code generator platform.
Finally, I would like to thank my friends and family for their support.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the background and motivation for the subject of this thesis. Furthermore the hypothesis, goals and scope for the thesis is presented, as well as the approach employed to reach these goals.

1.1. Overview

The evolution of technology has led to increasingly complex systems, many of which are mission critical. The software for these systems requires increasingly complex development which makes it harder to ensure dependability of such systems. Insufficient validation of critical software can result in failures, which could lead to loss of lives or substantial loss of material property.

One way to increase the level of confidence and correctness for software, is to apply formal methods during the development process [1, 2]. This is done by replacing or augmenting the requirement specification with a specification written in a formal notation, such as *Vienna Development Method* (VDM). Doing this enables more rigorous methods for validation and verification (V & V) of system properties, which leads to better and earlier detection of inconsistencies of the system specification.

Developing a software program is the process of analysing, modelling and implementing software, which realises the requirements. In the absence of formal models and related automation tools, this process is carried out manually, which increases the implementation time, and may increase the risk of introducing inconsistencies between specification and system during the development process. This manual process is inefficient, because it requires the functionality and logic of the specification to be reproduced in the concrete system implementation.

The process of turning a specification into a concrete implementation can be automated, when the specification is expressed in a formal notation such as VDM. In this scenario a translation program transforms the source representation of the model into the desired target representation such as a programming language or directly to machine code.

This chapter introduces the background for the subsequent sections of this chapter. Section 1.2 introduces the general subjects, that form the background of this thesis. Section 1.3 presents the motivation for this thesis. Then, section 1.4 presents the overall hypothesis, goals, scope and approach of this thesis. Afterwards, section 1.5 introduces the case study used in this project. Lastly, Sections 1.6 and 1.7 present the reading guide and the thesis structure, respectively.

1.2. Background

This section introduces the *Vienna Development Method* (VDM), translation and generation of code, and the Rust programming language.

1.2.1 Vienna Development Method

The *Vienna Development Method (VDM)*, is a formal method, which allows modelling, specification and evaluation of systems. It was initially conceived at IBM in the 1970s, and has been developed since [3]. At present three dialects of VDM exists: *VDM-SL*, which was made an ISO-standard in 1996 [4, 5], *VDM++* an object-oriented extension of VDM-SL[3] with support for concurrency concepts such as threads and synchronization, and *VDM-RT*, which adds modelling support for real-time systems to VDM++[6]. VDM models can be created and evaluated using either the commercial tool VDMTools [7], or Overture [8], which is open-source tool built on Eclipse.

1.2.2 Translation and Code Generation for VDM

Code translation is the process of transforming one representation of a code to another representation, while preserving the semantics of the original representation. After transformation the new representation may be used to generate source code for a language which is different from the original language. In the context of this thesis code generation refers to a translation from a VDM model to source code expressed in a general programming language.
Code generation for VDM exists in both available tools, VDMTools and Overture, with code generators in various development stages being available with C++ and Java as source languages. The work carried out in this thesis builds on the code generation platform of the Overture tool [9], which supports both the C++ and the Java as target languages through the code generation backend of the Overture tool.

1.2.3 Rust

Rust is a general purpose, multi-paradigm, compiled language, which is being developed as an open-source project sponsored by Mozilla [10]. Rust guarantees memory safety without a garbage collector, as well as freedom from data races, both of which are checked at compile-time by leveraging Rust's type system. Memory safety means that a no memory should become unreachable without being deallocated, and no uninitialised memory should ever be accessible. Rust attempts to achieve this by ensuring that all memory has exactly one owner, which is responsible for deallocation, and that no memory is ever aliased *and* mutable.

1.3. Motivation

The motivation for this project is to increase the degree of automation in the software engineering process by increasing the options available for model based software engineering. The motivation for using Rust as the target language of the translation, is that Rust includes many of the language constructs found in the VDM languages, simplifying the translation by requiring fewer and simpler transformations. In addition, Rust is a restrictive language, enforcing *memory safe* source code which is suitable for mission-critical systems. Furthermore, Rust is capable of being compiled to efficient native code on many processor architectures using the LLVM compiler [11].

In addition to the motivational factors mentioned above, the work carried out in this thesis addresses a specific need, as part of an effort to integrate the Overture Tool with *The ASSERT Set of Tools for Engineering* (TASTE) [12]. TASTE is a toolset for model-centric development, which is being developed by the European Space Agency (ESA)[13].
The personal motivation for the author of this thesis, was the opportunity to work with and explore Rust, a new language with many interesting aspects and possible applications, in the context of translation and code generation, requiring intimate knowledge of the languages involved.

1.4. Thesis Overview

Section 1.4.1 presents the hypothesis, underpinning this thesis. Sections 1.4.2, 1.4.3 and 1.4.4 present the goals, scope and approach of this thesis, respectively.

1.4.1 Hypothesis

The hypothesis for this project is, that there is a subset of VDM++ constructs adequate for non-trivial models, for which a correctness preserving translation to efficient Rust code exists.
In this context, a non-trivial model has the following characteristics:

- Has multiple classes or modules, each comprising multiple multi-statement/expression functions/operations.
- Utilizes the collections of VDM (Sequences, Sets, Mappings), and their associated operators, as well as comprehensions.
- Uses control flow constructs such as loops, `if`-statements and the `match` statement.

1.4.2 Goals

The hypothesis will be tested by attempting to achieve the goals of this thesis, listed below.

Goal 1: Propose translations from VDM++ to efficient[1] Rust.

Goal 2: Develop a proof-of-concept VDM++ to Rust code generator based on the code generator platform in the Overture platform. This code generator must be based on the findings from Goal 1.

Goal 3: The translation produced by the prototype must be validated by evaluating case studies.

Goal 4: Improve personal problem solving skills with regards to identifying, analysing and describing problems and their impact, and reviewing possible solutions.

Goals 1 through 3 are steps toward validating the hypothesis, and hence these are directly related to the hypothesis. Goal 4 is a personal goal, thus unrelated to the hypothesis.

[1] In the context of this thesis, *efficiency* refers to the memory space requirements and speed of execution.

1.4.3 Scope

The scope of this thesis is to make and describe a proof-of-concept VDM++ to Rust code generator, which is able to translate a subset of VDM++ to correct Rust code. The translatable subset must be non-trivial as presented in the hypothesis in Section 1.4.1. A full translation is not inside the scope of this thesis; in particular, constructs which are inherently inefficient are not translated.

1.4.4 Approach

The applied approach can be described as an iterative process which was carried out for each language construct of the the source language (VDM++). Initially, several small VDM models were translated manually, as part of the effort to analyse the semantic and syntactic differences between VDM and Rust, and find possible mappings for the language constructs involved. The translation was implemented in the Overture code generator, and the output tested. Larger VDM models, or models featuring previously unmapped constructs were then considered.
The approach can be described by the four steps below:

Analysis: Analysis of the languages involved in the process of code generation. Acquiring the necessary understanding of the semantics of the constructs to be mapped, and possible mapping targets in the target language in order to propose a design solution.

Design: Propose a transformation for the constructs, which enables code generation to the target language.

Implementation: Implement the proposed transformation and code generation logic. Implementation of the translation may include transformation of the Intermediate Representation (IR), introduction of new IR nodes, and the creation and alteration of templates for generating the target source code.

Evaluation: The translation of each construct was validated using small sample models.

As the translatable subset grew to a sufficient size, larger case-studies were translated to ensure that the individual translations integrated to correct Rust source code. The case studies used for this thesis are described in Section 1.5.

1.5. Case Studies

The two case studies treated in this thesis are presented in this section. The case studies have used to validate the translations proposed, and the corresponding implementation in the code generator. The case studies documents the implemented translations, and shows them in the context of a full VDM model. In addition, the case studies have illuminated incomplete areas of the translation, and weaknesses of the Overture code generation platform.

1.5.1 The Alarm Model

The alarm model is a VDM++ model adapted from the book *"Validated Designs for Object-oriented Systems"* [3, pp. 20-38]. This model is ideal for the initial case study because it is small, but still contains a large amount of VDM constructs, thus ensuring that the translation is able to handle most basic constructs of VDM. The case study on the alarm model is presented in detail in Chapter 5.

1.5.2 The Rover Model

The Rover model is a larger VDM-RT model developed to model the control system for the six drivetrains of an planetary rover vehicle. It models the drivetrains, as well as a selection of manoeuvres supported by the rover. This case study shows how larger models can be translated, and how to structure a model in order to facilitate translation to efficient Rust. The rover model is presented in detail in Chapter 7. The model was built by ESA-ESTEC as part of an *Industry Follow Group* challenge [14, 15].

1.6. Reading Guide

The conventions used in this thesis are presented in this section.

Emphasis
Words that are emphasised are written in *italic*, such as *emphasised*.

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

Keywords in VDM and Rust
Keywords in VDM and Rust are written in **`boldface typewriter font`**. An example is "**`struct`**".

Element of a VDM models or Rust
Elements which are part of VDM models or Rust code are written in a `typewriter font`, such as names of classes or functions. An example is the name of a class, such as `ClassName`. A function or operation is indicated when the name ends with (), independently of whether or not it takes parameters. An example is the operation `setBalance()`.

VDM models and Rust listings
Listings of VDM models and Rust code are presented as shown in listing 1.1 and 1.2, respectively.

```
-- A VDM comment
public createSet: () ==> set of int
createSet() == return {1,2,3};
```

Listing 1.1: Example of a VDM code listing.

```
// A Rust comment
pub fn createSet(&mut self) -> Set<i64> {
  return set!{1, 2, 3};
}
```

Listing 1.2: Example of a Rust code listing.

1.7. Structure of Thesis

The suggested reading of this thesis is shown in Figure 1.1. In the figure every chapter in the thesis is represented by a solid box. Furthermore, the arrows between the boxes indicate the dependencies between the chapters. The main themes of the specific chapter(s) are indicated by the horizontal lines and the text to the left in Figure 1.1.

Chapter 1: Describes the background and context of this thesis. In addition, Chapter 1 describes goals of this thesis, and the approach to achieve and evaluate them.

Chapter 2: Describes the basic concepts of the Rust programming language, enabling the reader to understand the work carried out in this thesis.

Chapter 3: Introduces the code generation platform of the Overture tool and describes the steps necessary to add and alter translations.

Chapter 4: Presents and explains design decisions made for translating VDM constructs to Rust. In addition, this chapter provides guidelines for writing translatable VDM and includes small examples showing translation of each construct.

Chapter 5: Presents and explains the translation of the alarm model. Furthermore, this chapter discusses resolved and unresolved issues that arose during translation of the model.

Chapter 6: This chapter discusses and explains the translation of specification constructs; pre- and postconditions and invariants.

Chapter 7: Presents and explains the translation of the Rover model. Furthermore, this chapter discusses resolved and unresolved issues that arose during translation of model. This chapter also treats structuring of VDM models to facilitate translation to efficient Rust.

Chapter 8: This chapter discusses the different approaches to concurrency employed by Rust and VDM++, as well as possible solutions for translation of basic concurrency constructs.

Chapter 9: This chapter concludes the work conducted in this thesis, and the goals presented in section 1.4.2 are evaluated. Possible directions for future work are discussed followed by the final remarks.

Appendix A: This appendix is the Rust code of the Rover, obtained by automatic translation of the VDM++ model of the Rover.

The Master's thesis report, the developed code generator, and code and model for case studies are also enclosed on the CD.

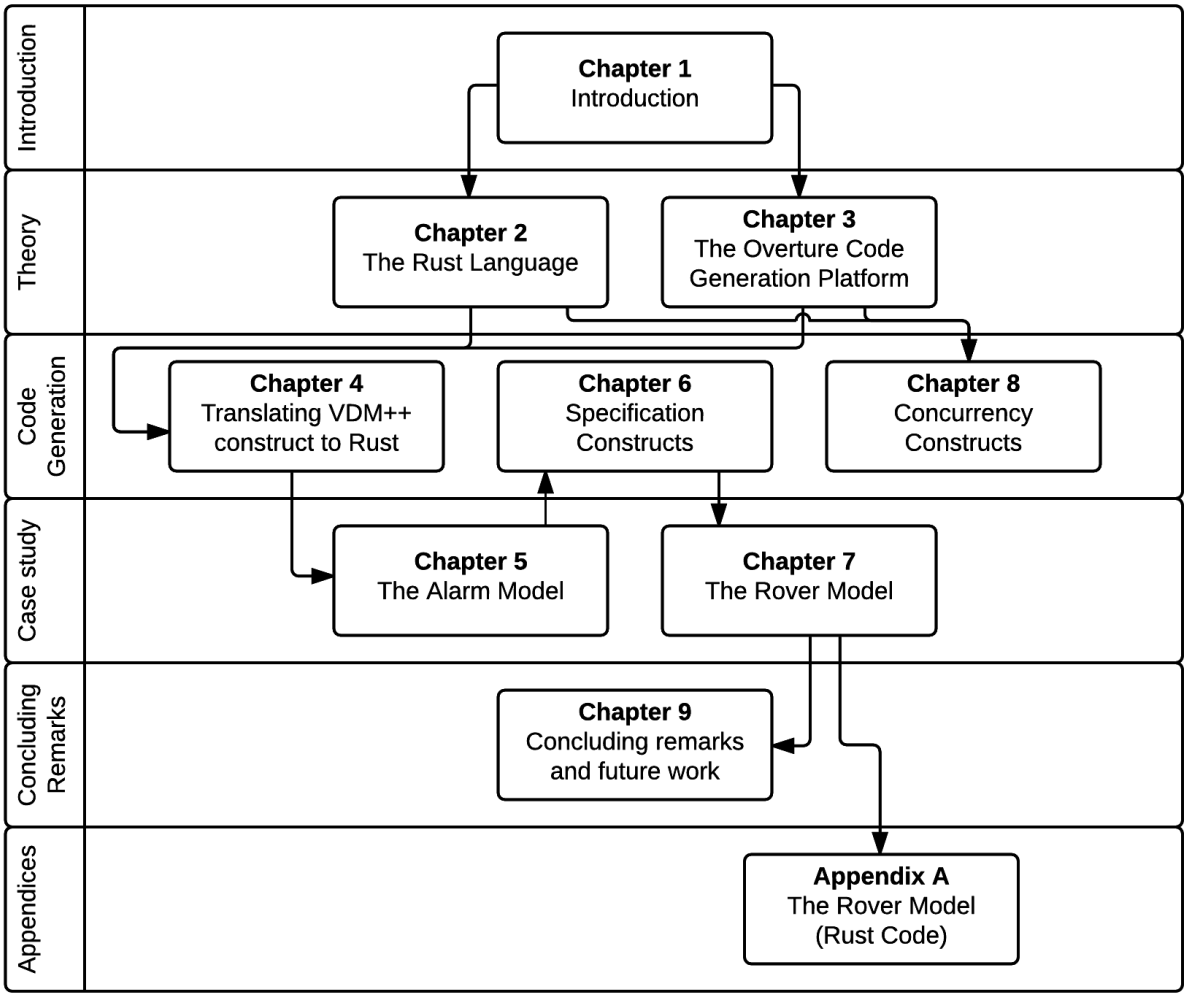

Figure 1.1: Structure of the thesis.

Chapter 2

The Rust Language

Rust is the target language of the translations proposed in this work. This chapter introduces the Rust programming language and the basic concepts of the language.

2.1. Introduction

Rust is a general purpose, multi-paradigm, compiled language, which is being developed as an open-source project sponsored by Mozilla. The Rust developers describe Rust as follows[10]: *"Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety."*. Rust is a new programming language, which was recently released in the first stable version 1.0 on May 15, 2015. Rust is syntactically in the C-family, but semantically the language is influenced by a wide range of programming languages [16]. Rust includes features like algebraic data types, pattern matching and type inference from SML and OCaml, and the *Resource Acquisition Is Initialisation* idiom (RAII) [17], move semantics and the memory model from C++. Rust's type system makes heavy use of *traits* [18], which are similar to the typeclasses of Haskell [19].
The defining feature of Rust is the memory safety guarantees provided at compile-time by a *lifetime*-system based on the work on *regions* by Tofte and Talpin [20, 21]. Rust guarantees that the following memory safety errors never occur:

- Dereferencing null or dangling pointer
- Use of uninitialised memory
- Breaking the pointer aliasing rules, as explained in Section 2.2.
- Producing invalid values of primitive types
- Causing a data race
- Unwinding the stack into another language

The guarantees provided by Rust have not been formally proved, but work have been done prove the soundness of the language [22].
The examples used in this chapter have been adapted from the Rust book[23].

2.2. Ensuring Memory Safety

The main goal of Rust is achieving memory safety, which Rust does using a concept which enforces data ownership. Through the type system the compiler is able to verify - at compile time, that there is *exactly one* binding for each resource. When the owner of a resource goes out of scope, the resource is deallocated. In other words; the owner is *solely* responsible for deallocating owned resources and performing associated clean up. Example:

```
1 let v = vec![1, 2, 3];
2
3 let v2 = v;
4
5 println!("v[0] is: {}", v[0]);
```

Listing 2.1: Rust move sematics.

The code in listing 2.1 will produce a compiler error. Examining the steps; in line 1, a vector is created and bound to `v`, which becomes the owner. Line 3 then transfers ownership of the vector from `v` to `v2`, thus invalidating `v`; in Rust parlance: *"v was moved to v2"*. Hence, when attempting to access the first element of `v` through the indexing operator in line 5, the compiler will issue an error and halt compilation. The reason `v` is invalidated in line 3, is that the vector's data is allocated on the heap, and when `v` is moved to `v2`, the data pointer is copied to `v2`. Keeping `v` valid would allow aliasing. In listing 2.2, the vector is passed to a function. Here, the compiler issues the same error.

```
1 fn take(v: Vec<i32>) {
2     // what happens here isn't important.
3 }
4
5 let v = vec![1, 2, 3];
6
7 take(v);
8
9 println!("v[0] is: {}", v[0]);
```

Listing 2.2: Passing resource ownership to a function.

The ownership system alone, would require deep-copying data or passing ownership back and forth, which would be tedious and inefficient. For this reason Rust introduces *borrowing*. The error from listing 2.2 can be fixed by passing a reference to the vector to the function instead. The added `&` to the type of the formal parameter, and when invoking the function signifies a reference.

```
1 fn take(v: &Vec<i32>) {
2     // <implementation of take()>
3 }
4
5 let v = vec![1, 2, 3];
6
7 take(&v);
8
9 println!("v[0] is: {}", v[0]);
```

Listing 2.3: Passing resources by reference.

A reference, like the one passed to the `take` function in listing 2.3, does not allow mutation, that is making changes to the referenced value. The type system allows for multiple immutable references(or borrows) at the same time. Mutating a resource requires a *mutable reference*, denoted **`&mut T`**. Exactly one mutable reference may be alive, but not when the resource is already immutably borrowed. Listing 2.4 is an example of passing a mutable borrow:

```
1 fn take(v: &mut Vec<i32>) {
2     v.push(1337);
3 }
4
5 let v = vec![1, 2, 3];
6
7 take(&mut v);
8
9 println!("v[0] is: {}", v[0]);
```

Listing 2.4: Passing a mutable reference.

2.3. Concurrency

In addition to memory safety, Rust also removes the possibility of data races, as described by the Rust documentation [10]. Rust does this by using the type system - specifically Rust relies on the marker traits `Send` and `Sync` to enable reasoning about concurrency properties of data types.

2.3.1 Send and Sync

Types implementing the `Send` trait must be able to have ownership safely transferred between threads. Notably, the non-atomic reference counting construct `Rc<T>`, does not implement `Send`. `Send` is required for types being sent through a channel to another thread, or being passed to a new thread when spawned.
The `Sync` trait may be implemented by types, for which `&T` is safe to use from multiple threads. Types implementing `Sync` includes primitive types, collection types, and types which only holds member data implementing `Sync`. In addition synchronisation primitives such as `Mutex<T>` implement `Sync` as well.

2.3.2 Sharing data

As part of the standard library, Rust provides multiple concurrency primitives. Sharing mutable data can be done using `Mutex<T>`. In Rust, the protected data is wrapped by the mutex, making the data inaccessible without having the lock. Example given:

```
1 let data = Arc::new(Mutex::new(vec![1, 2, 3]));
2
3 for i in 0..3 {
4   // bump reference count
5   let data = data.clone();
6   thread::spawn(move || {
7     // acquire mutex lock
8     let mut data = data.lock().unwrap();
```

```
9       data[i] += 1;
10      // mutex unlocks and refCount is decreased here
11    });
12  }
```

Listing 2.5: Using the Rust mutex.

In the example in listing 2.5, line 1 shows the data being instantiated, wrapped by a mutex, which itself is being contained by an atomic reference counted wrapper. Since each thread requires access to the data-holding mutex, the `clone` method is called to increase the reference count, allowing the data to be moved into the thread. Each thread then acquires the lock, and modifies the data through it. Rust relies on the RAII idiom (Resource Acquisition is Instantiation) to unlock the mutex, and decrease the resource reference count when the lock and reference counter leaves the scope.

2.4. Complex data structures

To support creation of complex data structures, Rust provides three ways to compose data: tuples, structs and enum. However, Rust does not provide classes.

2.4.1 Tuples

The simplest way to create a complex data type is by using tuples. A tuple is an ordered, fixed sized list, which allows heterogeneous data types, as is familiar from many functional programming languages. Members of tuples can be accessed by indexing and pattern matching. The empty tuple `()` is called unit. Listing 2.6 is an example of tuple instantiation and usage.

```
let tuple = (1, 2, 3);

let x = tuple.0;        // Indexing
let (_, y, z) = tuple;  // Pattern matching

println!("tuple holds {}, {}, {}", x, y, z);
```

Listing 2.6: Tuples in Rust.

2.4.2 Structs

Rust also provides the **`struct`** construct for creating new data types. Structs come in multiple flavors. The following are of interest for this thesis: record-like and unit-like. Unit-like structs are used for creating distinct types, which do not define any member variables.

```
// record-like
struct Point {
    x: i32,
    y: i32,
}

// unit-like
```

```
struct Atom;

// instantiation
let origin = Point{ x: 0, y: 0 };
let atom = Atom;
```

Listing 2.7: Structs in Rust.

2.4.3 Enums

The **enum** type in Rust is an implementation of algebraic data types. An enum is a single type, which can represent multiple variants. Variants can be declared much like structs:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
}

// instantiations
let q: Message = Message::Quit;
let c: Message = Message::ChangeColor(0, 0, 0);
let m: Message = Message::Move { x: 3, y: 4 };
```

Listing 2.8: Declaring enum variants.

Handling enum variants is typically done through the **match** expression, which relies on pattern matching to determine the variant of the enum.

2.5. Functions, Methods and Traits

Rust supports multiple ways of defining functionality: free standing functions, methods or functions associated to a **struct** or **enum**, and **traits** which can be implemented for any type.

2.5.1 Methods and associated Functions

The following is an example of a type with a method (`area()`) and an associated function, `new`, which by convention is the constructor.

```
1  struct Circle {
2      x: f64,
3      y: f64,
4      radius: f64,
5  }
6
7  impl Circle {
8      fn new(x: f64, y: f64, radius: f64) -> Circle {
9          Circle { x: x, y: y, radius: radius }
10     }
11
12     fn area(&self) -> f64 {
```

```
let k: i64 = 123;

// types of f and closure are annotated
let mut f: Fn(i64) -> bool =  |i: i64| -> bool { i == k};

// types are inferred
f = |i| i == k;
```

Listing 2.11: Closures in Rust.

```
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
    println!("{}", c.area());
}
```

Listing 2.9: Methods and associated functions.

In Rust, method call syntax is shorthand for *Universal Function Call Syntax (UFCS)*. Listing 2.10 shows an example calling the `area()` function of the `Circle` type using UFCS.

```
...

let c = Circle::new(0.0, 0.0, 2.0);
println!("{}", Circle::area(&c));
```

Listing 2.10: Using universal function call syntax to call `area()`.

Rust also support closures. Closures implement the **`traits`** `Fn` and `FnMut` depending on whether they mutate the environment they close over. The signature of a closure can be defined using type annotations, but if the type can be inferred, the type annotations are optional. An example using closure can be seen in listing 2.11.

2.5.2 Traits

The trait system is a mechanism for specifying required and provided functionality. A trait is a collection of method signatures and overridable default methods. The example in listing 2.9 may be reorganized by use of a trait like shown in Listing 2.12.

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Listing 2.12: Defining and implementing a trait.

Through traits Rust provides both compile-time and runtime polymorphism, also referred to as static and dynamic dispatch, respectively. Static dispatch relies on generics and trait bounds, whereas dynamic dispatch utilizes *trait objects* - that is, a combination of a data pointer and a virtual call table (vtable).

Complex data structures may also use generics with trait bounds.

2.6. Macros

The abstractions already presented, covers many scenarios, but in some cases, *macros* are needed. Unlike text manipulation macros offered by the C preprocessor [24], the macro system in Rust is hygienic, and works by manipulating the syntax tree early in the compilation process. Macros in Rust are based on pattern matching. An example is given in Listing 2.13.

```
1  macro_rules! map {
2    () => ( Map::new() );
3      ($( $key: expr => $val: expr ),*) => {{
4           let mut map = Map::new();
5           $( map.insert($key, $val); )*
6           map
7      }}
8  }
9
10 fn main() {
11     let m: Map<char,u64> = map!('a' => 1, 'b' => 2, 'c' => 3);
12 }
```

Listing 2.13: Pattern matching in Rust macros.

Pattern matching during expansion of the macro `foo!` causes the second clause to be selected, hence the macro invocation in line 11 of Listing 2.13 causes the code to expand as shown in Listing 2.14.

```
1  fn main() {
2      let m: Map<char,u64> = {
3      let mut map = Map::new();
4           map.insert('a', 1);
5           map.insert('b', 2);
6           map.insert('c', 3);
7           map
8      };
9  }
```

Listing 2.14: Expanded `map!`-macro.

2.7. Unsafe

The Rust compiler is very conservative by design; it will only allow programs, that can be proved by the compiler to be safe. In some situations, the compiler cannot prove that a program is safe, even though it is. For these situations, the language provides the *unsafe* feature. Functions can be marked unsafe, which is a requirement for external functions, such as functions defined in C, which is inherently unsafe. In addition, traits can be marked unsafe, which means implementing such a trait is unsafe. An example of an unsafe trait is the `Sync` trait described in Section 8. In unsafe Rust it is possible to the following:

- Dereference raw pointers
- Call functions marked as unsafe
- Implement unsafe traits
- Mutate static variables

When using unsafe code, it is the responsibility of the developer to avoid unsafe behaviour, as the compiler does not provide the normal guarantees. Unsafe code can be wrapped in safe code, to provide safe abstractions by building on code that is inherently unsafe.

Chapter 3

The Overture Code Generation Platform

This chapter describes the code generation platform used to implement the transformations and code generation described in Chapter 4.

3.1. Introduction

The *Overture tool* [8] is an integrated development environment for developing and analysing VDM models, which includes a code generation platform, and supports translation from VDM to Java [25, 9], as well as to C++ [9]. As mentioned in the Chapter 1, the patterns for translation and code generation developed for this thesis will be implemented using the code generation platform of the Overture tool.

This Chapter will first provide an overview of the Overture code generation process in section 3.2. Section 3.3 then describes the code representation used as input for the code generation process. Section 3.4 finally presents the components of the code generator.

3.2. Overview of the Code Generation Process

Figure 3.1 presents an overview of steps in the process which generates Rust code from a VDM model.

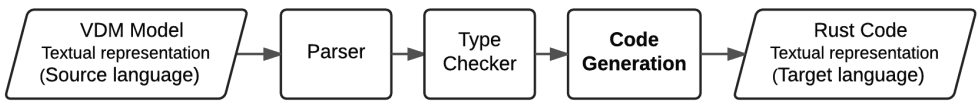

Figure 3.1: Overview of the process of code generation of VDM to Rust.

The *parser* and *type checker* components of Overture comprise the foundation for all operations and analysis of VDM models. The parser transforms the textual representation of a VDM model into an *Abstract Syntax Tree* (AST), which is then validated by the type checker. From the valid AST, a common *Intermediate Representation* (IR) is generated, subject to transformations - both

common and Rust-specific, resulting in an IR closely resembling the target language. Finally, the resulting IR is passed to a template engine, generating the final Rust code.

3.3. Representing a VDM Model

The Overture tool uses an *Abstract Syntax Tree* (AST) which is isomorphic with the VDM syntax to represent VDM models [26]. In effect, each syntax construct of the VDM languages can be represented as a type of node in the AST. Instances of the AST is created using the parser and type checker components of the Overture tool. Each node has exactly one parent, and keeps track of child nodes, allowing for bi-directional traversal of the tree. The AST can be analysed and transformed by applying the visitor pattern [27].

The Java code implementing the AST nodes and associated visitor classes is generated by an AST generator tool from a textual description of the AST nodes, as shown on Figure 3.2.

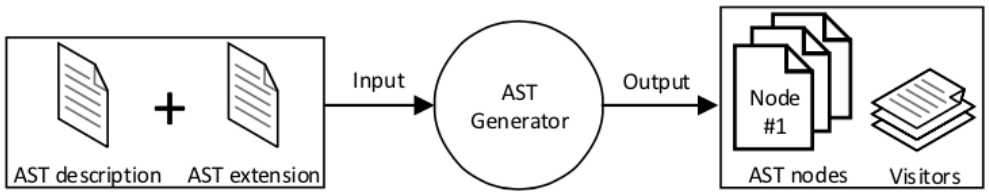

Figure 3.2: The generation of AST nodes and associated visitor classes by the AST generator. The Figure is adapted from [9].

3.4. Components of the Code Generation Platform

Figure 3.1 shows an overview of the process and the main components of the code generation platform. The goal of the process carried out by the code generation platform is to transform the AST representation of the VDM model in to a representation, that matches the target language while preserving the semantics of the original model.

3.4.1 The Intermediate Representation

Since the Overture AST is isomorphic with the VDM language, it is not suitable as a representation for any target language, which is not a subset of the VDM language. For this purpose, the code generation platform defines an *Intermediate Representation* (IR), which is a semantic preserving simplification of the AST. The IR of a given VDM model is created by applying visitors to the original AST, resulting in a transformation from AST to IR. Since the IR is seperate from the AST, it can be extended with nodes corresponding to constructs of the target language. The implementation of the IR is generated using the same tool generating the AST, as described in Section 3.3.

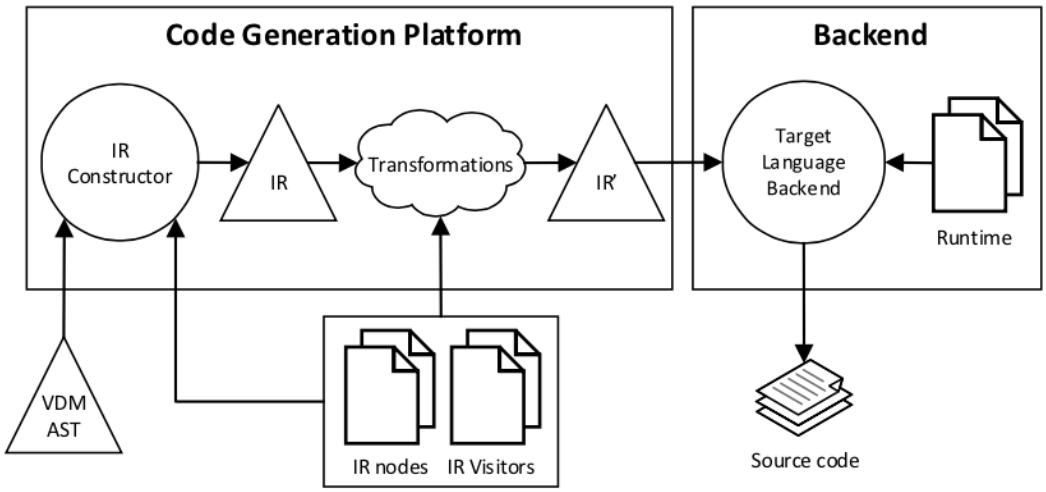

Figure 3.3: Overview of the code generation platform of the Overture tool. The Figure is adapted from [9].

Transforming the Intermediate Representation

The translation from the source language, VDM, to the target language, Rust, is carried out by visitors applying transformations to the IR, rearranging, altering, replacing and removing nodes, resulting in a transformed intermediate representation, IR', as shown in figure 3.1. The code generation platform includes a number of common transformations, utilized by both the Java and Rust code generator. Additionally, the Rust code generator defines Rust specific transformations.

An example is the transformation handling identifier variable expressions for instance variables; VDM allows accessing instance variables implicitly using an identifier, whilst Rust requires explicitly using **`self`**-keyword to indicate access to an instance variable.

```
1  instance variables
2    instance_val: int;
3
4  operations
5    -- operation get_val() before transformation.
6    public get_val: () ==> int
7    get_val() == return instance_val;
8
9    -- operation get_val() after transformation.
10   public get_val: () ==> int
11   get_val() == return self.instance_val;
```

Listing 3.1: Transform access to instance variables to be explicit.

Listing 3.1 shows the effect of a transformation, targeting nodes for identifier variable expressions. If the expression accesses instance variables, the node is replaced with a field expression node, where **`self`** is the object expression, and the member name is the identifier from the original node.

3.4.2 Generating Source Code for the Target Language

When the IR has been transformed to the final representation, IR', as explained Section 3.4.1, the IR' is used as input for the process generating the code for the target language. This process is performed using the template engine Apache Velocity [28]. Each node type is associated with a template defining formatting for nodes of that type.

```
#set( $returnType = $RustFormat.format($node.getType().getResult()))
#set( $params = $RustFormat.formatList($node.getParams()))
#set( $expr = $RustFormat.format($node.getExp()))
|$params| -> $returnType { $expr }
```

Listing 3.2: Apache Velocity template for the lambda expression node.

Listing 3.2 shows the template for the lambda expression node. The first three lines are directives for the template engine to create and initialise the variables `$returnType`, `$params` and `$expr`. The final line of the template contains the output to be rendered. The calls to `$RustFormat.format()` and `$RustFormat.formatList()` invoke the template engine on the node(s) passed as arguments, thus allowing each template to control which child nodes to render, and where to insert the rendered output.
A lambda expression with a formal parameter, `n`, of type **`nat`**, returning a value of type **`nat`** from the expression `'n + 2'` will render as shown in Listing 3.3.

```
|n: u64| -> u64 { n + 2 }
```

Listing 3.3: Rendering of a lambda expression.

3.4.3 Introducing Support for Constructs of the Target Language

When defining translations for VDM constructs, it is sometimes necessary to use constructs in the target language, which are not defined by the current IR nodes. When using the Overture code generation platform two options exists; extending an existing template and/or introducing new IR node types.

Extending an existing template

In some cases, the new construct can be implemented as part of a template for a pre-existing IR node, eliminating the need to construct the necessary IR through transformations. A consequence of this solution is that the introduced construct is not represented by the IR, which means it is unavailable for analysis. This solution has been used extensively for implementing **`trait`**s in the generated Rust code. When using this solution with Rust as the target language, it is feasible to utilise the macro feature of Rust intended for this purpose. Using Rust macros to delegate the template logic to Rust code, which is directly testable, allows for concise logic and reduces the verbosity of the final Rust code.

Extending the IR

When the existing IR provides no way of representing the information needed for further analysis and/or code generation, the solution is to introduce a new node type to the IR, or extend an existing IR node with additional information.

```
...
|       {UnionEnum}
        [name]:java_String
        [variants]:CG.#decl.namedType*
...
```

Listing 3.4: Union enumeration declaration as IR extension.

Listing 3.4 shows the definition of a declaration, used as the mapping target for VDM union type definitions. This particular node declaration will result in a IR node with a `name` property of type `String`, and `variants` property holding a list of `namedType` declarations. The node declaration in Listing 3.4 is part of the description serving as input for the AST generator tool creating the implementation of the IR, using the same method applied when creating the AST as described in Section 3.3.

3.5. Overview of transformations in the Rust code generator

The following lists the transformations developed for the Rust code generator:

Borrowing: Separates evaluation of operation arguments from the operation invocation to satisfy the Rust borrow checker, as described in section 5.4.1.

Constructs with set bindings: Transforms **`set`**-, **`seq`**- and **`map`**-comprehensions, quantified statements as well as **`let be .. st`** statements and expressions to method calls utilizing lambda expressions.

Constructor: Transforms class constructor operations to explicitly create a value of the **`struct`** implementing the VDM **`class`**.

Explicit function and operation invocations: Transforms method calls on the current instance to properly call using **`self`**, eg. **`self`**`.op()`. Also transform function and static operation calls to explicitly call using the defining class type, eg. `classT::func()`.

Preconditions: Transform IR to generate precondition functions or operations, and insert checks of preconditions.

Type conversion: Insert type conversion expressions for union types, optional types and numeric types based on declared types.

Union types: Transform declarations of named union types into a corresponding declaration for an enumerated type.

Value semantics: Insert `clone()` calls to preserve value semantics.

Operations on collections: Transform operations on VDM collections into method calls.

In addition, templates for the set of all possible node types in the final intermediate representation was developed.

Chapter 4

Translating VDM++ Constructs to Rust

This chapter lists the translations used to create Rust source code from core VDM constructs. The translations are presented primarily by way of examples to be intuitively understandable. The presented examples are straight forward to generalise into concrete translation rules. Specification constructs are treated in Chapter 6 and concurrency constructs are treated in Chapter 8.

4.1. General

4.1.1 The Runtime

In order to support the generated code, a library has been created, which includes common functionality found in VDM models. This includes the collections; **`seq`**, **`set`**, **`map`**, and the **`token`** type. These collections are built on top of their Rust equivalents, **`Vec<T>`**, **`HashSet<T>`** and **`HashMap<K,V>`**, respectively, and implements the operators available in VDM as methods. This library is designed to be distributed along with the generated source code, either directly, or using the Rust package manager, Cargo [29].

4.1.2 Structure

VDM++ classes when translated, are placed in a Rust module each, along with associated type definitions, and values. These modules then comprise a Rust crate, which correspond to a VDM++ model. A crate root module (`main.rs`) is also created, which is responsible for loading external crates, and declaring the modules of the crate.

4.1.3 Type Requirements

Every type must implement the Rust traits `PartialEq, Eq, Hash, Clone` along with the formatting trait `Debug`. This ensures, that the types can be used in the set and map types, compared for equality and be printed. The `Clone` trait is required in order to be able to translate value semantics and evaluation of post conditions correctly. Most built-in types implement these, and in most cases these traits can be automatically derived by the compiler - a notable exception being the VDM *equality abstraction* for structs described in section 4.3.3. Listing 4.1 shows auto derivation of traits in Rust.

```
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct Plant {
  alarms: Set<Alarm>,
  schedule: Map<Period, Set<Expert>>
}
```

Listing 4.1: Automatic derivation of Rust traits.

4.2. Basic Data types

This section describes how the basic data types of VDM are translated to semantically equivalent Rust.

4.2.1 The Boolean type

The boolean type of VDM, **`bool`** maps directly to the corresponding Rust type of the same name.

4.2.2 The Character type

The character type, **`char`**, is mapped to the Rust character of the same name. However, this is not a direct mapping, because the VDM character type represents a *Unicode Codepoint* excluding surrogates, while the Rust character type represents a *Unicode Scalar Value*. This leads to inconsistencies, as some code points is being represented by two **`char`**s in VDM, and one in Rust.

4.2.3 The Numeric types

The VDM numeric types are mapped to Rust types as shown in table 4.1.

VDM	Rust
real,rat	F64
int	i64
nat,nat1	u64

Table 4.1: Mapping of numeric VDM types

In Rust, unlike in VDM, the numeric types are disjoint and conversion requires explicit casting. Rust does not perform any type coercion between numeric types, which means, that all operands of any arithmetic operation, must have identical types. In addition the VDM language manual[30] only specifies lower bound for the values of **`nat`**/**`nat1`**, and no bounds for the other numeric types. Furthermore, the required precision of the **`real`**/**`rat`** types are left unspecified. The corresponding Rust types limits the bounds and precision of the represented values. Hence, the Rust types are a subset of the VDM types. Note that **`real`**/**`rat`** and **`nat`**/**`nat1`** are mapped to the same types, respectively. In order to properly enforce **`nat1`** invariant checks would have to be added to the generated code, as discussed in Section 6.6.2.

Mapping floating point numbers

The **real** and **rat** types are not mapped to the 64-bit wide Rust floating primitive, `f64`. Instead the wrapper `F64` is utilized. This is necessary, because the primitive type `f64` omits implementations for the **traits** `Eq` and `Hash`, which implement equivalence relations and computation of hash-values, respectively. This restriction is in place for good reason [31], but since VDM allows floating point numbers in sets and maps, the code generation runtime for Rust provides a wrapper implementing the relevant traits. The current implementation implements `Eq` using exact equality, and `Hash` by interpreting the bit-pattern of the floating point value as an unsigned value, using that as the base for the hash-value evaluation.

4.2.4 The Quote type

In VDM, each quote literal is a disjoint type from other quote literals. In Rust this concept is translated by creating a *unit-like struct* for each literal, implying that each quote type must be declared before instances can be instantiated. An example declaring and using the quote literal `<Bio>` is shown in Listing 4.2.

```
// declaration of literal type omitting trait implementations.
pub struct Bio;

// instantiation and use.
let b1: Bio = Bio;
let b2: Bio = Bio;
assert_eq!(b1, b2);
```

Listing 4.2: Quote literal in Rust

4.2.5 The Token type

The token type represents a countably infinite set of distinct values. A token is constructed from an arbitrary expression, and only allows equality testing, based on the expression the token was instantiated with. In Rust, the token type is part of the runtime, described in section 4.1.1. The Rust token-type represents expressions using their string representation, thus the equality of two tokens are determined by the string representation of the contained expressions.

```
// instantiation of a token in Rust
let tok: Token = Token::new(&set!(1,2,3));
```

Listing 4.3: Token literal in Rust

4.3. Compound Data types

This section describes translation of the complex data types defined by VDM.

4.3.1 The Collections

The VDM collections **seq**, **set** and **map**, have been implemented in the runtime as Rust structs, as mentioned in section 4.1.1. All valid VDM operations on the collection types have been implemented as methods on the corresponding structs.
The specialized types **seq1** and **inmap** are translated into the corresponding general type. The specialized types may be implemented using invariants, discussed in Section 6.6.2.

Examples

```
let seq: Seq<u64> = seq![1,2,3];
let tail: Seq<u64> = seq.tail(); //[2,3]
let e1: u64 = tail.get(1); //2
```

Listing 4.4: Sequence in Rust

```
let s1: Set<u64> = set!{1,2,3,3};   //{1,2,3}
let s2: Set<u64> = set!{3,4};
let u: Set<u64> = s1.union(s2);   //{1,2,3,4}
```

Listing 4.5: Set in Rust

```
let m1: Map<Seq<char>, Seq<char>> = map!{
  strseq!("foo") => strseq!("bar"),
  strseq!("bar") => strseq!("baz")
};

let m2: Map<u64, Seq<char>> = map!{
  1 => strseq!("foo"),
  2 => strseq!("bar")
};

let exp_result: Map<u64, Seq<char>> = map!{
  1 => strseq!("bar"),
  2 => strseq!("baz")
};

assert_eq!(exp_result, m1.compose(m2));
```

Listing 4.6: Map in Rust

4.3.2 Product types

The tuple types provided by VDM and Rust have equivalent semantics and operations, including pattern matching options.

Example

The following is an example of using a tuple to represent a point in VDM.

```
-- type declaration
point = int * int;

-- instantiation and use
let p: point = mk_(1,2),
  x = p.#1 in
(
  cases p:
    mk_(0,0) -> IO`println("at origin"),
    mk_(-,y) -> IO`println(y)
  end
);
```

Listing 4.7: Point tuple definition in VDM

Listing 4.8 is the output of the translation of Listing 4.7 to Rust.

```
// type declaration
type point = (i64, i64);

// instantiation and use
let p: point = (1, 2);
let x = p.0;

match p {
  (0, 0) => IO::println("at origin"),
  (_,y) => IO::println(y)
}
```

Listing 4.8: Point tuple definition in Rust

4.3.3 Composite types

Composite types in VDM, also called record types, correspond directly to the struct type of Rust. The VDM notation specifies an *equality abstraction* field, which denotes a field, that is ignored during equality evaluation. This construct requires the `PartialEq, Eq` traits in the Rust output to be explicitly defined, instead of automatically derived by the compiler, which is possible without *equality abstraction* as shown on line 1 of Listing 4.10. It is not implemented in the work for this thesis. Although the VDM language allows record types with unnamed fields, the translation requires fields to be named.

Example

The VDM sample in Listing 4.9 shows declaration of a record type, as well as instantiation, field reference and update syntax.

```
Alarm ::  descr : seq of char
          level : int;
...

let al = mk_Alarm("Fire!", 9000),
  level = al.level in
  mu (al, level |-> level + 1)
```

Listing 4.9: Alarm record definition and use in VDM.

Listing 4.10 shows the Rust translation of the VDM model in Listing 4.9. The macro `impl_record!` expands to an implementation of a trivial constructor function called `new()` as per Rust convention.

```
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
struct Alarm {
  pub descr: Seq<char>,
  pub level: i64
}
impl_record! { Alarm:
  descr as Seq<char>,
  level as i64
}

...

let al: Alarm = Alarm::new(strseq!("Fire!"), 9000);
let level: i64 = al.level;
Alarm { level: level, ... al }
```

Listing 4.10: Alarm record definition and use in Rust.

4.3.4 Class types

The VDM++ dialect contains a class-construct, which allows for various OO-patterns to be implemented in VDM. In VDM classes allow inheritance, including multiple inheritance and polymorphism. Rust does not have the concept of classes as explained in Chapter 2, hence there is no *direct* mapping. However, it is possible to define methods and associated functions for Rust structs, which is a direct mapping to a VDM class with no super classes. Thus, if we limit VDM models from using inheritance, a direct translation to Rust can be made. Implementing inheritance in Rust could require heavy transformations of the original VDM model, and should rely on Rust **`trait`**s to support polymorphism.
In VDM classes may have associated type definitions and constant values in the **`types`** and **`values`** sections, respectively. In Rust, structs only hold instance variables, hence type and constant values definitions are moved to the enclosing module.

Example

Listing 4.11 shows a an example simple class with an operation and a function.

```
class Plant

values
experts: set of Expert = {...};

types
public Period = token;

instance variables
```

```
alarms   : set of Alarm := {};
schedule : map Period to set of Expert;

functions
PlantInv: set of Alarm * map Period to set of Expert +>
          bool
PlantInv(als,sch) ==
  ...

operations
public ExpertToPage: Alarm * Period ==> Expert
ExpertToPage(a, p) == ...;

public Plant: map Period to set of Expert ==> Plant
Plant(sch) ==
(
  schedule := sch
)
end Plant
```

Listing 4.11: Plant class definition in VDM

Listing 4.12 shows the corresponding Rust code for the VDM class in Listing 4.11.

```
lazy_static! {
    static ref experts: Set<Expert> = set!{...};
}

pub type Period = Token;

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct Plant {
  alarms: Set<Alarm>,
  schedule: Map<Period, Set<Expert>>
}

impl Plant {
  pub fn new(sch: Map<Period, Set<Expert>>) -> Plant {
    let mut instance: Plant = Plant::default();
    instance.cg_initPlant_1();
    return instance;
  }

  pub fn cg_initPlant_1(&mut self, sch: Map<Period, Set<Expert>>) {
    self.schedule = sch;
  }

  pub fn PlantInv(als: Set<Alarm>, sch: Map<Period, Set<Expert>>) -> bool {
    ...
  }

  pub fn ExpertToPage(&mut self, a: Alarm, p: Period) -> Expert {
    ...
  }
}

impl Default for Plant {
    fn default() -> Plant {
```

```
        let alarms: Set<Expert> = set!();
        let schedule: Map<Period, Set<Expert>> = Default::default();

        Plant {
            alarms: alarms,
            schedule: schedule,
        }
    }
}
```

Listing 4.12: Plant class definition in VDM

In the translation in Listing 4.12, the type definition, and the value sections have been moved outside the class. VDM values may be initialized using expressions, however in Rust static values can only be initialized using literals. This limitation is overcome by using the `lazy_static` macro from the lazy_static crate, which allows initialisation using expressions.

Functions and operations

The operation `ExpertToPage` was translated into a method, denoted by the explicit **`self`** parameter. Since operations may mutate object state, but do not need ownership of the instance, the explicit self parameter becomes a *mutable borrow*(**`&mut self`**).

Operations in VDM may be defined as *pure* or only declaring **`rd`**-mode variables in the external section, when using the *Extended Explicit Operation* syntax, in which case the **`self`** parameter becomes an *immutable borrow* (**`&self`**). Alternatively, it is possible to perform call graph analysis to determine whether the **`self`** parameter needs to be mutable. Passing **`self`** by immutable borrow enables reader/writer usage in a concurrent scenario, enables cleaner code for the consumer of the translated model, and explicitly conveys the semantics of the operation.

The constructor in VDM is an operation, which works on an instance whose instance variables have been initialised to either the initial value if defined, or a default value. This is explicit in Rust; classes implement the `Default` trait, defining a function for creating a class instance with the initial values. If a constructor is defined, it is then called on that default instance, as shown can be seen in the `new()` function. The constructor is translated to a method named `cg_init<className>_1()`, in this case `cg_initPlant_1()`. The function `PlantInv` has been translated into an *associated function*.

Note, that referring to associated functions requires scoping, even within the defining **`struct`**. Eg. referring to `PlantInv` is done like so: `Plant::PlantInv`.

Object references and limitations

In VDM++ class instances are passed by reference, and there is no concept of ownership. This differs from Rust, which requires that each object has exactly one owner. Calling operations on a VDM class instance requires a *mutable borrow*, as detailed in section 4.3.4. Since Rust doesn't allow aliasing of mutable borrows, only one can exist at a time. This limitation can be enforced at run-time instead of compile-time by using an abstraction which provides reference counting and dynamic checking of borrow status, such as `Rc<RefCell<T»` or `Arc<Mutex<T»` for concurrent scenarios. Using these abstractions could result in increased code generation complexity for explicitly managing reference counts, borrow status and cyclic references. In addition, introducing reference counting could lead to decreased performance of the generated code [32, 33]. This issue is compounded by Rust requiring runtime checking of borrow status for aliased instances,

further decreasing performance. Because of this, translation input is limited to the VDM models in which object owners are clearly defined.

Modelling for translation 1 : Classes

- Class instances must have exactly one owner, thus cannot be aliased.
- Passing references (borrows) to class instances as function arguments is not supported, which is a limitation of the current intermediate representation used in the code generation platform as described in Section 5.5. Thus passing classes instance to functions or operations is not supported.
- Because of these limitations, classes are best suited as code modules, while other data structures must be used to pass data around.
- Function and operation overloading is not supported by Rust, and is not translated.
- Inheritance and polymorphism is not supported.

4.3.5 Union types

VDM provides union types, that is, values of a union type may be any of the types in the set of types comprising the union type. Two Rust constructs are considered for mapping VDM union types; `Box<Any>` and the **`enum`**-type. `Box<Any>` can hold any value, but requires boxing(heap allocation), similar to an `Object`-reference in Java. Using `Box<Any>` would also remove explicit union-types from the generated code, making it harder to reason about; in particular it removes compile-time guarantees about which type a value might hold, if the value is a union type. The **`enum`** construct in Rust is an implementation of tagged, disjoint unions, and hence is a subset of the VDM union. However, it allows for compile-time guarantees, and pattern matching to determine the type similar to the VDM approach. The limitations of this approach is listed after the example.
In this work, the **`enum`** construct is used for translation to exploit the benefits of compile-time type guarantees and possible performance benefits offered by using pattern matching over the explicit dynamic type checking which would be required by the `Box<Any>`-approach.

Example

Listing 4.13 defines two VDM union types.

```
foo = <Q1> | <Q2>;
bar = <Q1> | int;
```

Listing 4.13: VDM union definitions.

In Rust this gets translated to an enumerated type, with a variant for each type of the original union type as shown below. The invocations of the macro `impl_union!` implements `From`-traits as described below, as well as traits for string representations.

```
enum foo { Ch0(quotes::Q1), Ch1(quotes::Q2), }
```

```
impl_union! { foo,
  quotes::Q1 as foo::Ch0,
  quotes::Q2 as foo::Ch1
}

enum bar { Ch0(quotes::Q1), Ch1(i64), }
impl_union! { bar,
  quotes::Q1 as foo::Ch0,
  i64 as foo::Ch1
}
```

Listing 4.14: Rust union definitions.

The following example shows instantiation and conversion between union types and the comprising types.

```
let f: foo = <Q1>,
    b: bar = f,
    i:<Q1> = b
in (...);
```

Listing 4.15: VDM union instantiation and conversion.

VDM provides implicit conversion between types - this is not the case in Rust. When dealing with **enum**-types Rust requires "packing" and "unpacking" using pattern matching. Listing 4.16 presents the naive Rust translation of Listing 4.15.

```
// pack
let f: foo = foo::Ch0(quotes::Q1);

// unpack + pack
let b: bar = match f {
               foo::Ch0(val) => bar::Ch0(val),
               _ => panic!("could not convert"),
             };

// unpack
let q: quotes::Q1 = match b {
                      bar::Ch0(val) => val,
                      _ => panic!("could not convert"),
                    };

...
```

Listing 4.16: Naive Rust union instantiation and conversion.

The packing and unpacking logic can be moved into implementations of the **trait** `From<T>` **for** `U`. Thus, implementing

- `From<quotes:Q1>` **for** `foo`
- `From<foo>` **for** `bar`
- `From<bar>` **for** `quotes::Q1`

```
let f: foo = foo::from(quotes::Q1); //pack

let b: bar = bar::from(f); // unpack + pack

let q: quotes::Q1 = quotes::Q1::from(b); // unpack

...
```

Listing 4.17: Rust union instantiation and conversion.

allows taking advantage of type inference to produce the Rust code in Listing 4.17.
Identifying the current concrete type of a union type variable can be done using the **cases**-expression. The following Listing is the body of the **let**-expression in Listing 4.15.

```
cases f:
  <Q1> -> IO`println("got Q1"),
  <Q2> -> IO`println("got Q2")
end
```

Listing 4.18: VDM union identification.

Translating Listing 4.18 to Rust we the code in Listing 4.19.

```
1 match f {
2     foo::Ch0(quotes::Q1) => IO::println(strseq!("got Q1")),
3     foo::Ch1(quotes::Q2) => IO::println(strseq!("got Q2"))
4 }
```

Listing 4.19: Rust union identification.

Note that the **match**-expression matches on the variants of the generated **enum** type, and not directly on the type.

Limitations

The approach for unions in this thesis requires union types to be declared and named before use. In VDM unions are not always declared, but may "appear" as needed. Union types may be implicitly created in the following situations, which are not covered by this work:

Directly in type annotations:

```
let x: nat | char = ...

f(nat | char) -> bool
f(in) =
```

Heterogeneous collections:

```
-- type of s1 is seq of (int | char)
```

```
let s1 = {1,'1'}
```

Expressions with multiple possible value types:

```
-- value of if-expression is of type int | char
if x then 1 else '1'
```

Tests for equality between disjoint types can be interpreted as a test between two values of a union type comprised of the two types.

```
-- interpreted as test over the type nat | bool
1 = false
```

4.3.6 Optional types

VDM includes an optional type `[T]`, which is shorthand for the union `T | nil`. This type will be represented by the `Option` type in Rust. Values of `Option<T>` can be tested for equality with `None`, similar to testing for equality with `nil` in VDM.

```
enum Option<T> {
    None,
    Some(T),
}
```

Listing 4.20: The `Option` type.

In VDM, values can be implicitly converted between the types `T` and `[T]`, however Rust requires explicit conversion between of `T` and `Option<T>`. Conversions are shown in Listing 4.21. Line 1 and 2 creates values of an optional type corresponding to `[char]`. Line 3 converts from `[char]` to `char`, but will *panic* with the given message, if the the optional is `nil`/`None`.

```
1 let noValue: Option<char> = None;
2 let hasVal: Option<char> = Some('A');
3 let charVal: char = hasVal.expect("optional was nil");
```

Listing 4.21: Conversions for `Option` types.

4.3.7 Lambda expressions

VDM allows function values to be created using the **`lambda`** keyword. Listings 4.22 and 4.23 shows a lambda expressed VDM and Rust, respectively.

```
let int_test = lambda i: int & i = 1 in ...
```

Listing 4.22: VDM lambda expression.

```
let int_test = |i: i64| -> bool { i == 1 };
...
```

Listing 4.23: Rust lambda expression.

Limitations

The code generator implemented for this thesis does not implement function types, and only supports lambda expressions as part of transforming quantified expressions, comprehensions and **`let be ... st`**. Thus lambda expressions cannot be used directly in VDM models to be translated.

Possible translations

VDM permits passing functions around by value, a Rust translation of this feature requires heap allocating function values and performing reference counting.
It is possible to implement limited support for higher-order functions while avoiding heap allocation and reference counting by using references as shown in Listings 4.24 and 4.25. With this approach lambda references can be passed to functions and operations, but not be returned from functions or operations, and not assigned to any reference which outlives the lambda value.

```
static call_lambda: int -> int ==> int
call_lambda(f) == return f(3);

...
call_lambda(lambda i: int & i + 1 )
```

Listing 4.24: VDM higher order operation.

```
fn call_lambda(f: &Fn(i64) -> i64) -> i64 {
  return f(3);
}

...
call_lambda(&|i: i64| -> i64 { i + 1 })
```

Listing 4.25: Rust higher order method.

4.4. Value semantics

All types in VDM excluding class references are value-types, and thus are passed by-value copying the original value. In Rust this must be done explicitly using the `clone` method of the `Clone` trait, since the default behavior is *move*-semantics. To ensure correct behavior values are cloned when:

- Passed to a function or operation.
- Assigned to a new variable name.

Only values referred to by a variable expression, field expression or application on a **map** or **seq** type requires copying, as these are the only expressions referring to a value, whereas all other expressions produces values.

4.5. Patterns

VDM defines patterns for matching values of a particular type. The following pattern types are translated:

- Pattern identifier
- Match value
- Tuple pattern
- Record pattern

Rust allows patterns matching similarly to VDM. The translated patterns each have a direct Rust counterpart.
Patterns identifiers is identical in Rust and VDM, except Rust uses '_' as 'don't care' symbol. *Match value* patterns are identical when using literals, however matching against an expression is translated using guard patterns as shown in Listings 4.26 and 4.27. Rust does not allow refutable patterns in **let**-bindings or formal parameters. Because of this, match value patterns cannot be used in these constructs.

```
let k = 2 in
cases 1:
  (k) -> ...,
  others -> ...
end;
```

Listing 4.26: Pattern matching against an expression value in VDM.

```
let k: u64 = 2;
match 1 {
  val if val == k => ...,
  _ => ...
}
```

Listing 4.27: Pattern matching against an expression value in Rust using a guard pattern.

Listings 4.28 and 4.29 show record and tuple patterns in VDM and Rust, respectively.

```
-- record type declaration
Point :: x : int
     y : int;

-- record pattern
let mk_Point(x,y) = mk_Point(1,2) in ...

--tuple pattern
let mk_(x,y) = mk_(1,2) in ...
```

Listing 4.28: Record and tuple patterns in VDM.

```
// record type declaration
struct Point {
  pub x: i64,
  pub y: i64,
}

-- record pattern
let Point{x: x, y: y } = Point::new(1,2);

--tuple pattern
let (x, y): (i64, i64) = (1,2);
```

Listing 4.29: Record and tuple patterns in Rust.

Limitations

In addition to the translated patterns, VDM defines patterns for the collection types and object references. These patterns do not have a direct translations to Rust, and as a result the following pattern types are not translated:

- Set enumeration and union patterns
- Sequence enumeration and concatenation patterns
- Map enumeration, maplet and munion patterns
- Object pattern

4.6. Local bindings

VDM offers **let**-bindings as both expressions and statements. **let**-bindings has the form shown in Listing 4.30, where `p1, ..., pn` are patterns, `e1, ..., en` are expressions matching the corresponding pattern, and `b` is the body involving the patterns. The body is either an expression or a statement depending on the context.

```
let p1 = e1, ..., pn = en in b
```

Listing 4.30: A **let**-binding in VDM.

let-bindings create *non-updatable* variable bindings, corresponding to **let** in Rust with one exception: operations can be called on bindings to class instances. Hence, bindings created to class instances must be mutable using **let mut** in Rusts.
The identifiers created by the patterns are only valid in the body of the **let**-binding, this scoping is translated to Rust using blocks. Blocks can contain both statements, and act as expressions. Thus, **let**-bindings are translated as shown in Listing 4.31.

```
}
  let p1 = e1;
  ...
  let pn = en;

  b
```

```
}
```

Listing 4.31: A **let**-binding in Rust.

The same translations apply to **def**-bindings. Values declared using the **dcl**-statement are translated in a similar fashion, however, those are mutable corresponding to **let mut** in Rust.

Examples

Listings 4.32 and 4.33 shows variable binds in VDM and Rust, respectively.

```
static op: int * int ==> int
op(i, j) == (
  dcl n: int := 0;
  if i = j then
    n := 1;

  return let r = i + j in r + n;
);
```

Listing 4.32: Variable-bindings in VDM.

```
fn op(i: i64, j: i64) -> i64 {
  let mut n: i64 = 0;
  if i == j {
    n = 1;
  }

  return {
    let r: i64 = i + j;
    r + n
  };
}
```

Listing 4.33: Variable-bindings in Rust.

4.7. Set binds

In VDM multiple constructs work with set or type binds. For both single and multiple binds, the set of values defined by the binding patterns are iterated. In the case of single binds, the values of a single set is iterated and evaluated. In the case of multi bindings, the set of values to be evaluated is the cartesian product of the values defined by the bindings. To create the set of the cartesian product, the macro `cartesian_set!` is utilized:

```
let set: Set<(i64, i64, i64)> = cartesian_set!{set!{3,6,9},
                                               set!{3,6,9},
                                               set!{2,5,8}}
```

4.7.1 Limitations

Type binds require performing a potentially infinite number of predicate evaluations, thus, these are not feasible in executable code. Hence, type binds are not supported by the translation.

4.8. Quantified expressions

VDM includes three types of quantified expressions: **`exists1`**, **`exists`**, **`forall`**.

4.8.1 `exists1`

The simplest expressions is the *unique existential expression*, **`exists1`**, which has the form:

```
exists1 bd & e
```

and thus only has a single bind. Translation of **`exists1`** may be exemplified as follows:

```
exists1 x in set {1,2,3} & x = 2
```

In Rust, the pattern, in this case `x`, becomes a formal parameter of the closure, and the predicate becomes the body.

```
set!{1,2,3}.exists1(|x| x == 2)
```

4.8.2 `exists`

`exists` allows multi binds. For multi binds, the values to iterate are the values defined by the cartesian product of the values of each set in the binding.

```
exists x,y in set {3,6,9}, z in set {2,5,8} & x > y && x > z
```

Note that the tuples values are pattern matched in the formal parameter for the closure given to the `exists` function, giving direct access to each member of the tuple in the predicate.

```
cartesian_set!{set!{3,6,9}, set!{3,6,9}, set!{2,5,8}}
  .exists(|(x,y,z)| x > y && x > z)
```

4.8.3 `forall`

`forall` is similar the other quantified statements. Example given:

```
forall i in set {2,4,6} & i mod 2 = 0
```

Translated to Rust:

```
set!{2,4,6}.forall(|i| i % 2 == 0)
```

4.9. Collection comprehensions

VDM provides comprehension expressions to construct collections. Comprehensions are comprised by a filter- and a map-operation.

Examples

Set comprehension

```
{ i*2 | i in set {1,2,3,0} & i mod 2 = 0}
```

```
set!{1,2,3,0}.set_compr(|i| i % 2 == 0, |i| i * 2)
```

Seq comprehension

```
[ i*2 | i in set {1,2,3,4} & i mod 2 = 0]
```

```
set!{2,3,1,4}.seq_compr(|i| i % 2 == 0, |i| i * 2)
```

Map comprehension

```
{ i |-> i*2 | i in set {1,2,3,4} & i mod 2 = 0}
```

```
set!{2,3,1,4}.map_compr(|i| i % 2 == 0, |i| (i, i * 2 ))
```

Note that in the translation of the map comprehension, the maplet expression becomes a tuple expression.

4.10. *let-be-such-that* statements

The **`Set<T>`** type of the Rust runtime for translated models includes a `be_such_that` function:

Example

```
let i,j in set {1,2,3} be st i + 2 = j in ...
```

In Rust:

```
let (i,j) = cartesian_set!(set!{1,2,3}, set!{1,2,3})
              .be_such_that(|(i,j)| i + 2 == j);
...
```

4.11. Conditional constructs

VDM defines **`if`**- and **`cases`**-expressions and statements. Rust defines the **`if`**- and **`match`**-expressions with similar semantics, both of which may also be used as a statement. **`if`**-expressions and statements may be from VDM to Rust as shown in Listings 4.34 and 4.35.

```
-- if-expression
let x = 5,
  y = if x = 5 then 1 else 2 in ...

-- if-statement
if x = 5 then
  IO'println("x is 5")
elseif x = 6
  IO'println("x is 6")
else
  IO'println("x is not 5 or 6")
```

Listing 4.34: **`if`** expressions and statements in VDM.

```
// if-expression
let x = 5;
let y = if x == 5 { 1 } else { 2 };

// if-statement
if x == 5 {
  IO::println(strseq!("x is 5"));
} else if x = 6 {
  IO::println(strseq!("x is 6"));
} else {
  IO::println(strseq!("x is not 5 or 6"));
}
```

Listing 4.35: **`if`** expressions and statements in Rust.

The **cases**-expression and statement may be translated from VDM to Rust as shown in Listings 4.36 and 4.37.

```
-- cases-expression
let x = 5,
  y = cases x:
    5 -> 1,
    others -> 2
  end in ...;

-- cases-statement
cases x:
  5 -> IO'println("x is 5"),
  6 -> IO'println("x is 6"),
  others -> IO'println("x is not 5 or 6")
end;
```

Listing 4.36: **cases** expressions and statements in VDM.

```
// match-expression
let x = 5;
let y = match x {
      5 => 1,
      _ => 2,
    }

// match-statement
match x {
  5 => IO::println(strseq!("x is 5")),
  6 => IO::println(strseq!("x is 6")),
  _ => IO::println(strseq!("x is not 5 or 6")),
}
```

Listing 4.37: **match** expressions and statements in Rust.

Limitations

As discussed in Section 4.3.5, both **if**- and **cases**-expressions are allowed to return different types depending which condition is satisfied. In this case, the type of the **if**- or **cases**-expression is the union of the possible value types. This is not supported by the translation. In addition, the **cases**-statement has not been implemented in the code generator.

4.12. Looping constructs

This sections presents translations for the looping constructs provided by VDM.

4.12.1 While-loop

VDM and Rust provides while loops with identical semantics. The Rust while loop is defined as shown in Listing 4.38, where `e` is a boolean expression, and `s` is a statement. `s` is evaluation as a

long as e is **true**.

```
while e {
  s
}
```

Listing 4.38: The while-loop in Rust.

4.12.2 Index-For loop

The VDM index for loop is defined as shown in Listing 4.39.

```
for id = e1 to e2 by e3 do
s
```

Listing 4.39: Definition of the index for loop in VDM.

Rust does not provide a similar construct, so the loop is transformed to a while-loop, and translated as shown in Listing 4.40.

```
let mut id = e1;
let to = e2;
let step_by = e3;

while id <= to {
  s
  id = id + step_by;
}
```

Listing 4.40: Rust translation of the index for loop.

4.12.3 set and seq for loop

VDM provides looping constructs for **set** and **seq** collections. These looping constructs are translated to the for-loop provided by Rust as shown in Listing 4.41, where e is bound the each element in the collection col, and s is a statement which is evaluated for each element of the collection col. For loops in Rust works on any value implementing the trait IntoIterator, which is implemented by the collections Set<T> and Seq<T>.

```
for e in col {
  s
}
```

Listing 4.41: For loops in Rust.

Chapter 5

Case study: The Alarm model

This chapter introduces a simple VDM++ model, the translated Rust version, and discusses translation of some of the VDM constructs required as well as future improvements to the translation. This chapter builds on the basic understanding of Rust described in chapter 2, and the translations from VDM to Rust described in chapter 4.

5.1. Introduction

This chapter presents a case study for the VDM++ to Rust code generator. This case study was used to validate the choices made when designing and implementing the VDM++ to Rust code generator. Furthermore, this case study has served to direct and document the work carried out.
The chapter presents the structure and purpose of the VDM Alarm model, on which this case study is based. Section 5.3 will then present the original VDM specification, and present and describe the corresponding Rust code. Section 5.4 discusses some of the issues which was resolved for the translation, in particular creating correct Rust code, when working with mutable borrows, unions, and how to handle type conversions in Rust.
Finally, section 5.5 describes issues which result in an incomplete translation to Rust, or Rust code which may run slow or be memory inefficient. The performance issues are caused by a lack of support for references in the code generator platform. This limitation is worked around by exclusively passing values by value, which requires copying values using the `clone()`-function.

The following VDM constructs are *covered* by this case study:

- Classes with **`instance variables`** and **`values`**
- Custom constructors
- Functions and operations
- Preconditions
- Named types
- Union type declaration and instantiation
- Quote literals
- Token type
- Record types
- Enumeration of **`set`**, **`seq`** and **`map`**
- Operations on **`set`** and **`map`**
- Quantified expressions (**`forall`** and **`exists`**)

- **let be ... st** expressions.
- **if**-expressions.
- Set comprehension

The following VDM constructs from the original model are *not covered* by this case study:

- Type invariants
- Postcondtions

Specification constructs are covered in Chapter 6.

5.2. The Alarm model

The alarm model used for this example is adapted from the book *"Validated Designs for Object-oriented Systems"* [3, pp. 20-38], and can be found in the "Examples" Section on the Overture website [8]. This example is based on a chemical plant and models the alarm system used to determine which expert to call for a given alarm, as well as ensuring the correctness of the expert duty schedule.
This particular model was chosen for the initial case study, because it is a simple model, yet covers enough features of the VDM language to require a non-trivial translation. The translation is non-trivial because multiple VDM constructs such as unions, set comprehensions and preconditions, have no direct equivalent in Rust, and thus requires transformations of the original model. In addition, the alarm model is comprised of multiple classes, and multiple operations, and thus the basic structure of the resulting Rust code must be defined in a way which preserves the relationship between these.
In section 5.2.1 the modifications made to the model in order to make it suitable for translation to Rust are discussed. Section 5.2.2 discusses the purpose and structure of the model.

5.2.1 Transforming data classes to record types

It is a goal of this thesis to create a translation, which yields *efficient* Rust code. In order to be able achieve that goal, it is required that the VDM models to be translated are written with this goal in mind. As discussed in section 4.3.4, a model should be using classes to modularize the model, and record types to carry data. To that end the Alarm model has been modified for the case study; the original model included two classes (`Expert` and `Alarm`), which did not contain logic, and was used as records; that is, only one reference to each instance, and no mutation after instantiation. These have been converted to actual record-types for the case study. The model retains the original behaviour after being modified.

5.2.2 Structure of the Model

The structure of the alarm model can be seen on the class diagram in Figure 5.1. All logic is contained in the `Plant`-class, which holds the current schedule as a **map** `Period to` **set** `of Expert`. The main functionality is implemented in the operations `ExpertToPage()` and `ExpertIsOnDuty()`. `ExpertToPage()` finds a qualified, on-duty expert, given a `Period` and an `Alarm`, specifying the required qualification. The operation `ExpertIsOnDuty()` will, given an `Expert` return a **set** `of Period`, describing during which periods the `Expert` is on duty. The operations `AddExpertToSchedule()` and `RemoveExpertFromSchedule()` will add and remove, respectively, an expert from the schedule in the given `Period`. The class `Test1` holds test data and logic.

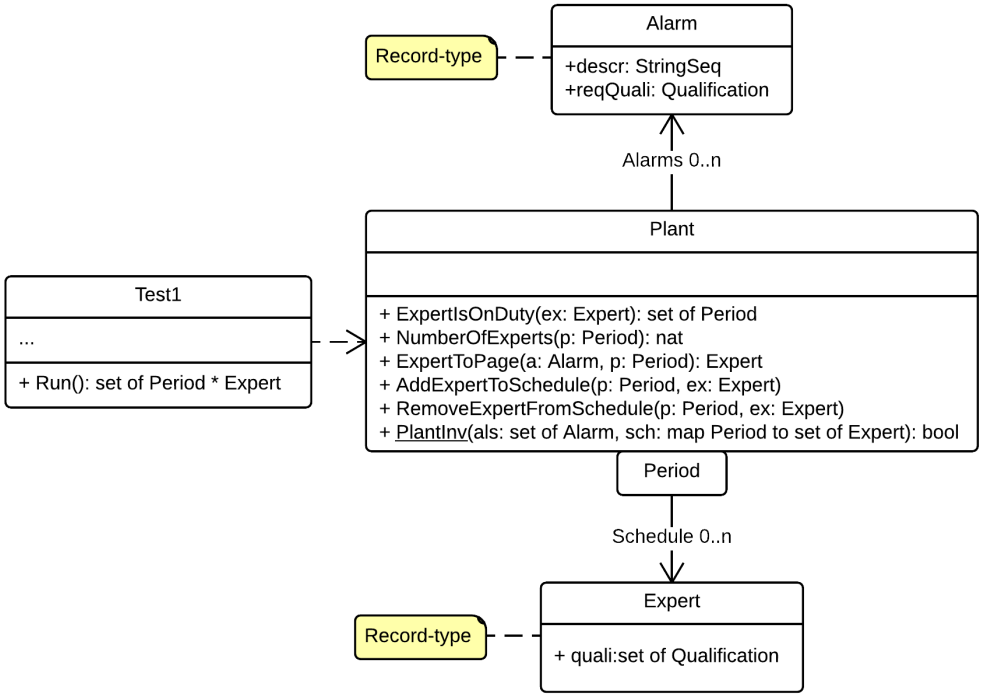

Figure 5.1: Class diagram for the Alarm model.

5.3. Rust translation

This section documents the output of the code generation. Translation of the whole model is covered, except translation of invariants and postconditions, which is not implemented in the code generator. Specification constructs are discussed in Chapter 6.

5.3.1 The `Plant` class

The `Plant` class contains all logic and data types of the model. This section covers the contents of the plant class, which belongs inside the class definition as shown in Listing 5.1.

```
1 class Plant
2 -- content omitted
3 end Plant
```

Listing 5.1: Definition of the `Plant` class.

Type definitions

This section describes the translations of the type definitions in the `Plant` class.

```
1 public Period = token;
2 public StringSeq = seq of char;
3
4 public Qualification = <Mech> | <Chem> | <Bio> | <Elec>;
```

```
5
6  public Alarm :: descr : String
7                  reqQuali : Qualification;
8
9  public Expert :: quali : set of Qualification;
```

Listing 5.2: VDM type definitions in the `Plant` class.

The named types translate to aliases in Rust as seen in Listing 5.3.

```
1  pub type Period = Token;
2  pub type StringSeq = Seq<char>;
```

Listing 5.3: Plant named type definitions.

The `Qualification` type in line 4 of Listing 5.2 is a named union, which is translated to a Rust **`enum`** as described in Section 4.3.5. The translated union type is split into two parts: the **`enum`** declaration, and an invocation of the `impl_union!` macro, which expands to implementations of the `From<T>`-**`trait`** described in Section 4.3.5. These implementations of the `From<T>`-**`trait`** allows conversion between the instances of the enum variants and the contained value. The translated union type can be seen in Listing 5.4.

```
1  #[derive(PartialEq, Eq, Clone, Hash)]
2  pub enum Qualification {
3      Ch0(quotes::Bio),
4      Ch1(quotes::Chem),
5      Ch2(quotes::Elec),
6      Ch3(quotes::Mech),
7  }
8
9  impl_union! { Qualification:
10   quotes::Bio as Qualification::Ch0,
11   quotes::Chem as Qualification::Ch1,
12   quotes::Elec as Qualification::Ch2,
13   quotes::Mech as Qualification::Ch3
14 }
```

Listing 5.4: Declaration of the `Qualification` union type in Rust.

The quote literals in line 4 of Listing 5.2 are translated to *unit-like structs* as described in 4.2.4, and placed in a module(in separate source file) called `quotes`. Unit-like structs are distinct types without data members as described in section 2.4.2.

The VDM records `Alarm` and `Expert`, as defined on line 6 and 9, respectively, of Listing 5.2, have been translated as shown in Listing 5.5.

```
1  #[derive(PartialEq, Eq, Clone, Hash, Debug)]
2  pub struct Alarm {
3      pub descr: StringSeq,
4      pub reqQuali: Qualification,
5  }
6  impl_record! { Alarm: descr as StringSeq,  reqQuali as Qualification }
7
8  #[derive(PartialEq, Eq, Clone, Hash, Debug)]
```

```
9  pub struct Expert {
10     pub quali: Set<Qualification>,
11 }
12 impl_record! { Expert: quali as Set<Qualification> }
```

Listing 5.5: Declaration of the records Alarm and Expert in Rust.

Each record is translated as a **struct** definition and an invocation of the impl_record! macro, which expands to a new-function, as shown in Listing 5.6. The new() function simply creates an instance of the record-**struct** and initializes the member variables with the provided values.

```
1 impl Alarm {
2   pub fn new(descr: StringSeq, reqQuali: Qualification) -> Alarm {
3     Alarm {descr: descr, reqQuali: reqQuali}
4   }
5 }
```

Listing 5.6: Generated new() function for the Alarm record.

Instance variables and constructors

The instance variables of the Plant class are declared in VDM as shown in Listing 5.7.

```
1 alarms   : set of Alarm;
2 schedule : map Period to set of Expert;
3 inv PlantInv(alarms,schedule); -- not part of translation
```

Listing 5.7: Instance variables for the Plant class in VDM.

Like records, classes are also represented as **struct**s in Rust. Invariants are not part of the translation, but are treated in Section 6.6. In addition to the struct, the Default trait has been implemented for the class, which allows creating instances of the class with default values, as is needed for the constructor. In this case none of the instance variables declare an initial value, and hence are assigned their default values as seen on line 11 and 12 in Listing 5.8. Rust disallows creating an instance without initializing the instance variable, hence a default value is needed. Accessing a value which has not been assigned a value produces a runtime error in VDM, which means that not assigning a value to an instance variable before accessing it, is a model error. Because of this, the default value assigned in the implementation of the Default trait will always be overwritten with a valid value before use, if the model is correct. If the model fails to assign a value to an instance variable before using, the Rust program will produce incorrect results instead of reporting an error.

```
1 #[derive(PartialEq, Eq, Clone, Hash, Debug)]
2 pub struct Plant {
3     alarms: Set<Alarm>,
4     schedule: Map<Period, Set<Expert>>,
5 }
6
7 ...
8
```

```
9  impl Default for Plant {
10     fn default() -> Plant {
11         let alarms: Set<Alarm> = Default::default();
12         let schedule: Map<Period, Set<Expert>> = Default::default();
13
14         Plant {
15             alarms: alarms,
16             schedule: schedule,
17         }
18     }
19 }
```

Listing 5.8: Instance variables and `Default` implementation for the `Plant` class in Rust.

The model declares a constructor, which in VDM is an operation with the name of the class, returning an instance of the class. The constructor also defines a precondition.

```
1  public Plant: set of Alarm *
2                map Period to set of Expert ==> Plant
3  Plant(als,sch) ==
4  ( alarms := als;
5    schedule := sch
6  )
7  pre PlantInv(als,sch);
```

Listing 5.9: Constructor for the `Plant` class in VDM.

In VDM, creation of the class instance is implicit, and it is optional to explicitly return it from the constructor. This is not the case in Rust. In Rust this process is split into the value creation, implemented in the `default()` function as part of the `Default` trait, and the invocation of the constructor operation, called `cg_init_Plant_1` in this case. These two steps are then wrapped in the `new()` function, which is an *associated function* to the `Plant` class, and also the conventional way to create values in Rust. This results in the output in Listing 5.10. The precondition of the constructor results in a method, with an immutable **`self`** reference, and hence, the precondition method cannot alter the state of the model. Assertion of the precondition has been inserted into the constructor operation, as seen on line 8 in Listing 5.10.

```
1  fn pre_Plant(&self, als: Set<Alarm>, sch: Map<Period, Set<Expert>>) -> bool {
2      return Plant::PlantInv(als.clone(), sch.clone());
3  }
4
5  pub fn cg_init_Plant_1(&mut self, als: Set<Alarm>
6          , sch: Map<Period, Set<Expert>>) -> () {
7    assert!(self.pre_Plant(als.clone(), sch.clone()));
8    self.alarms = als.clone();
9    self.schedule = sch.clone();
10 }
11
12 pub fn new(als: Set<Alarm>, sch: Map<Period, Set<Expert>>) -> Plant {
13   let mut instance: Plant = Plant::default();
14   {
15     let arg0: Set<Alarm> = als.clone();
16     let arg1: Map<Period, Set<Expert>> = sch.clone();
17     instance.cg_init_Plant_1(arg0.clone(), arg1.clone());
18   }
```

```
19    return instance;
20  }
```

Listing 5.10: Constructor implementation for the Plant class in Rust.

Functions and Operations

All functions and operations, including the constructor in Listing 5.10, defined by the Plant-class, are defined inside an implementation block as shown in Listing 5.11.

```
1  impl Plant {
2    // functions and methods omitted.
3  }
```

Listing 5.11: Implementation block for the Plant class in Rust.

The VDM model contains a single function, which is used to evaluate the invariant for the Plant class. The VDM function PlantInv() is listed in Listing 5.12.

```
1  PlantInv: set of Alarm * map Period to set of Expert +>
2            bool
3  PlantInv(as,sch) ==
4    (forall p in set dom sch & sch(p) <> {}) and
5    (forall a in set as &
6       forall p in set dom sch &
7         exists expert in set sch(p) &
8           a.reqQuali in set expert.quali);
```

Listing 5.12: The function PlantInv in VDM.

The corresponding Rust output is shown in Listing 5.13. When translated, a function becomes an *associated function*, in this case for the Plant class. The quantified expressions of the VDM model become methods on **Set**<T>, and the predicates are transformed to VDM lambda expressions, which are then translated to Rust closures.

```
1  fn PlantInv(als: Set<Alarm>, sch: Map<Period, Set<Expert>>) -> bool {
2    sch.domain().forall(|p: Period| -> bool {
3      sch.get(p.clone()) != set!() }) &&
4    als.forall(|a: Alarm| -> bool {
5      sch.domain().forall(|p: Period| -> bool {
6        sch.get(p.clone())
7          .exists(|expert: Expert| -> bool {
8            expert.quali.in_set(a.reqQuali.clone())})
9      })
10   })
11 }
```

Listing 5.13: The function PlantInv in Rust.

Listing 5.14 shows the operations for making queries about the schedule. These operations could have been annotated as **pure** in VDM10 or defined an external section using *extended explicit operation* definitions, indicating that they do not mutate state of the class instance. This would

have allowed translation to methods with read-only access to the state using an immutable reference to self (**`&self`**). Neither syntax is handled by the code generator platform, and hence not by the Rust code generator.

```
1  public ExpertToPage: Alarm * Period ==> Expert
2  ExpertToPage(a, p) ==
3    let expert in set schedule(p) be st
4        a.reqQuali in set expert.quali
5    in
6      return expert
7  pre a in set alarms and
8      p in set dom schedule;
9  -- postcondition omitted
10
11 public NumberOfExperts: Period ==> nat
12 NumberOfExperts(p) ==
13   return card schedule(p)
14 pre p in set dom schedule;
15
16 public ExpertIsOnDuty: Expert ==> set of Period
17 ExpertIsOnDuty(ex) ==
18   return {p | p in set dom schedule &
19               ex in set schedule(p)};
```

Listing 5.14: The query operations of `Plant` in VDM.

The corresponding Rust methods are listed in 5.15. It can be seen, that preconditions for regular operations are handled identically to the precondition of the constructor operation. Furthermore, the **`let be .. st`** expression, like the quantified expressions has been transformed to a method on the **`set`**-type, and the predicate transformed to a lambda expression as seen in lines 14-16. The set comprehension in line 26-32 follows a similar approach using Rust iterators internally, with the predicate expression being applied by a filter-operation, and the mapping expression being applied by a map-operation.

```
1  fn pre_ExpertToPage(&self, a: Alarm, p: Period) -> bool {
2    return self.alarms.in_set(a.clone()) &&
3      self.schedule.domain().in_set(p.clone());
4  }
5
6  fn pre_NumberOfExperts(&self, p: Period) -> bool {
7      return self.schedule.domain().in_set(p.clone());
8  }
9
10 pub fn ExpertToPage(&mut self, a: Alarm, p: Period) -> Expert {
11   assert!(self.pre_ExpertToPage(a.clone(), p.clone()));
12   let expert: Expert = self.schedule
13                            .get(p.clone())
14                            .be_such_that(|expert: Expert| -> bool {
15                                 expert.quali.in_set(a.reqQuali.clone())
16                            });
17   return expert;
18 }
19
20 pub fn NumberOfExperts(&mut self, p: Period) -> u64 {
21   assert!(self.pre_NumberOfExperts(p.clone()));
22   return self.schedule.get(p.clone()).card();
```

```
23 }
24
25 pub fn ExpertIsOnDuty(&mut self, ex: Expert) -> Set<Period> {
26   return self.schedule.domain().set_compr(|p: Period| -> bool {
27                                     self.schedule.get(p.clone())
28                                         .in_set(ex.clone())
29                                     },
30                                     |p: Period| -> Period {
31                                       p.clone()
32                                     });
33 }
```

Listing 5.15: The query operations of Plant in Rust.

Finally, the Plant class defines state mutating operations for adding and removing experts from the schedule as shown in Listing 5.16.

```
1  public AddExpertToSchedule: Period * Expert ==> ()
2  AddExpertToSchedule(p,ex) ==
3    schedule(p) := if p in set dom schedule
4                   then schedule(p) union {ex}
5                   else {ex};
6
7  public RemoveExpertFromSchedule: Period * Expert ==> ()
8  RemoveExpertFromSchedule(p,ex) ==
9    let exs = schedule(p)
10   in
11     schedule := if card exs = 1
12                 then {p} <-: schedule
13                 else schedule ++ {p |-> exs \ {ex}}
14 pre p in set dom schedule and
15 (forall a in set alarms &
16   exists expert in set schedule(p) \ {ex} &
17     a.reqQuali in set expert.quali);
```

Listing 5.16: Operations for adding and removing Experts from the schedule as defined by the Plant class in VDM.

Translation of the AddExpertToSchedule and RemoveExpertFromSchedule operations listed in 5.16 results in the Rust code in Listing 5.17. Evaluation of the argument expressions for the insert() method in line 8 has been separated from the method invocation to satisfy the borrow checker, which is discussed further in section 5.4.1. Both map- and set-enumeration are shown in line 30, translated using the map! and set! macros, respectively, both of which are part of the runtime library for the generated Rust code.

```
1  pub fn AddExpertToSchedule(&mut self, p: Period, ex: Expert) -> () {
2      let arg0: Period = p.clone();
3      let arg1: Set<Expert> = if self.schedule.domain().in_set(p.clone()) {
4          set!(ex.clone()).union(self.schedule.get(p.clone()))
5      } else {
6          set!(ex.clone())
7      };
8      self.schedule.insert(arg0.clone(), arg1.clone());
9  }
10
11 fn pre_RemoveExpertFromSchedule(&self, p: Period, ex: Expert) -> bool {
```

```
12    return self.schedule.domain().in_set(p.clone()) &&
13           self.alarms.forall(|a: Alarm| -> bool {
14              self.schedule
15                .get(p.clone())
16                .difference(set!(ex.clone()))
17                .exists(|expert: Expert| -> bool {
18                      expert.quali.in_set(a.reqQuali.clone())
19                })
20              });
21  }
22
23  pub fn RemoveExpertFromSchedule(&mut self, p: Period, ex: Expert) -> () {
24      assert!(self.pre_RemoveExpertFromSchedule(p.clone(), ex.clone()));
25      let exs: Set<Expert> = self.schedule.get(p.clone());
26      self.schedule = if exs.card() == 1 {
27          self.schedule.dom_restrict_by(set!(p.clone()))
28      } else {
29          self.schedule.ovrride(
30              map!(p.clone() => set!(ex.clone()).difference(exs.clone()))
31            )
32      };
33  }
```

Listing 5.17: `AddExpertToSchedule RemoveExpertFromSchedule` operations of `Plant` in Rust.

5.3.2 The `Test1` class

The `Test1` class contains the entry point for the simulation and sets up, and runs a simple test.

Instance variables

The instance variables of the class are declared as shown in listing 5.18.

```
1  a1   : Plant`Alarm  := mk_Plant`Alarm("Mechanical fault", <Mech>);
2  a2   : Plant`Alarm  := mk_Plant`Alarm("Tank overflow", <Chem>);
3  ex1  : Plant`Expert := mk_Plant`Expert({<Mech>,<Bio>});
4  ex2  : Plant`Expert := mk_Plant`Expert({<Elec>});
5  ex3  : Plant`Expert := mk_Plant`Expert({<Chem>,<Bio>,<Mech>});
6  ex4  : Plant`Expert := mk_Plant`Expert({<Elec>,<Chem>});
7  plant: Plant  := new Plant({a1},{p1 |-> {ex1,ex4},
8                                  p2 |-> {ex2,ex3}});
```

Listing 5.18: Instance variables of `Test1` in VDM.

These declarations result in the **`struct`** declaration in Listing 5.19. Note that the `Alarm` and `Expert` record types from the `Plant` class are qualified (scoping with `::`)) by module, and not by class, since in Rust, types are declared in a module with no association to other types.

```
1  #[derive(PartialEq, Eq, Clone, Hash, Debug)]
2  pub struct Test1 {
3      a1: ::Plant_mod::Alarm,
4      a2: ::Plant_mod::Alarm,
5      ex1: ::Plant_mod::Expert,
```

```
    ex2: ::Plant_mod::Expert,
    ex3: ::Plant_mod::Expert,
    ex4: ::Plant_mod::Expert,
    plant: Plant,
}
```

Listing 5.19: The Test1 **struct**.

As with the instance variables of the Plant class, the initial values of the instance variables are placed in an implementation of the Default trait as shown in Listing 5.20. Note the calls to the function ::Plant_mod::Qualification::from(), which convert plain types into appropriate variants of the enumerated type implementing the Qualification union type.

```
impl Default for Test1 {
    fn default() -> Test1 {
        let a1: ::Plant_mod::Alarm =
            ::Plant_mod::Alarm::new(strseq!("Mechanical fault"),
                ::Plant_mod::Qualification::from(quotes::Mech));
        let a2: ::Plant_mod::Alarm =
            ::Plant_mod::Alarm::new(strseq!("Tank overflow"),
                ::Plant_mod::Qualification::from(quotes::Chem));
        let ex1: ::Plant_mod::Expert =
            ::Plant_mod::Expert::new(set!
                (::Plant_mod::Qualification::from(quotes::Mech),
                 ::Plant_mod::Qualification::from(quotes::Bio)));
        let ex2: ::Plant_mod::Expert =
            ::Plant_mod::Expert::new(set!
                (::Plant_mod::Qualification::from(quotes::Elec)));
        let ex3: ::Plant_mod::Expert =
            ::Plant_mod::Expert::new(set!
                (::Plant_mod::Qualification::from(quotes::Chem),
                 ::Plant_mod::Qualification::from(quotes::Bio),
                 ::Plant_mod::Qualification::from(quotes::Mech)));
        let ex4: ::Plant_mod::Expert =
            ::Plant_mod::Expert::new(set!
                (::Plant_mod::Qualification::from(quotes::Elec),
                 ::Plant_mod::Qualification::from(quotes::Chem)));
        let plant: Plant = Plant::new(set!(a1.clone()),
            map!(::Test1_mod::p1.clone() => set!(ex1.clone(), ex4.clone()),
                 ::Test1_mod::p2.clone() => set!(ex2.clone(), ex3.clone())));

        Test1 {
            a1: a1,
            a2: a2,
            ex1: ex1,
            ex2: ex2,
            ex3: ex3,
            ex4: ex4,
            plant: plant,
        }
    }
}
```

Listing 5.20: The Test1 Default implementation.

The Test1 class has no custom constructor, so the constructor returns an instance with default values as seen in Listing 5.21.

```
pub fn new() -> Test1 {
  let instance: Test1 = Test1::default();
  return instance;
}
```

Listing 5.21: The `Test1` constructor.

The `Test1` class defines **`values`** as shown in Listing 5.22.

```
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");
```

Listing 5.22: Values defined by `Test1` in VDM.

In Rust, static values must be initialised to some constant value. To allow values being initialised by arbitrary expressions, the `lazy_static!` macro from a crate of the same name is used. Translation of the values in Listing 5.22 results in the code in Listing 5.23.

```
lazy_static! {
   static ref p1: ::Plant_mod::Period = Token::new(&strseq!("Monday day"));
   static ref p2: ::Plant_mod::Period = Token::new(&strseq!("Monday night"));
   static ref p3: ::Plant_mod::Period = Token::new(&strseq!("Tuesday day"));
   static ref p4: ::Plant_mod::Period = Token::new(&strseq!("Tuesday night"));
}
```

Listing 5.23: Values defined by `Test1` in Rust.

Operations

The `Test1` class defines a single operation, `Run()`, which performs a simple test of the logic implemented in the `Plant` class. The VDM definition is shown in Listing 5.24.

```
public Run: () ==> set of Plant`Period * Plant`Expert
Run() ==
  let periods = plant.ExpertIsOnDuty(ex1),
      expert  = plant.ExpertToPage(a1,p1)
  in
    return mk_(periods,expert);
```

Listing 5.24: `Run()`-operation defined by `Test1` in VDM.

The output of translating the `Run()`-operation is listed in 5.25.

```
pub fn Run(&mut self) -> (Set<::Plant_mod::Period>, ::Plant_mod::Expert) {
  let periods: Set<::Plant_mod::Period> =
      self.plant.ExpertIsOnDuty(self.ex1.clone());
  let expert: ::Plant_mod::Expert =
      self.plant.ExpertToPage(self.a1.clone(), ::Test1_mod::p1.clone());
  return (periods.clone(), expert.clone());
}
```

Listing 5.25: Values defined by `Test1` in Rust.

5.4. Resolved Challenges

This section discusses issues which were resolved to enable translation of the Alarm VDM model.

5.4.1 Mutable borrows

As discussed in section 2.2, Rust disallows multiple borrows if one or more is mutable. Given the VDM map update statement in Listing 5.26, the code generator outputs the method call in line 1 of Listing 5.27, which is shorthand for the *Universal Function Call Syntax(UFCS)* notation in line 7 as discussed in section 2.5.1. The arguments are evaluated as shown in line 4-6, hence the `schedule` variable will be mutably borrowed as the first argument, when the third expression is reached. Since `schedule` is mutably borrowed at this point, any sort of access to that variable will cause the Rust compiler to issue an error and halt compilation.

```
1 schedule(p) := schedule(p);
```

Listing 5.26: Updating a **`map`** in VDM.

```
1 schedule.insert(p, schedule.get(p));
2
3 // de-sugars to UFCS
4 let map_self = &mut schedule;
5 let arg0 = p;
6 let arg1 = schedule.get(p);
7 Map::insert(map_self, arg0, arg1);
```

Listing 5.27: Updating a **`map`** in Rust.

A possible solution to this problem, is separation of argument evaluation and method invocation as seen in Listing 5.28. This changes the evaluation order, freeing the schedule variable to be mutably borrowed for the method call. VDM does not specify evaluation order.

```
1 {
2   let arg0 = p;
3   let arg1 = schedule.get(p);
4   schedule.insert(arg0, arg1);
5 }
```

Listing 5.28: Satisfying the borrow checker.

This problem may occur in method calls when mutable borrows are involved, which includes calls to non-pure operations and update statements for **`map`** and **`seq`** types.

5.4.2 Unions

Translating the alarm model requires dealing with union types. As described in sections 4.3.5 and 5.3, unions are translated to enumerated types with a variant for each member type. For the `Qualification` union type in this case(shown in Listing 5.2), all member types are quote literals, neither of which are used outside the `Qualification` type. Thus, the VDM union type definition is really an enumerated type, which could be represented more clearly using a plain enumeration as shown in Listing 5.29. The consistent approach applied in the code generator treats all unions the same way, which allows for less logic to tested and implemented.

```
1 pub enum Qualification { Bio, Chem, Elec, Mech }
```

Listing 5.29: Simple enumerated type.

5.4.3 Type conversion and declared types

Due to the way union types work in VDM[30] expressions are *assignment compatible* if the type of the source expression is a subset of the target type. This approach is fundamentally different in Rust, which provides no type coercions, and instead requires explicit casts and type conversions. This difference becomes an issue when handling numeric types and unions.

```
1 // type declaration
2 Qualification = <Bio> | <Chem>;
3
4 // operation definition
5 op: set of Qualification ==> ()
6 op(s) == ...
7
8 // call operation
9 op({<Bio>});
```

Listing 5.30: Type coercion in VDM.

Listing 5.30 is an example of type coercion in VDM; the set expression `{<Bio>}` passed as an argument in line 9 is given the type **`set of`** `<Bio>` by the Overture type checker, and not **`set of`** `Qualification` as expected by the operation. The solution employed in the code generator, relies on detecting the declared type, and then inserting type conversion expressions. The expected type of an expression is declared as part of a function or operation signature, or in association with variable declarations (**`let`** or dcl statements).
When translating Listing 5.30 to Rust, the set enumeration passed as argument when invoking `op()` will be transformed to an invocation of the macro `set!`, as seen in line 18. The type of this macro will be determined to be a method of the type `Qualification, ... ->` **`set of`** `Qualification` by analysing the intermediate representation for the declared type of the set enumeration. The type is found to be declared as the expected argument type for the operation `op()`. At a later stage, a type conversion transformation is applied, which detects that the `quotes::Bio` quote value is not of type `Qualification`, and thus adds a conversion expression - the call to `Qualification::from()`. The result is the Rust code in Listing 5.31.

```
1 // type declaration
```

```
#[derive(PartialEq, Eq, Clone, Hash)]
pub enum Qualification {
    Ch0(quotes::Bio),
    Ch1(quotes::Chem),
}
impl_union! { Qualification:
  quotes::Bio as Qualification::Ch0,
  quotes::Chem as Qualification::Ch1
}

// operation definition
fn op(&mut self, Set<Qualification>) {
  ...
}

// call operation
op(set!(Qualification::from(quotes::Bio)));
```

Listing 5.31: Type conversion in Rust.

5.5. Open issues

This section discusses issues which is preventing complete translation of Alarm model and issues which result in Rust code that may run slower and require more memory. Specification constructs have not been fully translated, but are discussed in chapter 6.

5.5.1 Utilizing Rust references

The intermediate representation in the current Overture code generator platform does not support references. This shortcoming means that the current Rust code generator is unable to utilize Rust references. The employed workaround is to pass-by-value, and copy object instances, not only to simulate the value semantics of VDM, but also to satisfy Rust's borrow checker. Because of this, objects are copied more than necessary, which may reduce performance, and increases code clutter as is evident by the inserted `clone()`-calls in the Listings in section 5.3.
Adding support for references could allow removing most `clone()`-calls as can be seen in Listing 5.32, when compared to the code generated output in Listing 5.15. Pervasive use of references could lower *locality of reference*, which could reduce performance as demonstrated by Bjarne Stroustrup in the keynote at Going Native 2012 [34].

```
pub fn ExpertToPage(&mut self, a: Alarm, p: Period) -> Expert {
   let expert: Expert = self.schedule
                                .get(&p)
                                .be_such_that(|expert: Expert| -> bool {
                                    expert.quali.in_set(&a.reqQuali)
                                });
   return expert;
}

pub fn NumberOfExperts(&mut self, p: Period) -> u64 {
    return self.schedule.get(&p).card();
}

```

```
14 pub fn ExpertIsOnDuty(&mut self, ex: Expert) -> Set<Period> {
15     return self.schedule.domain().set_compr(|p: Period| -> bool {
16                                                 self.schedule.get(&p)
17                                                         .in_set(&ex)
18                                             },
19                                              |p: Period| -> Period {
20                                                  p.clone()
21                                             });
22 }
```

Listing 5.32: Example of Rust output with references enabled.

Implementing references in the code generator

In order to be able to handle types, the intermediate representation of the code generator platform must be modified, in particular every type must specify whether it is a reference or not. This enables methods and other constructs to express whether a reference type is expected, and expressions to specify whether their value is a reference.
To take advantage of these capabilities, all functions and operations should be modified to take non-class and non-primitive parameters by reference. The by-reference semantics of VDM classes require explicit reference counting and dynamic borrow checking, which are not treated here. The type of expressions in the intermediate representation should be corrected to account for types being a reference or not. Most expressions create values, not references, but identifier- and field expressions take special care since their type depends on the value they refer to. This allows `clone()` calls to be removed when values are passed by reference.
To create syntactically correct Rust code, the code generator backend, introduced in chapter 3, should be updated. In particular formal parameters and argument passing involving references requires a different syntax as seen in Listing 5.33.

```
1  fn by_ref(p: &Point) { /* do stuff */ }
2
3  fn by_val(p: Point) { /* do stuff */ }
4
5  fn main() {
6      // Instantiate some struct
7      let point = Point { x: 1.3, y: 3.7 };
8
9    // Perform two function calls
10     by_val(point.clone());
11     by_ref(&point);
12 }
```

Listing 5.33: Example of Rust reference syntax.

Chapter 6

Specification Constructs

This section discusses the specification constructs; pre- and postconditions and type invariants. This chapter builds on the basic understanding of Rust described in chapter 2, and the translations from VDM to Rust described in chapter 4.

6.1. Introduction

One of the major strengths of the VDM-languages is the ability to specify pre- and postconditions for functions and operations, as well as type invariants. This chapter presents translations of the specification constructs, and discusses issues of the translations proposed. The VDM++ language does not specify semantics for pre- and postconditions; the translations presented in this chapter is based on the definitions for pre- and postconditions defined by VDM-SL.

6.2. Definitions in VDM-SL

The VDM-SL language specifies the semantics for pre- and postconditions [30, 35]. Given a a function `f()` as defined in listing 6.1 using the VDM-SL dialect, the interpreter will create precondition functions as shown in listing 6.2.

```
1 f:I1 *...*IK -> R
2 f(i1,...,ik) == e1(i1,...,ik)
3 pre e2(i1,...,ik)
4 post e3(i1,...,ik,RESULT);
```

Listing 6.1: User defined function `f()`.

```
1 pre_f:I1 *...*IK +> bool
2 pre_f(i1,...,ik) == e2(i1,...,ik)
3
4 post_f:I1*...*IK*R +> bool
5 post_f(i1,...,ik,RESULT) == e3(i1 ,...,ik ,RESULT)
```

Listing 6.2: Pre- and postcondition functions for `f()`.

Similarly, given an operation `op()` as defined in listing 6.3, the corresponding pre- and postconditions are defined as in listing 6.4.

```
1 op:I1 *...*IK ==> R
2 op(i1,...,ik) == e1(i1,...,ik)
3 pre e2(i1,...,ik,s)
4 post e3(i1,...,ik,RESULT,s~,s);
```

Listing 6.3: User defined operation `op()`.

```
1 pre_op:I1 *...*IK*S +> bool
2 pre_op(i1,...,ik,s) == e2(i1,...,ik,s)
3
4 post_op:I1*...*IK*R,S,S +> bool
5 post_op(i1,...,ik,RESULT,s~,s) == e3(i1 ,...,ik ,RESULT,s~,s)
```

Listing 6.4: Pre- and postcondition functions for `op()`.

In VDM++, signatures for the pre- and postconditions are not defined, however, the VDM-SL definitions shown in this section will serve as a base for the definitions in VDM++.

6.3. Preconditions

Translating preconditions for functions can be done as outlined in section 6.2. To insert the precondition check, the function needs to be converted to a static operation, to allow the insertion of an *assert*-statement, since statements are not allowed in the body of a function. Given the function in listing 6.1, the corresponding(omitting the postcondition) Rust version will look as shown in listing 6.5 assuming the defining class is called `ClassT`.

```
1 fn pre_f(i1: I1,...,ik: IK) -> bool {
2   e2(i1,...,ik)
3 }
4
5 fn f(i1: I1,...,ik: IK) -> R {
6   assert!(ClassT::pre_f(i1,...,ik));
7   return e1(i1,...,ik);
8 }
```

Listing 6.5: `f()` and precondition function.

Preconditions for operations in VDM10 have read-only access to class state, static variables and public static variables of other classes. In VDM-SL access to the state is provided through an extra argument to the precondition function. In Rust, read-only access to class state can be represented using a method with an immutable reference to self (**`&self`**). Given the operation defined in listing 6.3, a translated version (omitting the postcondition) would be defined as shown in listing 6.6.

```
1 fn pre_op(&self, i1: I1,...,ik: IK) -> bool {
2   e2(i1,...,ik)
```

```
3 }
4
5 fn op(&mut self, i1: I1,...,ik: IK) -> R {
6   assert!(self.pre_op(i1,...,ik));
7   return e1(i1,...,ik);
8 }
```

Listing 6.6: `op()` and precondition method.

6.4. Postconditions

Translating postconditions for functions can be carried out as outlined in section 6.2. Given a function `f()` defined by the class `ClassT` as shown in listing 6.1, the translation to Rust would yield the result in listing 6.7. The value of the function's expression `e1` is bound to the **`RESULT`** variable in line 6 and the postcondition is evaluated in line 7. VDM defines the keyword **`RESULT`**, which in a postcondition expression refers to the result of the associated operation or function.

```
1 fn post_f(i1: I1,...,ik: IK, RESULT: R) -> bool {
2   e3(i1,...,ik, RESULT)
3 }
4
5 fn f(i1: I1,...,ik: IK) -> R {
6   let RESULT: R = e1(i1,...,ik);
7   assert!(ClassT::post_f(i1,...,ik, RESULT));
8   return RESULT;
9 }
```

Listing 6.7: `f()` and postcondition function.

Evaluating the postcondition for a non-static operation requires access to the current state, and the old state; that is, the state before evaluating the operation. As with the precondition, in Rust the postcondition function is transformed to a method, with immutable access to the state through **`&self`**. Additionally, the old state is provided through the parameter `oldSelf` as a reference. Like all functions and methods in Rust the postcondition expression has access to the current value of **`static`** variables of other classes' modules, corresponding to access to **`values`** and **`static instance variables`**. Given the operation `op()` as defined in listing 6.3, the definition and assertion of the postcondition will be translated as shown in listing 6.8. The translation is similar to that of a postcondition for a function, however the old state of the class instance is preserved as `oldSelf` in line 10, and passed to postcondition function. Furthermore, the body of the operation, `e1`, is moved to another function, to allow capturing the return value in the **`RESULT`** variable.

```
1 fn post_op(&self, i1: I1,...,ik: IK, RESULT: R, oldSelf: &ClassT) -> bool {
2   e3(i1,...,ik, RESULT, oldSelf, self)
3 }
4
5 fn op_implementation(&mut self, i1: I1,...,ik: IK) -> R {
6   return e1(i1,...,ik);
7 }
8
```

```
fn op(&mut self, i1: I1,...,ik: IK) -> R {
  let oldSelf: ClassT = self.clone();
  let RESULT: R = self.op_implementation(i1,...,ik);
  assert!(self.post_f(i1,...,ik, RESULT, &oldSelf));
  return RESULT;
}
```

Listing 6.8: `op()` and postcondition method.

Listing 6.9 shows an example for a translation of a postcondition for a class.

```
class ATM
instance variables
accounts: map token to real

operations
Withdraw: token * real ==> real
Withdraw(id, amount) ==
  ...
post
let accountPre = accounts~(id),
  accountPost = accounts(id)
in
  accountPre = accountPost + amount and
  accountPost = RESULT;

end ATM
```

Listing 6.9: A class representing an ATM defining a `Withdraw()` operation with a postcondition.

```
struct ATM {
  accounts: Map<Token, f64>,
}

impl ATM {
  // other methods and associated functions omitted

  fn post_Withdraw(&self, id: Token, amount: f64, RESULT: f64, oldSelf: &ATM)
  -> bool {
    let accountPre: f64 = oldSelf.accounts.get(id.clone());
    let accountPost: f64 = self.accounts.get(id.clone());

    accountPre == accountPost + amount &&
    accountPost == RESULT
  }

  fn Withdraw_implementation(&mut self, id: Token, amount: f64) -> f64 {
    ...
  }

  fn Withdraw(&mut self, id: Token, amount: f64) -> f64 {
    let oldSelf: ATM = self.clone();
    let RESULT: f64 = self.Withdraw_implementation(id.clone(), amount);
    assert!(self.post_Withdraw(id.clone(), amount, RESULT, &oldSelf));
    RESULT
```

```
26     }
27 }
```

Listing 6.10: Rust translation of `ATM` class defining a `Withdraw()` operation with a postcondition.

When translating **`static`** operations the postcondition is translated to an *associated function* and the old state parameter is omitted. If the operation has no return value, the **`RESULT`** parameter is omitted from the postcondition function.

6.4.1 Limitations and future work

As mentioned in section 6.2, the signature and semantics of the pre- and postcondtions are not defined for the VDM++ dialect. Thus, the solutions outlined in this section may not align with official definitions issued in the future.

Passing old state to the postcondition

Listing 6.8 shows a copy of the current state being obtained in line 10. The copy of the state, `old_self`, contains a copy of the instance variables of **`self`**. References to class instances are not implemented in the work carried out for this thesis, which means that values can be trivially copied. If references were to be included in the translation, the state of the class instance becomes the **`instance variables`** of **`self`** itself, and all objects reachable through references held by **`self`**; a graph of objects. This case is left for future improvement.
Old values of public **`static instance variables`** are not accessible with the current approach. A solution to this would involve defining how to store and pass the old values of **`static instance variables`** to the postcondition operation. This is not treated further in this thesis.

6.5. Argument passing and patterns

Checking both pre- and postconditions relies on being able to pass the arguments to the original function or operation on to the functions implementing those checks. Both VDM and Rust allows pattern matching in the formal parameters, although Rust only allows irrefutable patterns. An example is given in listing 6.11 and translated to Rust in listing 6.12.

```
1 takes_pattern: (int * int) -> int
2 takes_pattern(mk_(x, y)) == ...
3 pre ...;
```

Listing 6.11: Function with tuple pattern in formal parameters.

```
1 fn takes_pattern((x, y): (i64, i64)) -> i64 {
2     // Can't refer to value of type '(i8, i8)'.
3 }
4
5 fn pre_takes_pattern((x, y): (i64, i64)) -> bool { /* ... */ }
```

Listing 6.12: Function with tuple pattern in formal parameters in Rust.

As can be seen from listing 6.12, the original value can only be passed on, when a function receives it using an identifier pattern. Thus, if a function or operation whose formal parameters include a non-identifier pattern, invocations of the original function or operation along with invocations of pre- and postconditions must be wrapped in a another operation as seen in listing 6.13.

```
1 fn wrap_takes_pattern(arg0: (i64, i64)) -> i64 {
2     assert!(ClassT::pre_takes_pattern(arg0));
3     return takes_pattern(arg0);
4 }
```

Listing 6.13: Wrapping the original function to allow passing correct arguments to precondition function.

6.6. Type invariants

This section discusses translation and enforcement of type invariants.

6.6.1 Translating invariants

Invariants for classes are translated to methods on the defining class. Given the following class definition shown in listing 6.14, where `e1` is some expression with access to self, the translation yields the Rust code in listing 6.15.

```
1 class ClassT
2
3 instance variables
4 inv e1(self);
5
6 end ClassT
```

Listing 6.14: Class invariant method.

```
1 impl ClassT {
2   fn inv_ClassT(&self) -> bool {
3     e1(self)
4   }
5 }
```

Listing 6.15: Class invariant method in Rust.

For all other type declarations, the invariant expression is translated to a *free-standing function* - a function declared in the module associated to the class, which is declaring the type in question. Listing 6.16 shows a *named type invariant* in VDM, where `T` is the name of the type, `D` is the domain type, `p` is the pattern for variable bindings, used in the invariant expression `e`. The Rust translation is listed as 6.17.

```
1 types
2 T = D
```

```
3 inv p == e(p)
```

Listing 6.16: A named type invariant.

```
1 type T = D;
2
3 fn inv_T(p: T) -> bool {
4   e(p)
5 }
```

Listing 6.17: A named type invariant in Rust.

Listing 6.18 and the Rust version in listing 6.19 show an example of a *named type invariant* for a 3-tuple.

```
1 RGBColor = nat * nat * nat
2 inv mk_(r,g,b) == r <= 255 and g <= 255 and b <= 255;
```

Listing 6.18: Example of a named type invariant.

```
1 type RGBColor = (u64, u64, u64);
2 fn inv_RGBColor((r, g, b): (u64, u64, u64)) -> bool {
3   r <= 255 && g <= 255 && b <= 255
4 }
```

Listing 6.19: Example of a named type invariant in Rust.

6.6.2 Enforcing invariants

This section presents methods for checking invariants for the available classes of VDM types.

Inserting invariant checks

According to the Jørgensen, Larsen and Leavens [35], five constructs must be considered when enforcing invariants. These are listed below:

- Values returned from **`return`** statements and function values must be checked against the specified result type.
- Parameters of functions and operations must be checked prior to evaluation of the body of the function or operation.
- When a *state designator* is updated, the new value must be checked for violations of defined invariants.
- The value of initialized variables and *parameters*, that is values created using the **`let`** keyword, must be checked against the declared type.
- Values of explicitly typed value definitions must be checked against the declared type.

In VDM languages *state designators* refer to state, which can be updated through value assignment. State designators can be an identifier, a field reference, or a map or sequence reference as defined by the VDM language manual [30]. For VDM++ classes state designators can refer to **`static instance variables`** and **`instance variables`** if the object designator was declared as a formal parameter for the enclosing operation. In addition, state designators can be declared locally using the **`dcl`** keyword.

Checking VDM type invariants

Checking type invariants can be accomplished through the use of an assert-statement. Thus, every invariant check has the form shown in Listing 6.20, where `inv()` is some expression evaluating the invariant for the value `v`.

```
assert!(inv(v));
```

Listing 6.20: Asserting an invariant.

As noted in Section 4.3.5 on union types, the translation from VDM to Rust only covers statically typed VDM, ensuring that the types of all values are known at compile-time and are explicit in the generated code.
The examples in this section are adapted from the paper Jørgensen, Larsen and Leavens [35], substituting ar for JML-annotated Java.

Invariants for basic types
The VDM code in Listing 6.21 uses type annotation to ensure `amount` is a valid value.

```
let amount : nat1 = expense - profit in ...
```

Listing 6.21: Explicit type annotation ensures that `amount` is a natural number with a value of at least 1.

Translating Listing 6.21 to Rust will produce the code in Listing 6.22, where `inv_nat1` is a function of the type `nat1 -> bool` to be provided by the Rust code generation runtime library.

```
1 let amount : u64 = expense - profit;
2 assert!(inv_nat1(amount));
3 ...
```

Listing 6.22: Invariant check for basic types in Rust.

The VDM types **`seq1`** and **`inmap`** and invariants defined on records and any named type invariant are translated in a similar fashion. For classes the invariant expression is defined as an instance method, but the check can be performed in similar fashion.

Invariants for union types and optional types
Checking the invariant for a union type is required, if any of the contained types has defined an invariant, as is the case in the example in Listing 6.23.

```
1 types
2   Id = seq1 of char | Token;
```

```
...
withdraw: Id * nat ==> ()
withdraw(accId, amount) == ...;
```

Listing 6.23: Union type with an invariant.

The invariant check might be generated in Rust using pattern matching to match the types with defined invariants as shown in Listing 6.24. Since optional types are translated using the enumerated type `Option<T>`, optional values are a special case of union type, and the invariant check can be translated in a similar way.

```
enum Id {
  Ch0(Seq<char>),
  Ch1(Token),
}
// trait implementation omitted.

fn withdraw(accId: Id, amount: u64) -> () {
  assert!(match accId {
  Ch0(val) => inv_seq1(val),
  _ => true,
  });
  ...
}
```

Listing 6.24: Union type with an invariant in Rust.

Invariants for tuple types
Similarly to union-, and optional types, checking invariants for tuple types is required if any of the values contained has a type declaring an invariant.

```
getStatus: () ==> bool * inmap char to char
getStatus == (
  ...
  return status;
);
```

Listing 6.25: Tuple type with an invariant.

The VDM example in Listing 6.25 may be translated as the Rust code in Listing 6.26.

```
fn getStatus() -> (bool, Map<char, char>) {
  ...
  return {
    assert!(inv_inmap(status.1));
    status
  };
}
```

Listing 6.26: Tuple type with an invariant in Rust.

Invariants for collection types
Checking the invariant for collection types amounts to checking the invariant for each element of the collection.

```
values
  IDs: seq of nat1 = ...;
```

Listing 6.27: A **seq** containing elements with an invariant defined.

The example in Listing 6.27 can be translated to Rust as shown in Listing 6.28. Checking the invariant for elements of **set** and **map** types can be achieved in a similar way.

```
lazy_static! {
  static ref IDs: Seq<u64> = {
    let values: Seq<u64> = ...;
    for value in values {
      assert!(inv_nat1(value));
    }
    values
  };
}
```

Listing 6.28: A **seq** containing elements with an invariant defined.

Atomic blocks

Special care must be taken, when state designators are updated inside a *multiple assigment* block denoted by the **atomic** keyword. Listing 6.29 gives an example of a multiple assignment block.

```
atomic
(
  sd1 := exp1;
  ...
  sdn := expn;
)
```

Listing 6.29: An **atomic** block.

The **atomic** block is evaluated as shown in listing 6.30.

```
let t1: T1 = exp1,
  ...
  tn: Tn = exp1
in
(
  -- Turn off invariants
  sd1 := t1;
  ...
  sdn := tn;
  -- Turn on invariants
  -- Check invariants hold
)
```

Listing 6.30: Evaluation of an **atomic** block.

atomic blocks are handled by keeping track of which state designators are updated in the block, and then inserting appropriate check statements at the end of the block as presented in Section 6.6.2.

Complex state designators

If a state designator refers to a composite type, then it is a *complex state designator*. The VDM model in Listing 6.31 introduces three record type definitions which are nested. The invariants of `R1` and `R2` depend on a field in `R3`, hence, altering the state of the nested `R3` value may violate the invariant of `R1` and `R2`.

```
types
  R1 :: r2 : R2
  inv r1 == r1.r2.r3.x <> -1;

  R2 :: r3 : R3
  inv r2 == r2.r3.x <> -2;

  R3 :: x : int
  inv r3 == r3.x <> -3;

operations
  op: () ==> ()
  op() == (
    dcl r1 : R1 := mk_R1(mk_R2(mk_R3(5)));
    r1.r2.r3.x := -1;
  );
```

Listing 6.31: Record nesting in VDM.

Listing 6.32 shows the Rust translation of the VDM model in Listing 6.31. Note, that mutating the state of the nested field `x` necessitates invariant checking for the enclosing types.

```
// trait implementations and implementations of new() functions omitted.

struct R1 { pub r2: R2 }
fn inv_R1(r1: R1) -> bool {
  r1.r2.r3.x != -1
}

struct R2 { pub r3: R3 }
fn inv_R2(r1: R2) -> bool {
  r2.r3.x != -2
}

struct R3 { pub x: i64 }
fn inv_R3(r3: R3) -> bool {
  r3.x != -3
}
...

fn op() -> () {
  let mut R1: R1 = R1::new(R2::new(R3::new(5)));
  r1.r2.r3.x = -1;

  assert!(inv_R3(r1.r2.r3));
```

```
24     assert!(inv_R2(r1.r2));
25     assert!(inv_R1(r1));
26 }
```

Listing 6.32: Struct nesting in Rust.

Chapter 7

Case study: The Rover Model

This chapter presents a VDM-RT model, a rewritten version using VDM++ as well as guidelines for writing VDM++ models suitable for automated translation to Rust. In addition this chapter presents issues with the translation currently implemented. This chapter builds on the basic understanding of Rust described in Chapter 2, and the translations from VDM to Rust described in Chapters 4 and 6.

7.1. Introduction

This chapter presents a case study for the VDM++ to Rust code generator. This case study was used to validate the choices made when designing and implementing the VDM++ to Rust code generator. Furthermore, this case study has served to study how a VDM++ model might be structured in order to facilitate automated translation.
The purpose and structure of the original VDM-RT model, and the changes made to convert the original VDM-RT model made for simulation purposes into a translatable VDM++ model are presented in section 7.2. The guideline for writing a VDM++ model, which translates to efficient Rust is presented in Section 7.3.
Finally, section 7.4 describes issues with the current translation from VDM to Rust. Issues discussed include translation of concurrency, real-time constructs and numeric conversions. In addition it is discussed how non-deterministic dynamic allocation might be avoided.
The following VDM constructs are *covered* by this case study:

- Classes with **`instance variables`** and **`values`**
- Custom constructor
- Functions and operations
- Preconditions
- Type invariant evaluation function generation
- Named types
- Union type declaration and conversions
- Using the **`cases`** expression to pattern match union types
- Optional types
- Numeric calculations and conversions
- Use of VDM standard library classes `IO` and `MATH`
- Record types; field access and update syntax

- Enumeration of **set**, **seq** and **map**
- Operations on **set**, **seq** and **map**
- Quantified expressions (**forall** and **exists**)
- **map** comprehension
- **set** and index loops
- **if**-expressions and statements

The following VDM constructs from the original model are *not covered* by this case study:

- Type invariant checks are not generated, however, for each type defining an invariant, a function for evaluation of the invariant is generated.
- Concurrency; threads and *permission predicates*
- Real-time constructs; periodic threads, **time**-expression
- Distributed system constructs; **system**, `CPU` and `BUS`

Specification constructs are covered in Chapter 6. A discussion on translating concurrency constructs can be found in Chapter 8.

7.2. The Rover model

The Rover model is a VDM-RT model developed to model and simulate the control system for the six drivetrains of a planetary rover vehicle. It models the drivetrains, as well as a selection of manoeuvres - *gaits* - supported by the rover. The model is the *Descrete-Event* part of a model built for *co-modelling* and *co-simulation* [36] by ESA-ESTEC as part of an *Industry Follow Group* challenge [14, 15]. The rover as modelled can be seen on Figure 7.1.

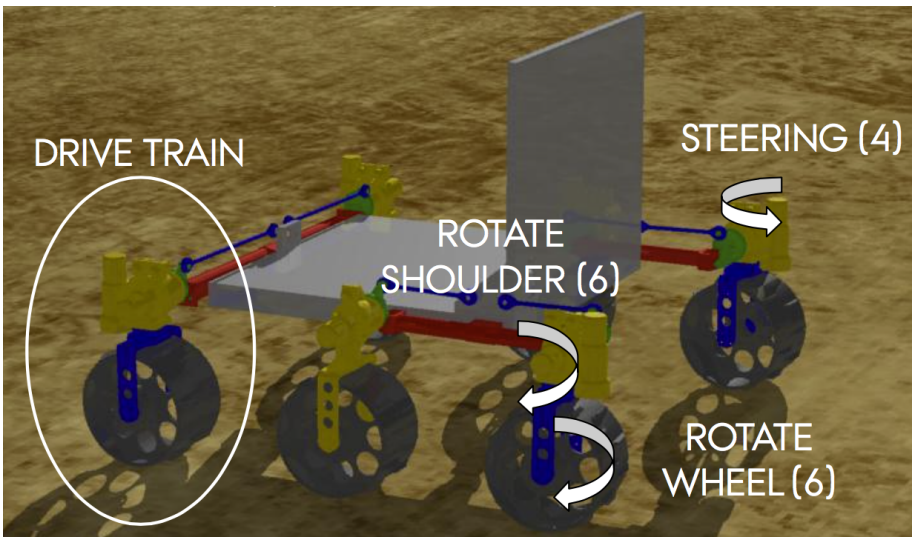

Figure 7.1: ESA-ESTEC Mars rover. Figure from [14].

A class diagram describing the original VDM-RT model can be seen on Figure 7.2.
The class `RobotController` holds the current state of the controller using instance variables for keeping track of beam- and wheel velocities, as well as desired and measured steering angles. Additionally, the `RobotController` class keeps a `plan`, which is a `CompositeAction`

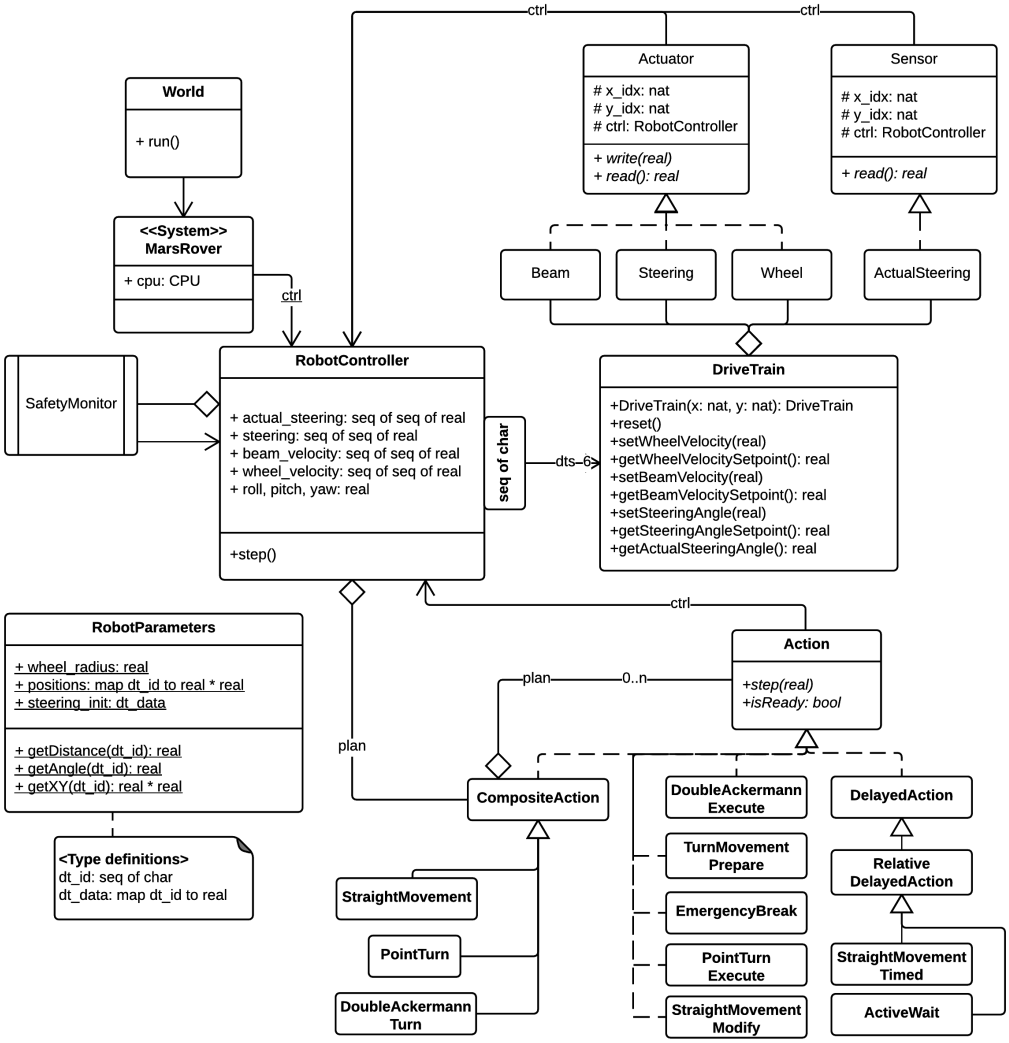

Figure 7.2: Class diagram for the original Rover model.

holding a **`seq`** of `Actions` to be executed. Finally, the class defines an operation, `step()`, to perform a single step of the control loop. The `step()` operation is executed periodically using the **`periodic`** thread feature of VDM-RT.

The drive train structure is modelled by the classes in the top-right corner of the class diagram in Figure 7.2. Each instance of the `DriveTrain` class represents one of the six drivetrains of the rover, and has the `Actuators` `Beam`, `Steering` and `Wheel`, as well as the `Sensor` `ActualSteering`. The operations defined by these classes have access to the state of the controller through a reference to the single instance of `RobotController`.

The actions implemented by the model are represented using a class hierarchy with the class `Action` as the root. Each `Action` has the operations `step()` and `isReady()`. The class `CompositeAction` implements an action as a sequence of actions. The `RobotController` class holds an instance of `CompositeAction` to hold the plan, representing the sequence of actions to be executed by the model. In addition, the three *gaits* modelled by the model are represented as instances of `CompositeAction`. The defined actions modify the state of the controller by accessing the drivetrain mapping `dts`, which is held by the `RobotController` instance.

In addition to the drivetrain and actions, the model defines the active class `SafetyMonitor`. A single instance of the `SafetyMonitor` class is instantiated by the `RobotController` class. The `SafetyMonitor` defines a single operation, `check()`, which monitors the `roll` variable of the `RobotController` periodically, running concurrently with the controller thread.

7.3. Enabling an efficient Rust translation

In order to translate the rover VDM-RT model to Rust, it was first converted to a VDM++ model as required by the current translation implemented by the code generator. This was achieved by removing the **`system`** class `MarsRover`, **`thread`**-declarations and the VDM-RT directives **`cycles`** and **`duration`**, and replacing occurences of the **`time`**-expression with an explicitly declared variable. As part of this, the `SafetyMonitor` class was removed, since it relies on concurrency constructs.

The VDM-RT model of the rover presented in the Section 7.2 relies heavily on inheritance and polymorphism, features which have not been translated as part of this thesis, and thus have no translation yet. In addition the model makes heavy use of reference aliasing, which may lead to complex and inefficient Rust code as discussed in Section 4.3.4. As a result, multiple references to a class instance are not allowed in VDM models used as input for translation to Rust. Mutable static instance variables are not supported by the code generator for similar reasons. To meet these criteria, the model has been rewritten as shown in Figure 7.3.

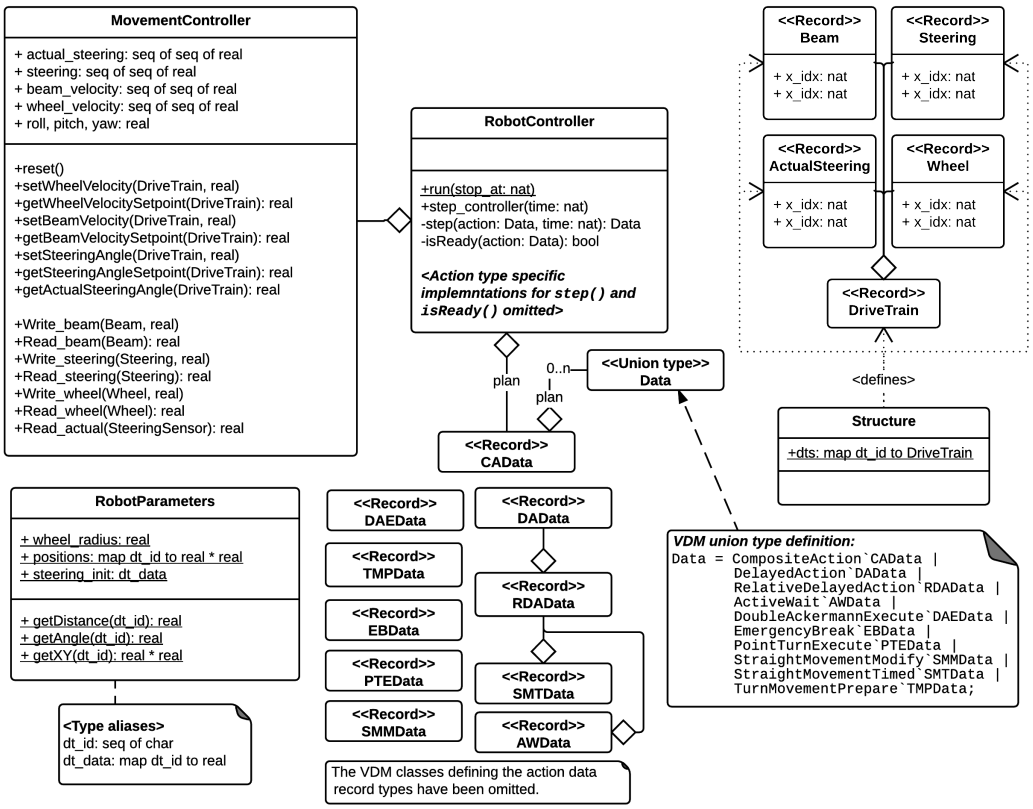

Figure 7.3: Class diagram for the rewritten Rover model.

In the rewritten model, the state of the actuators and sensor of the original model has been moved to the class `MovementController` along with the operations previously defined on the classes modelling the drivetrain in the original model. The `RobotController` class now holds an instance of the `MovementController` and defines the implementations of the operations `step()` and `isReady()` for each type of `Action`. These operations have been moved closer to the state they mutate as a consequence of the fact that passing values by references are unsupported by the code generator. Because of this limitation operations can only mutate the state of local values, and instance variables of the enclosing class.

As discussed in Section 4.3.4, keeping multiple references to the same instance is not supported. Hence, to preserve semantics between VDM and Rust, the classes modelling the drivetrain and actions in the original model, has been converted to record types. The classes representing the drivetrain in the original VDM-RT model has been transformed to the type definitions shown in Listing 7.1.

```
types

public StructureItem :: x_idx : nat
                        y_idx : nat;

  public Beam = StructureItem;
  public Steering = StructureItem;
  public SteeringSensor = StructureItem;
  public Wheel = StructureItem;

  public DriveTrain ::  wheel:    Wheel
                        beam:     Beam
                        steering: Steering
                        actual:   SteeringSensor;
```

Listing 7.1: Representing the drivetrain using records types.

The classes implementing the actions in the original model has been reused as modules in the rewritten VDM++ model. The rewritten model represents instances of actions using record types, and the use of polymorphism in the original model has been converted to the use of the union type `Action`Data`, as shown in Figure 7.3. When executing the steps of the model, the original model relies on polymorphism to call the implementation of `step()` corresponding to the action currently executing. When using union types, this process becomes explicit as shown in Listing 7.2. It can also be seen from Listing 7.2, that all implementations of `step()` return an updated instance of `Action`Data`, which is necessary because records are value types, thus changes must be propagated back using the return value.

```
private step: Action`Data * real ==> Action`Data
  step (data, pst) == return
    cases data:
      mk_CompositeAction`CAData(-) -> CAstep(data, pst),
      mk_ActiveWait`AWData(-) -> AWstep(data, pst),
      mk_DelayedAction`DAData(-,-) -> DAstep(data, pst),
      mk_RelativeDelayedAction`RDAData(-,-) -> RDAstep(data, pst),
      mk_DoubleAckermannExecute`DAEData(-,-,-,-) -> DAEstep(data, pst),
      mk_EmergencyBreak`EBData(-) -> EBstep(data, pst),
      mk_PointTurnExecute`PTEData(-,-) -> PTEstep(data, pst),
      mk_StraightMovementModify`SMMData(-,-,-) -> SMMstep(data, pst),
      mk_StraightMovementTimed`SMTData(-,-) -> SMTstep(data, pst),
```

```
      mk_TurnMovementPrepare`TMPData(-,-,-) -> TMPstep(data, pst)
    end;
```

Listing 7.2: Dispatching the `step()`-call using pattern matching.

The `step()` operation is translated to the Rust code shown in Listing 7.3. Some scoping operators were removed from the code listing to improve readability.

```
fn step(&mut self, data: ::Action_mod::Data, pst: F64) -> ::Action_mod::Data {
        return match data.clone() {
            ::Action_mod::Data::Ch1( CAData { plan: _, }) =>
                self.CAstep(CAData::from(data.clone()), pst),
            ::Action_mod::Data::Ch0(AWData { superAction: _, }) =>
                self.AWstep(AWData::from(data.clone()), pst),
            ::Action_mod::Data::Ch2(DAData { stop_at: _,  finished: _, }) =>
                RobotController::DAstep(DAData::from(data.clone()), pst),
            ::Action_mod::Data::Ch6(RDAData { delay: _,  superAction: _, }) =>
                RobotController::RDAstep(RDAData::from(data.clone()), pst),
            ::Action_mod::Data::Ch3(DAEData {
                    last: _,  radius: _,  omax: _,  turn: _,
                }) =>
                self.DAEstep(DAEData::from(data.clone()), pst),
            ::Action_mod::Data::Ch4(EBData { dummy: _, }) =>
                self.EBstep(EBData::from(data.clone()), pst),
            ::Action_mod::Data::Ch5(PTEData { last: _,  angle: _, }) =>
                self.PTEstep(PTEData::from(data.clone()), pst),
            ::Action_mod::Data::Ch7(SMMData { acc: _,  vel: _,  last: _, }) =>
                self.SMMstep(SMMData::from(data.clone()), pst),
            ::Action_mod::Data::Ch8(SMTData { vel: _,  superAction: _, }) =>
                self.SMTstep(SMTData::from(data.clone()), pst),
            ::Action_mod::Data::Ch9(TMPData {
                    last: _,  dt_sps: _,  done: _,
                }) =>
                self.TMPstep(TMPData::from(data.clone()), pst),
        };
    }
```

Listing 7.3: Dispatching the `step()`-call using pattern matching in Rust.

Inheritance in the hierarchy of actions was converted to composition as the example in Listing 7.4 shows. However, the subclasses of `CompositeAction` in the original model - `StraightMovement`, `PointTurn` and `DoubleAckermannTurn` - did not redefine the action logic of their super class or define additional instance variables, and thus was modelled using instances of `CompositeAction`.

```
-- Original model
class ActiveWait
  is subclass of RelativeDelayedAction
...
end ActiveWait

-- Rewritten model
class ActiveWait
types
  public AWData :: superAction: RelativeDelayedAction`RDAData;
...
end ActiveWait
```

Listing 7.4: Replacing inheritance with composition.

The behavior of the original model and and Rust translation was compared, by comparing the text printed when executing a predefined sequence of actions. The outputs were found to be identical. The source for the resulting Rust code can be found in Appendix A. The sources for the original VDM-RT model and the rewritten VDM++ are not publicly available.

Modelling for translation 2 : Structuring VDM++

- Use classes as modules
- Pass data using records
- Model polymorphism using union types
- Model inheritance using composition
- Avoid global state in mutable static instance values

7.4. Open issues

This section discusses issues that might arise from the use of dynamic allocation in the current translation from VDM to Rust and possible solutions.
As mentioned in the introduction to this chapter, concurrency constructs - threads and *permission predicates*, constructs for real-time and distributed systems and generation of invariant checks are not implemented in the code generator. Concurrency and generation of invariant checks are discussed in Chapters 8 and 6, respectively.

7.4.1 Static allocation

As mentioned in the motivation for this thesis in Section 1.3, a possible target application of the work in this thesis is mission-critical, high-performance systems. Many such systems are also real-time systems where non-determinism is an unwanted characteristic.
The current translation to Rust relies on `Vec<T>`, `HashSet<T>` and `HashMap<K,V>` from the Rust standard library [23] to implement the corresponding VDM types **`seq`**, **`set`** and **`map`**, respectively. These types introduce non-determinism through the use of heap allocation. Thus, to reduce non-determinism, allocation and initialization of any VDM collection should be moved to a setup phase before executing the actual computation. This approach does not entirely solve the problem, since any operation on a VDM collection which returns a collection causes a dynamic allocation. The following operations cause dynamic allocation:

- **`seq`**-operations:
  - Enumeration and comprehension
  - Tail
  - Elements
  - Indexes
  - Reverse
  - Concatenation

- **set**-operations:
  - Enumeration and comprehension
  - Union
  - Intersection
  - Difference
- **map**-operations:
  - Enumeration and comprehension
  - Domain and range
  - Merge
  - Override
  - Restrict domain or range
  - Composition
  - Iteration
  - Inverse

As can be seen, iteration over the elements of a **map** is not possible without causing an allocation, since the domain and range operations allocates a new **set**. In addition, the current translation passes maps by value, as specified the VDM language manual [30] causing values to be cloned, which triggers an allocation. Thus, any solution requires the code generator to implement support for passing values by reference.
Removing dynamic allocation requires non-allocating Rust implementations of the corresponding VDM collections, and further requires that collection sizes are known at compile-time. Many of the available operations will not be able to be used when restricted to static allocation.
It is possible to leverage invariants to specify collection sizes at compile-time as shown in Listing 7.5. By analysing the invariant for the type `Counts` the code generator could detect fixed size collections, and emit code for a fixed sized array.

```
types
  Counts = seq of int
  inv cs == len cs = 3;

instance variables
  counts: Counts := [1,2,3];
```

Listing 7.5: Specifying fixed size using an invariant.

In addition to the use of standard collections, **values** defined by a class are mapped to Rust using the macro `lazy_static!` as presented in Section 4.3.4. Values defined using `lazy_static!` are initialized on first access, and the macro uses heap allocation internally. These characteristics introduce non-determinism. A simple solution would be to ensure, that all static values have been accessed during the setup phase of the program execution.

Chapter 8

Open Issue: Concurrency Constructs

This chapter discusses the different approaches to concurrency employed by Rust and VDM, and issues for possible translations. The discussion in this chapter relies on the introduction to Rust in Chapter 2.

8.1. Introduction

Rust is an interesting language in the context of concurrency, because the language guarantees freedom from *data races* using the type system. This guarantee could be used to ensure that generated Rust code is safe, and to provide feedback to the VDM model. Rust defines data races as follows:

- Two or more threads concurrently accessing a location of memory
- One of them is a write
- One of them is unsynchronized

Rust, however, does not prevent race conditions in general, which for instance means that deadlocks are considered *safe* in the context of Rust.
The possibility of exploring concurrency was an important factor in choosing VDM++ as the VDM dialect used as source language.

8.2. Concurrency in VDM++ and Rust

In VDM, *functionality* is protected using *permission predicates* as seen on lines 17 and 18 of Listing 8.1.

```
1 class Buffer
2 instance variables
3   data : [seq of char] := nil
4
5 operations
6   public Put: seq of char ==> ()
```

```
7    Put(newData) ==
8        data := newData;
9
10   public Take: () ==> seq of char
11   Take() ==
12     let oldData = data in (
13       data := nil;
14       return oldData );
15
16 sync
17   per Put => data = nil;
18   per Get => data <> nil;
19   mutex (Put, Get);
20
21 end Buffer
```

Listing 8.1: Synchronization in VDM++.

VDM allows multiple threads to access an instance concurrently only constrained by the permission predicates, that govern the rules for which operation is allowed to execute. If no permission predicate is specified for an operation, it can be executed by multiple threads unsynchronised, causing race conditions.

In Rust, *data* is protected using a synchronization abstraction such as `Mutex<T>` or `RwLock<T>`, which owns and synchronises access to the data. An example of sharing mutable access to data using a `Mutex<T>` is shown in Listing 8.2. The data - an instance of `Buffer` in this case - is wrapped in a mutex providing synchronised mutable access, and the mutex is wrapped in an instance of `Arc<T>`, providing atomic reference counting. The mutex is locked in line 8, providing mutable access to the data. The mutex is unlocked when `data` goes out of scope in line 11.

```
1  let data = Arc::new(Mutex::new(Buffer::new()));
2
3  for _ in 1..5 {
4    // The clone() call increases the reference count
5    let data = data.clone();
6    thread::spawn(move || {
7      //obtain a mutable reference to the data by locking the mutex
8      let mut data = data.lock().unwrap();
9      data.put("msg".to_owned());
10   });
11 }
```

Listing 8.2: Synchronization in Rust.

Using a synchronization abstraction to wrap the shared mutable state is the *only* way to share mutable access to data when using the safe subset of Rust; this way the language enforces that only one thread can have a mutable reference to an instance. Thus Rust enforces explicit synchronization of mutable access to shared state, ie; the shared state must be owned by a `Mutex<T>`, and every access to shared data must go through the mutex owning the shared data. By contrast, in VDM synchronization is implemented as part of the shared data type, as a consequence the synchronization is implicit when accessing shared data; that is the consumer code does not explicitly handle synchronization.

Because of this difference in synchronisation strategy the VDM approach is not trivially trans-

latable to Rust. Instead of relying entirely on translation, it may be possible to provide a special VDM type encapsulating synchronization in a way, that is compatible with the Rust approach. This approach would provide a simpler translation, but impose restrictions on the VDM model.

8.3. Concurrency in `unsafe` Rust

Using **unsafe** features of Rust, make it possible to provide unsynchronised mutable access to shared state, mimicking the VDM approach. This approach side-steps the safety constraints enforced by the Rust compiler, thus waiving the guarantee of freedom of data races, which is undesirable, however it allows for a simpler translation.
Unsynchronized mutable access is achieved by wrapping the shared state in an instance of `UnsafeCell<T>` to provide *internal mutability*; that is, obtain a mutable reference to the shared state from the immutable reference provided by reference counting type `Arc<T>`. The `Mutable<T>` type in Listing 8.3 provides internal mutability, specifically using the `get()` method. `Mutable<T>` must explicitly implement the unsafe trait `Sync`, causing `Send` to be automatically implemented, which is required for transferring values across thread boundaries as explained in Section 2.3.

```
1  struct Mutable<T>(UnsafeCell<T>);
2
3  unsafe impl<T> Sync for Mutable<T> {}
4
5  impl<T> Mutable<T> {
6    fn get(&self) -> &mut T {
7      unsafe { &mut *self.0.get() }
8    }
9
10   /* constructor function omitted */
11 }
```

Listing 8.3: Providing internal mutability using `UnsafeCell<T>`.

Using the `Mutable<T>` type, the synchronisation example in Listing 8.2 can be rewritten to allow unsynchronised mutable access as shown in Listing 8.4. In this scenario the `Buffer` class is responsible for implementing synchronisation internally. If the `Buffer` class does not implement synchronization, a risk of a data race exists, but the Rust compiler will be unable to detect that risk and provide feedback to the developer.

```
1  let data = Arc::new(Mutable::new(Buffer::new()));
2
3  for _ in 1..5 {
4    // The clone() call increases the reference count
5    let data = data.clone();
6    thread::spawn(move || {
7      //obtain a mutable reference to the data through the Mutable<T> wrapper.
8      let mut data = data.get();
9      data.put("msg".to_owned());
10   });
11 }
```

Listing 8.4: Unsynchronized mutable access in Rust.

Chapter 9

Concluding Remarks

Chapter 1 presented the background, hypothesis, goals and approach of this thesis. The goals are evaluated in this Chapter.

9.1. Introduction

This thesis demonstrates a subset of VDM constructs, large enough to support non-trivial models, that can be translated to efficient Rust code. This chapter will revisit and evaluate the goals of this thesis in Sections 9.2 and 9.3, respectively. Section 9.4 will then discuss the contributions made by this thesis project. Issues requiring further research are discussed in Section 9.5. Section 9.6 will highlight and discuss possible enhancements or extensions, which could be subject to future work. Finally, Section 9.7 contains the final remarks.

9.2. Revisiting the Goals of the Thesis

Recall that the goals of this thesis project are:

Goal 1: Propose translations from VDM++ to efficient Rust.

Goal 2: Develop a proof-of-concept VDM++ to Rust code generator based on the code generator platform in the Overture platform. This code generator must be based on the findings from Goal 1.

Goal 3: The translation produced by the prototype must by validated by evaluating various case studies.

Goal 4: Improve personal problem solving skills, with regards to identifying, analysing and describing problems and their impact, and reviewing of possible solutions.

9.3. Evaluation of the Achievement of the Goals

All of the goals stated were satisfied to varying degrees. Each goal will be evaluated separately in the following sections.

9.3.1 Goal 1: Propose translations from VDM++ to efficient Rust

Translations for the core VDM constructs has been proposed and presented in Chapter 4. Parts of the core VDM++ language, notably class inheritance and polymorphism has not been translated. Translations for generating and enforcing the specification constructs, pre-, postcondtions and invariants were discussed in Chapter 6. Issues when translating concurrency constructs have been considered in Chapter 8, but no translations have been proposed.

9.3.2 Goal 2: Develop a proof-of-concept code generator

The work carried out during the thesis resulted in a code generator, which translated VDM code as described in Chapter 4. A code generator project relying on the Overture code generator platform was created, which was able to generate code for two non-trivial case studies. Developing the code generator involved implementing transformations of the Intermediate Representation to fit the Rust language, in addition to the transformations from the Overture platform which were reusable.

9.3.3 Goal 3: Validate translation using case studies

Two case studies were used to validate the translation as implemented by the Rust code generator: the Alarm model in Chapter 5 and the planetary Rover presented in Chapter 7. Both case studies highlighted challenges of the translation, but showed that the translation was correct. However, the rover model had to undergo a substantial rewrite, part of which was to convert the model from a simulation focused model, to a control model suitable for translation. However, part of the rewrite was necessitated by the lack of support for passing values by reference, which is a shortcoming of the current implementation of the code generator.

9.3.4 Goal 4: Improve personal problem solving skills

All work carried out during this thesis project followed the basic strategy; identify and analyse the problem, find possible solutions and select a solution based on pros and cons of the found solutions. In particular, this was the approach employed when mapping constructs from VDM to Rust, and is documented throughout the thesis, problems encountered has been described, and solutions has been found and selected after careful deliberation.

9.4. Contributions

9.4.1 Translation of VDM++ to Rust

The main contribution of this thesis is a partial translation from VDM++ to Rust. The following list is an overview of translation status of VDM++ constructs.

- Completely implemented in the code generator:

  - Basic Data types: **bool**, **char**, **token**, quotes, numerics
  - Collections: **seq**, **set** and **map**
    * Enumeration
    * Comprehensions
    * All operations
    * **exists1**, **exists** and **forall** qualifiers
    * **let be .. st** expressions and statements
  - Tuples (Product types)
  - Records (Composite types)
  - Optional types
  - Named types
  - Operations and functions
  - Preconditions
  - Constant value definition
  - Patterns:
    * Pattern identifier
    * Match value
    * Tuple pattern
    * Record pattern
  - **let**- **def**- and **dcl**-statements and expressions
  - **if**-expression and statement
  - **cases**-expression
  - **lambda**-expression
  - loop statements: while-, for- and index-loops
  - **undefined**-expression and **error**-statement

- Partially implemented in the code generator:
  - Classes
  - Numeric conversions
  - Union types
  - VDM standard library: IO and MATH
  - Invariants

- Translation proposed, but not implemented:
  - Invariants
  - Postconditions
  - **cases**-statement

- Not translated:
  - Concurrency
    * **thread**-statement
    * Permission predicates
    * History counters
    * **threadid**-expression
  - Patterns:
    * Refutable patterns as formal parameters
    * Set enumeration and union
    * Sequence enumeration and concatenation
    * Map enumeration
    * Maplet pattern list
    * Object pattern
  - Function types
  - **is**-expressions and class membership expressions
  - Type binds for quantified expressions and comprehensions
  - Nondeterministic statement
  - Exception handling statements

9.4.2 Guidelines for VDM++ translating to efficient Rust

As part of case study presented in Chapter 7, a VDM-RT model modelling the movement of a planetary rover was rewritten to a subset of VDM++, which translates into efficient Rust. A result of this work, was the creation of a set of guidelines describing how a VDM++ model should be structured in order to obtain a model, capable of being translated to Rust.

9.4.3 Feedback to the Overture tool

A large part of the work carried out as part of this thesis, is the implementation of the Rust code generator using the Overture code generation platform. The code base developed during the thesis may serve as resource on implementing code generation for a language using the Overture code generation platform. Technical documentation of the implementation is, however, outside the scope of this thesis.

9.5. Open issues

This section discusses issues discovered during the course of this thesis regarding translation from VDM++ to Rust needing further research.

Translating classes: The translation for VDM++ classes proposed is only partial; inheritance and polymorphism are not supported by the current translation. These shortcomings have no obvious solutions, since Rust has omitted support for classes, but instead relies on *traits* for code reuse and composability.

Translating concurrency constructs: As discussed in Chapter 8, the concurrency models employed by VDM++ and Rust differ significantly, as a result translating concurrency constructs from VDM++ to Rust is non-trivial. If a translation to *safe* Rust was found, the freedom of data races guarantee enforced by the Rust compiler could be useful for determining whether a given VDM model contains data races.

Static allocation: As discussed in Section 7.4, the current translation relies on a multiple translations, which uses dynamic allocation. As dynamic allocation results in non-deterministic run time, this is undesirable when the generated code is part of a real-time system.

Proving correctness: Proving the correctness of the proposed and implemented translations is outside the scope of this thesis, but may be the subject of further research.

9.6. Future work

This section describes work items which were considered to be non-essential to accomplish the goals of this thesis, but nonetheless would be beneficial to explore in the future.

References in the intermediate representation: As discussed in Section 5.5, the current intermediate representation does not support representing references - *borrows* in Rust. As a result, all values in the generated Rust code must be passed by value requiring values to be copied when passed as argument to a function or operation invocation. In addition this

shortcoming prevents passing classes by reference as discussed in Section 7.3. Implementing references amounts to adding support for references in the intermediate representation and rewriting the Rust runtime library implementing builtin VDM types to take advantage of references. In addition, transformations must be implemented to insert proper referencing and dereferencing operators.

Adding support for additional VDM constructs: Field expressions, and operation calls with a union type value as the root object have not been translated or implemented, but should be fairly straight forward to implement using pattern matching. `is`-expressions can be translated to Rust by mapping to methods on the `Any` trait. Translating invariant checks and postconditions has not been implemented in the code generator. A future implementation may be based on the translations presented in Chapter 6.

Benchmarking performance of the generated code: During the work carried out for this thesis, the choices of translations from VDM++ to Rust was motivated by the goal of creating efficient Rust code. However, no benchmarks of the performance of the generated code have been carried out, hence, performing benchmarks would be a straight forward extension to the work presented in this thesis.

Integration into the Overture tool: The code generator created during this thesis was developed using the Overture code generation platform, but separate from the Overture code base. Integrating the Rust code generator in the Overture tool would allow public testing and use. Integration amounts to increasing the level of testing of the code generator and merging the code generator into the official Overture repository.

9.7. Final Remarks

The goals of this thesis have been partially met as discussed in Section 9.3. It is the hope, that the results of this thesis may be used to expand the range of applications for formal modelling, by enabling code generation for a broader range of targets, allowing increases in productivity and level of confidence in systems under development. In addition, it is the hope that the work carried out builds a strong foundation for the effort to integrate Overture with the *TASTE* tool set [12].

Bibliography

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, pp. 1–36, October 2009. [cited at p. 1]

[2] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996. [cited at p. 1]

[3] 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. 2, 4, 46]

[4] 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. 2]

[5] N. Plat and H. Toetenel, "A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax," Tech. Rep. 92-07, Delft University, March 1992. [cited at p. 2]

[6] P. G. Larsen, J. Fitzgerald, and S. Wolff, "Methods for the Development of Distributed Real-Time Embedded Systems using VDM," *Intl. Journal of Software and Informatics*, vol. 3, October 2009. [cited at p. 2]

[7] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: advances in support for formal modeling in VDM," Submitted for publication 2007. [cited at p. 2]

[8] Overture-Core-Team, "Overture Web site." http://www.overturetool.org, 2007. [cited at p. 2, 17, 46]

[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. 2, 17, 18, 19]

[10] "The rust website," 2015. `https://www.rust-lang.org`. [cited at p. 2, 9, 11]

[11] "Rust's entry on llvm projects list," 2015. `http://llvm.org/ProjectsWithLLVM/#rust`. [cited at p. 2]

[12] M. Verhoef and M. Perrotin, "TASTE for Overture to keep SLIM," pp. 132–139, June 2015. GRACE-TR-2015-06. [cited at p. 3, 89]

[13] E. Conquet, M. Perrotin, P. Dissaux, T. Tsiodras, and J. Hugues, "The taste toolset: turning human designed heterogeneous systems into computer built homogeneous software," in *European Congress on Embedded Real-Time Software (ERTS 2010)*, (Toulouse, France), May 2010. [cited at p. 3]

[14] J. F. Peter Gorm Larsen, Marcel Verhoef, "Future of formal methods." `http://www.ipa.go.jp/files/000005451.pdf`, October 2012. [cited at p. 5, 74]

[15] "Destecs industry follow group challenges," 2015. `http://www.destecs.org/ifgchallenges.html`. [cited at p. 5, 74]

[16] "The rust reference," 2015. `https://doc.rust-lang.org/reference.html`. [cited at p. 9]

[17] B. Stroustrup, *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. [cited at p. 9]

[18] N. Shärli, S. Ducasse, O. Nierstrasz, and A. Black, "Traits: Composable units of behavior," tech. rep., 2002. [cited at p. 9]

[19] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, (New York, NY, USA), pp. 60–76, ACM, 1989. [cited at p. 9]

[20] M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and computation*, vol. 132, no. 2, pp. 109–176, 1997. [cited at p. 9]

[21] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), pp. 282–293, ACM, 2002. [cited at p. 9]

[22] E. Reed, "Patina: A formalization of the rust programming language," February 2015. [cited at p. 9]

[23] "The rust programming language," 2015. `https://doc.rust-lang.org/book/`. [cited at p. 9, 79]

[24] R. M. Stallman and Z. Weinberg, "The c preprocessor," *Free Software Foundation*, 1987. [cited at p. 15]

[25] P. W. V. Jørgensen and P. G. Larsen, "Towards an Overture Code Generator," in *The Overture 2013 workshop*, August 2013. [cited at p. 17]

[26] 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. 18]

[27] 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. 18]

[28] "The Apache Velocity website," 2015. `http://velocity.apache.org/`. [cited at p. 20]

[29] "The cargo documentation," 2015. `http://doc.crates.io`. [cited at p. 23]

[30] P. G. Larsen, K. Lausdahl, N. Battle, J. Fitzgerald, S. Wolff, and S. Sahara, "VDM-10 Language Manual," Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org, April 2013. [cited at p. 24, 58, 61, 68, 80]

[31] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, pp. 5–48, Mar. 1991. [cited at p. 25]

[32] L. P. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Communications of the ACM*, vol. 19, no. 9, pp. 522–526, 1976. [cited at p. 30]

[33] Y. Levanoni and E. Petrank, "An on-the-fly reference counting garbage collector for java," *SIGPLAN Not.*, vol. 36, pp. 367–380, Oct. 2001. [cited at p. 30]

[34] B. Stroustrup, "C++11 style - a touch of class," February 2012. `https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style`. [cited at p. 59]

[35] P. W. V. Tran-Jørgensen, P. G. Larsen, and G. Leavens, "Automated translation of vdm to jml annotated java," *International Journal on Software Tools for Technology Transfer*, <not published yet>. [cited at p. 61, 67, 68]

[36] J. Fitzgerald, P. G. Larsen, and M. Verhoef, eds., *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014. [cited at p. 74]

Appendices

Appendix A

The Rover Rust Code

This is appendix contains the Rust code which was the result of translating the rewritten VDM++ model described in Chapter 7. This code was created using the Rust code generator developed during this thesis.

A.1. Rust code

A.1.1 Structure

```
/* use declarations omitted */

// types
pub type DriveTrainType = Map<::RobotParameters_mod::dt_id, DriveTrain>;
pub fn inv_DriveTrainType(dtt: Map<::RobotParameters_mod::dt_id, DriveTrain>) -> bool {
    dtt.domain().forall(|dtname: ::RobotParameters_mod::dt_id| -> bool {
        ::RobotParameters_mod::positions.domain().in_set(dtname.clone())
    })
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct StructureItem {
    pub x_idx: u64,
    pub y_idx: u64,
}
impl_record! { StructureItem:
    x_idx as u64,
    y_idx as u64
}

pub type Beam = StructureItem;
pub type Steering = StructureItem;
pub type SteeringSensor = StructureItem;
pub type Wheel = StructureItem;

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct DriveTrain {
    pub wheel: Wheel,
    pub beam: Beam,
    pub steering: Steering,
    pub actual: SteeringSensor,
}
impl_record! { DriveTrain:
    wheel as Wheel,
    beam as Beam,
    steering as Steering,
    actual as SteeringSensor
}

// values
```

```
lazy_static! {
  pub static ref MAX_STEERING: F64 = ((F64::from(70) / F64::from(180)) * MATH::pi);
  pub static ref dts: DriveTrainType = map!(
        strseq!("lf") => Structure::create_DriveTrain(1, 1),
        strseq!("rf") => Structure::create_DriveTrain(2, 1),
        strseq!("lm") => Structure::create_DriveTrain(1, 2),
        strseq!("rm") => Structure::create_DriveTrain(2, 2),
        strseq!("lr") => Structure::create_DriveTrain(1, 3),
        strseq!("rr") => Structure::create_DriveTrain(2, 3)
    );
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct Structure;
impl Structure {
    // operations
    pub fn new() -> Structure {
        let instance: Structure = Structure::default();
        return instance;
    }

    // functions
    fn create_DriveTrain(x: u64, y: u64) -> DriveTrain {
        DriveTrain::new(StructureItem::new(x, y),
                        StructureItem::new(x, y),
                        StructureItem::new(x, y),
                        StructureItem::new(x, y))
    }
}

impl Default for Structure {
    fn default() -> Structure {
        Structure
    }
}
```

Listing A.1: Structure class

A.1.2 RobotParameters

```
/* use declarations omitted */

// types
pub type dt_id = Seq<char>;
pub fn inv_dt_id(dti: Seq<char>) -> bool {
    ::RobotParameters_mod::positions.domain().in_set(dti.clone())
}

pub type dt_data = Map<dt_id, F64>;
pub fn inv_dt_data(dtd: Map<dt_id, F64>) -> bool {
    dtd.domain().card() == ::RobotParameters_mod::positions.domain().card()
}

// values
lazy_static! {
  pub static ref wheel_radius: F64 = F64(0.07);

  pub static ref positions: Map<Seq<char>,(F64, F64)> = map!(
        strseq!("lf") => (F64(0.3215), F64(2.23252)),
        strseq!("rf") => (F64(0.3215), F64(0.90907)),
        strseq!("lm") => (F64(0.199), MATH::pi),
        strseq!("rm") => (F64(0.199), F64(0.0)),
        strseq!("lr") => (F64(0.3215), -F64(2.23252)),
        strseq!("rr") => (F64(0.3215), -F64(0.90907))
    );

  pub static ref steering_init: Map<Seq<char>,F64> = map!(
        strseq!("lf") => F64(0.0), strseq!("rf") => F64(0.0),
        strseq!("lm") => F64(0.0), strseq!("rm") => F64(0.0),
        strseq!("lr") => F64(0.0), strseq!("rr") => F64(0.0)
    );
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct RobotParameters;
```

```
impl RobotParameters {
    // operations
    pub fn new() -> RobotParameters {
        let instance: RobotParameters = RobotParameters::default();
        return instance;
    }

    // functions
    pub fn getDistance(pdtid: dt_id) -> F64 {
        {
            let (res, _): (F64, F64) = {
                let arg0: dt_id = pdtid.clone();
                ::RobotParameters_mod::positions.get(arg0.clone())
            };
            res
        }
    }

    pub fn getAngle(pdtid: dt_id) -> F64 {
        {
            let (_, res): (F64, F64) = {
                let arg0: dt_id = pdtid.clone();
                ::RobotParameters_mod::positions.get(arg0.clone())
            };
            res
        }
    }

    pub fn getXY(pdtid: dt_id) -> (F64, F64) {
        {
            let (r, alpha): (F64, F64) = {
                let arg0: dt_id = pdtid.clone();
                ::RobotParameters_mod::positions.get(arg0.clone())
            };
            ((r * MATH::cos(alpha)), (r * MATH::sin(alpha)))
        }
    }
}

impl Default for RobotParameters {
    fn default() -> RobotParameters {
        RobotParameters
    }
}
```

Listing A.2: RobotParameters class

A.1.3 MovementController

```
/* use declarations omitted */

// values
lazy_static! {
   static ref zeroes: Seq<F64> = seq!(F64::from(0), F64::from(0), F64::from(0));
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct MovementController {
    pub actual_steering: Seq<Seq<F64>>,
    pub steering: Seq<Seq<F64>>,
    pub beam_velocity: Seq<Seq<F64>>,
    pub wheel_velocity: Seq<Seq<F64>>,
}

impl MovementController {
    // operations
    pub fn Write_beam(&mut self, beam: ::Structure_mod::Beam, pval: F64) -> () {
        self.beam_velocity.get_ref(beam.x_idx).put(beam.y_idx, pval);
    }

    pub fn Read_beam(&mut self, beam: ::Structure_mod::Beam) -> F64 {
        return self.beam_velocity.get(beam.x_idx).get(beam.y_idx);
    }

    pub fn Write_steering(&mut self, steer: ::Structure_mod::Steering, pval: F64) -> () {
```

```
27          assert!(self.pre_Write_steering(steer.clone(), pval));
28          self.steering.get_ref(steer.x_idx).put(steer.y_idx, pval);
29      }
30
31      pub fn Read_steering(&mut self, steer: ::Structure_mod::Steering) -> F64 {
32          return self.steering.get(steer.x_idx).get(steer.y_idx);
33      }
34
35      pub fn Read_actual(&mut self, sensor: ::Structure_mod::SteeringSensor) -> F64 {
36          return self.actual_steering.get(sensor.x_idx).get(sensor.y_idx);
37      }
38
39      pub fn Write_wheel(&mut self, wheel: ::Structure_mod::Wheel, pval: F64) -> () {
40          self.wheel_velocity.get_ref(wheel.x_idx).put(wheel.y_idx, pval);
41      }
42
43      pub fn Read_wheel(&mut self, wheel: ::Structure_mod::Wheel) -> F64 {
44          return self.wheel_velocity.get(wheel.x_idx).get(wheel.y_idx);
45      }
46
47      pub fn reset(&mut self, drvtr: ::Structure_mod::DriveTrain) -> () {
48          self.setWheelVelocity(drvtr.clone(), F64(0.0));
49          self.setBeamVelocity(drvtr.clone(), F64(0.0));
50          self.setSteeringAngle(drvtr.clone(), F64(0.0));
51      }
52
53      pub fn setWheelVelocity(&mut self, drvtr: ::Structure_mod::DriveTrain, pwv: F64) -> () {
54          self.Write_wheel(drvtr.wheel.clone(),
55                           (pwv / ::RobotParameters_mod::wheel_radius.clone()));
56      }
57
58      pub fn getWheelVelocitySetpoint(&mut self, drvtr: ::Structure_mod::DriveTrain) -> F64 {
59          return (::RobotParameters_mod::wheel_radius.clone() * self.Read_wheel(drvtr.wheel.clone()));
60      }
61
62      pub fn setBeamVelocity(&mut self, drvtr: ::Structure_mod::DriveTrain, pbv: F64) -> () {
63          self.Write_beam(drvtr.beam.clone(), pbv);
64      }
65
66      pub fn getBeamVelocitySetpoint(&mut self, drvtr: ::Structure_mod::DriveTrain) -> F64 {
67          return self.Read_beam(drvtr.beam.clone());
68      }
69
70      pub fn setSteeringAngle(&mut self, drvtr: ::Structure_mod::DriveTrain, psv: F64) -> () {
71          self.Write_steering(drvtr.steering.clone(), psv);
72      }
73
74      pub fn getSteeringAngleSetpoint(&mut self, drvtr: ::Structure_mod::DriveTrain) -> F64 {
75          return self.Read_steering(drvtr.steering.clone());
76      }
77
78      pub fn getActualSteeringAngle(&mut self, drvtr: ::Structure_mod::DriveTrain) -> F64 {
79          return self.Read_actual(drvtr.actual.clone());
80      }
81
82      pub fn new() -> MovementController {
83          let instance: MovementController = MovementController::default();
84          return instance;
85      }
86
87      fn pre_Write_steering(&self, steer: ::Structure_mod::Steering, pval: F64) -> bool {
88          return (pval >= -::Structure_mod::MAX_STEERING.clone()) &&
89                 (pval <= ::Structure_mod::MAX_STEERING.clone());
90      }
91  }
92
93  impl Default for MovementController {
94      fn default() -> MovementController {
95          let actual_steering: Seq<Seq<F64>> = seq!(::MovementController_mod::zeroes.clone(),
96                                                    ::MovementController_mod::zeroes.clone());
97          let steering: Seq<Seq<F64>> = seq!(::MovementController_mod::zeroes.clone(),
98                                             ::MovementController_mod::zeroes.clone());
99          let beam_velocity: Seq<Seq<F64>> = seq!(::MovementController_mod::zeroes.clone(),
100                                                  ::MovementController_mod::zeroes.clone());
101         let wheel_velocity: Seq<Seq<F64>> = seq!(::MovementController_mod::zeroes.clone(),
102                                                  ::MovementController_mod::zeroes.clone());
103
104         MovementController {
105             actual_steering: actual_steering,
106             steering: steering,
107             beam_velocity: beam_velocity,
108             wheel_velocity: wheel_velocity,
```

```
        }
    }
}
```

Listing A.3: MovementController class

A.1.4 RobotController

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct RobotController {
    mctrl: MovementController,
    plan: Option<::Action_mod::Data>,
}

impl RobotController {
    // operations
    pub fn cg_initRobotController_1(&mut self) -> () {
        let mut actions: Seq<::Action_mod::Data> = seq!();
        actions = actions.conc(seq!(::Action_mod::Data::from(ActiveWait::create(F64(1.0)))));
        actions = actions.conc(seq!(::Action_mod::Data::from(StraightMovement::create(F64(0.75),
                                                                                      F64(0.25),
                                                                                      F64(3.5)))));
        actions = actions.conc(seq!(::Action_mod::Data::from(ActiveWait::create(F64(1.0)))));
        actions =
            actions.conc(seq!(::Action_mod::Data::from(StraightMovement::create(-F64(1.25),
                                                                                F64(0.4),
                                                                                F64(3.0)))));
        actions = actions.conc(seq!(::Action_mod::Data::from(PointTurn::create((MATH::pi /
                                                                                F64::from(4))))));
        actions = actions.conc(seq!(::Action_mod::Data::from(StraightMovement::create(F64(1.25),
                                                                                      F64(0.4),
                                                                                      F64(3.5)))));
        actions =
            actions.conc(seq!(::Action_mod::Data::from(DoubleAckermannTurn::create(-F64(0.75),
                                                                                   MATH::pi))));
        actions = actions.conc(seq!(::Action_mod::Data::from(StraightMovement::create(F64(1.0),
                                                                                      F64(0.35),
                                                                                      F64(3.0)))));

        self.plan = Some(::Action_mod::Data::from(CompositeAction::create(actions.clone())));
    }

    pub fn new() -> RobotController {
        let mut instance: RobotController = RobotController::default();
        instance.cg_initRobotController_1();
        return instance;
    }

    pub fn run(&mut self, stopat: u64) -> () {
        let mut ctrl: RobotController = RobotController::new();
        {
            let step_by: u64 = ((::Action_mod::RESOLUTION.clone() / 10) as u64);
            let to: u64 = ((stopat * ::Action_mod::RESOLUTION.clone()) as u64);
            let mut time: u64 = (0 as u64);
            while time <= to {
                ctrl.step_controller(time);
                time = time + step_by;
            }
        }
    }

    pub fn step_controller(&mut self, time: u64) -> () {
        IO::print(strseq!("time = "));
        IO::print((F64::from(time) / F64::from(::Action_mod::RESOLUTION.clone())));
        IO::print(strseq!(" : "));
        if {
            let arg0: Option<::Action_mod::Data> = self.plan.clone();
            self.isReady(arg0.clone().expect("Optional was nil."))
        } {
            IO::println(strseq!("no action"));
        } else {
            self.plan = Some({
```

```
                let arg0: Option<::Action_mod::Data> = self.plan.clone();
                let arg1: u64 = time;
                self.step(arg0.clone().expect("Optional was nil."), F64::from(arg1))
            });
        }
    }

    fn step(&mut self, data: ::Action_mod::Data, pst: F64) -> ::Action_mod::Data {
        return match data.clone() {
            ::Action_mod::Data::Ch1(::CompositeAction_mod::CAData { plan: _, }) => {
                self.CAstep(::CompositeAction_mod::CAData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch0(::ActiveWait_mod::AWData { superAction: _, }) => {
                self.AWstep(::ActiveWait_mod::AWData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch2(::DelayedAction_mod::DAData { stop_at: _,  finished: _, }) => {
                RobotController::DAstep(::DelayedAction_mod::DAData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch6(::RelativeDelayedAction_mod::RDAData {
                    delay: _,
                    superAction: _,
                }) => {
                RobotController::RDAstep(::RelativeDelayedAction_mod::RDAData::from(data.clone()),
                                                                                    pst)
            }
            ::Action_mod::Data::Ch3(::DoubleAckermannExecute_mod::DAEData {
                    last: _,
                    radius: _,
                    omax: _,
                    turn: _,
                }) => {
                self.DAEstep(::DoubleAckermannExecute_mod::DAEData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch4(::EmergencyBreak_mod::EBData { dummy: _, }) => {
                self.EBstep(::EmergencyBreak_mod::EBData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch5(::PointTurnExecute_mod::PTEData { last: _,  angle: _, }) => {
                self.PTEstep(::PointTurnExecute_mod::PTEData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch7(::StraightMovementModify_mod::SMMData {
                    acc: _,
                    vel: _,
                    last: _,
                }) => {
                self.SMMstep(::StraightMovementModify_mod::SMMData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch8(::StraightMovementTimed_mod::SMTData {
                    vel: _,
                    superAction: _,
                }) => {
                self.SMTstep(::StraightMovementTimed_mod::SMTData::from(data.clone()), pst)
            }
            ::Action_mod::Data::Ch9(::TurnMovementPrepare_mod::TMPData {
                    last: _,
                    dt_sps: _,
                    done: _,
                }) => {
                self.TMPstep(::TurnMovementPrepare_mod::TMPData::from(data.clone()), pst)
            }
        };
    }

    fn AWstep(&mut self, data: ::ActiveWait_mod::AWData, pst: F64) -> ::Action_mod::Data {
        IO::print(strseq!("active wait "));
        for dt in ::Structure_mod::dts.range() {
            self.mctrl.setWheelVelocity(dt.clone(), F64(0.0));
        }

        return ::Action_mod::Data::from(
            ::ActiveWait_mod::AWData::new(::RelativeDelayedAction_mod::RDAData::from(
                RobotController::RDAstep(data.superAction.clone(), pst))));
    }

    fn CAstep(&mut self, data: ::CompositeAction_mod::CAData, pt: F64) -> ::Action_mod::Data {
        let mut data: ::CompositeAction_mod::CAData = data.clone();
        if (data.plan.len() > 0) {
            let current: ::Action_mod::Data = data.plan.head();
            {
                {
                    let arg0: u64 = 1;
                    let arg1: ::Action_mod::Data = self.step(current.clone(), pt);
                    data.plan.put(arg0, arg1.clone());
```

```
150                 }
151
152                 if self.isReady(data.plan.get(1)) {
153                     data.plan = data.plan.tail();
154                 }
155             }
156         }
157
158         return ::Action_mod::Data::from(data);
159     }
160
161     fn DAstep(data: ::DelayedAction_mod::DAData, pst: F64) -> ::Action_mod::Data {
162         assert!(RobotController::pre_DAstep(data.clone(), pst));
163         let mut data: ::DelayedAction_mod::DAData = data.clone();
164         data.finished = (pst >= data.stop_at.expect("Optional was nil."));
165         if !(data.finished) {
166             IO::print(strseq!("to go "));
167             IO::print(((data.stop_at.expect("Optional was nil.") - pst) /
168                        F64::from(::Action_mod::RESOLUTION.clone())));
169             IO::println(strseq!(" sec"));
170
171         } else {
172             IO::println(strseq!("done"));
173
174         }
175
176         return ::Action_mod::Data::from(data);
177     }
178
179     fn RDAstep(data: ::RelativeDelayedAction_mod::RDAData, pst: F64) -> ::Action_mod::Data {
180         let mut data: ::RelativeDelayedAction_mod::RDAData = data.clone();
181         if data.superAction.stop_at == None {
182             data.superAction.stop_at = Some((pst + data.delay));
183         }
184
185         return ::Action_mod::Data::from(
186             ::RelativeDelayedAction_mod::RDAData {
187                 superAction: ::DelayedAction_mod::DAData::from(
188                     RobotController::DAstep(data.superAction.clone(), pst))
189                 , .. data.clone()
190             });
191     }
192
193     fn setSpeed(&mut self, data: ::DoubleAckermannExecute_mod::DAEData, psp: F64) -> () {
194         for dtn in ::Structure_mod::dts.domain() {
195             let dtr: F64 = (psp *
196                             {
197                 let arg0: ::RobotParameters_mod::dt_id = dtn.clone();
198                 data.radius.get(arg0.clone())
199             });
200             let dt: ::Structure_mod::DriveTrain = {
201                 let arg0: ::RobotParameters_mod::dt_id = dtn.clone();
202                 ::Structure_mod::dts.get(arg0.clone())
203             };
204             {
205                 self.mctrl.setWheelVelocity(dt.clone(), dtr);
206             }
207         }
208     }
209
210     fn DAEstep(&mut self,
211                data: ::DoubleAckermannExecute_mod::DAEData,
212                pst: F64)
213                -> ::Action_mod::Data {
214         let mut data: ::DoubleAckermannExecute_mod::DAEData = data.clone();
215         if data.last == None {
216             IO::println(strseq!("initiate rotate"));
217         } else {
218             let dt: F64 = ((pst - data.last.expect("Optional was nil.")) /
219                            F64::from(::Action_mod::RESOLUTION.clone()));
220             {
221                 let dto: F64 = (data.omax * dt);
222                 {
223                     let sign: i64 = if (data.turn >= F64::from(0)) {
224                         (1 as i64)
225                     } else {
226                         -1
227                     };
228                     {
229                         let resp: F64 = if (data.turn.abs() < dto) {
230                             data.turn
231                         } else {
```

```
                        (F64::from(sign) * dto)
                    };
                    {
                        self.setSpeed(data.clone(), (data.omax * F64::from(sign)));
                        data.turn = (data.turn - resp);
                    }
                }
            }
        }
    }

    data.last = Some(pst);
    return ::Action_mod::Data::from(data);
}

fn EBstep(&mut self, data: ::EmergencyBreak_mod::EBData, ptime: F64) -> ::Action_mod::Data {
    IO::print(strseq!("EMERGENCY BREAK at "));
    IO::println((ptime / F64(1.0E9)));
    for dt in ::Structure_mod::dts.range() {
        self.mctrl.reset(dt.clone());
    }

    return ::Action_mod::Data::from(data);
}

fn PTEstep(&mut self, data: ::PointTurnExecute_mod::PTEData, pst: F64) -> ::Action_mod::Data {
    let mut data: ::PointTurnExecute_mod::PTEData = data.clone();
    if data.last == None {
        IO::println(strseq!("initiate rotate"));
    } else {
        let dt: F64 = ((pst - data.last.expect("Optional was nil.")) /
                       F64::from(::Action_mod::RESOLUTION.clone()));
        {
            let dr: F64 = (::PointTurnExecute_mod::ROT_RATE.clone() * dt);
            {
                let sign: i64 = if (data.angle >= F64::from(0)) {
                    (1 as i64)
                } else {
                    -1
                };
                {
                    let resp: F64 = if (data.angle.abs() < dr) {
                        data.angle
                    } else {
                        (F64::from(sign) * dr)
                    };
                    {
                        for dtnm in ::Structure_mod::dts.domain() {
                            let dev: ::Structure_mod::DriveTrain = {
                                let arg0: ::RobotParameters_mod::dt_id = dtnm.clone();
                                ::Structure_mod::dts.get(arg0.clone())
                            };
                            {
                                let dist: F64 = RobotParameters::getDistance(dtnm.clone());
                                {
                                    let vnew: F64 = (::PointTurnExecute_mod::ROT_RATE.clone() *
                                                     dist);
                                    if (data.angle >= F64::from(0)) {
                                        if dtnm.get(1) == 'l' {
                                            self.mctrl.setWheelVelocity(dev.clone(), -vnew);
                                        } else {
                                            self.mctrl.setWheelVelocity(dev.clone(), vnew);
                                        }
                                    } else {
                                        if dtnm.get(1) == 'r' {
                                            self.mctrl.setWheelVelocity(dev.clone(), -vnew);
                                        } else {
                                            self.mctrl.setWheelVelocity(dev.clone(), vnew);
                                        }
                                    }
                                }
                            }
                        }
                        data.angle = (data.angle - resp);
                    }
                }
            }
        }
    }

    data.last = Some(pst);
    return ::Action_mod::Data::from(data);
```

```
    }

    fn update(&mut self,
              data: ::StraightMovementModify_mod::SMMData,
              pdt: ::Structure_mod::DriveTrain,
              pdv: F64)
              -> () {
        let mut vnew: F64 = self.mctrl.getWheelVelocitySetpoint(pdt.clone());
        vnew = (vnew + pdv);
        if (data.acc >= F64::from(0)) {
            if (vnew >= data.vel) || ((vnew - data.vel).abs() <= ::Action_mod::EPS.clone()) {
                vnew = data.vel;
            }
        } else {
            if (vnew <= data.vel) || ((vnew - data.vel).abs() <= ::Action_mod::EPS.clone()) {
                vnew = data.vel;
            }
        }

        self.mctrl.setWheelVelocity(pdt.clone(), vnew);
        self.mctrl.setSteeringAngle(pdt.clone(), F64(0.0));
    }

    fn SMMstep(&mut self,
               data: ::StraightMovementModify_mod::SMMData,
               pts: F64)
               -> ::Action_mod::Data {
        assert!(self.pre_SMMstep(data.clone(), pts));
        let mut data: ::StraightMovementModify_mod::SMMData = data.clone();
        let mut dv: F64 = if data.last == None {
            F64::from(0)
        } else {
            ((data.acc * (pts - data.last.expect("Optional was nil."))) /
             F64::from(::Action_mod::RESOLUTION.clone()))
        };
        IO::print(strseq!("straight movement towards "));
        IO::print(data.vel);
        IO::print(strseq!(" m/sec (dv = "));
        IO::print(dv);
        IO::println(strseq!(")"));

        for dt in ::Structure_mod::dts.range() {
            self.update(data.clone(), dt.clone(), dv);
        }

        data.last = Some(pts);
        return ::Action_mod::Data::from(data);
    }

    fn SMTstep(&mut self,
               data: ::StraightMovementTimed_mod::SMTData,
               pst: F64)
               -> ::Action_mod::Data {
        IO::print(strseq!("straight movement at "));
        IO::print(data.vel);
        IO::print(strseq!(" m/sec, "));
        for dt in ::Structure_mod::dts.range() {
            self.mctrl.setSteeringAngle(dt.clone(), F64(0.0));
            self.mctrl.setWheelVelocity(dt.clone(), data.vel);

        }

        return ::Action_mod::Data::from(
            ::StraightMovementTimed_mod::SMTData {
                superAction: ::RelativeDelayedAction_mod::RDAData::from(
                    RobotController::RDAstep(data.superAction.clone(), pst))
                , .. data.clone() });
    }

    fn compute(&mut self,
               data: ::TurnMovementPrepare_mod::TMPData,
               pst: F64,
               pmv: F64,
               perr: F64)
               -> F64 {
        assert!(self.pre_compute(data.clone(), pst, pmv, perr));
        if perr == F64::from(0) {
            return pmv;
        }

        if data.last == None {
            return pmv;
```

```
396         }
397
398         {
399             let sign: i64 = if (perr >= F64::from(0)) {
400                 (1 as i64)
401             } else {
402                 -1
403             };
404             {
405                 let dt: F64 = ((pst - data.last.expect("Optional was nil.")) /
406                                F64::from(::Action_mod::RESOLUTION.clone()));
407                 {
408                     let dr: F64 = (::TurnMovementPrepare_mod::ROTATION_RATE.clone() * dt);
409                     {
410                         let res: F64 = if (perr.abs() < dr) {
411                             perr
412                         } else {
413                             (F64::from(sign) * dr)
414                         };
415                         return (pmv + res);
416                     }
417                 }
418             }
419         }
420     }
421
422     fn TMPstep(&mut self,
423                data: ::TurnMovementPrepare_mod::TMPData,
424                pst: F64)
425                -> ::Action_mod::Data {
426         let mut data: ::TurnMovementPrepare_mod::TMPData = data.clone();
427         let mut finish: bool = true;
428         for dtn in ::Structure_mod::dts.domain() {
429             let dt: ::Structure_mod::DriveTrain = {
430                 let arg0: ::RobotParameters_mod::dt_id = dtn.clone();
431                 ::Structure_mod::dts.get(arg0.clone())
432             };
433             {
434                 let dtsp: F64 = {
435                     let arg0: ::RobotParameters_mod::dt_id = dtn.clone();
436                     data.dt_sps.get(arg0.clone())
437                 };
438                 {
439                     let dtmv: F64 = self.mctrl.getActualSteeringAngle(dt.clone());
440                     {
441                         let err: F64 = (dtsp - dtmv);
442                         if (err.abs() > ::TurnMovementPrepare_mod::ONE_DEG_ERR.clone()) {
443                             finish = false;
444                             {
445                                 let arg0: ::Structure_mod::DriveTrain = dt.clone();
446                                 let arg1: F64 = self.compute(data.clone(), pst, dtmv, err);
447                                 self.mctrl.setSteeringAngle(arg0.clone(), arg1);
448                             }
449
450                             self.mctrl.setWheelVelocity(dt.clone(), F64(0.0));
451                         }
452                     }
453                 }
454             }
455         }
456
457         data.done = finish;
458         data.last = Some(pst);
459         return ::Action_mod::Data::from(data);
460     }
461
462     fn isReady(&mut self, data: ::Action_mod::Data) -> bool {
463         return match data.clone() {
464             ::Action_mod::Data::Ch1(::CompositeAction_mod::CAData { plan: _, }) => {
465                 RobotController::CAisReady(::CompositeAction_mod::CAData::from(data.clone()))
466             }
467             ::Action_mod::Data::Ch0(::ActiveWait_mod::AWData { superAction: _, }) => {
468                 RobotController::AWisReady(::ActiveWait_mod::AWData::from(data.clone()))
469             }
470             ::Action_mod::Data::Ch2(::DelayedAction_mod::DAData { stop_at: _,  finished: _, }) => {
471                 RobotController::DAisReady(::DelayedAction_mod::DAData::from(data.clone()))
472             }
473             ::Action_mod::Data::Ch6(::RelativeDelayedAction_mod::RDAData {
474                     delay: _,
475                     superAction: _,
476                 }) => {
477                 RobotController::RDAisReady(
```

```
                    ::RelativeDelayedAction_mod::RDAData::from(data.clone()))
            }
            ::Action_mod::Data::Ch3(::DoubleAckermannExecute_mod::DAEData {
                    last: _,
                    radius: _,
                    omax: _,
                    turn: _,
                }) => {
                self.DAEisReady(::DoubleAckermannExecute_mod::DAEData::from(data.clone()))
            }
            ::Action_mod::Data::Ch4(::EmergencyBreak_mod::EBData { dummy: _, }) => {
                RobotController::EBisReady(::EmergencyBreak_mod::EBData::from(data.clone()))
            }
            ::Action_mod::Data::Ch5(::PointTurnExecute_mod::PTEData { last: _,  angle: _, }) => {
                self.PTEisReady(::PointTurnExecute_mod::PTEData::from(data.clone()))
            }
            ::Action_mod::Data::Ch7(::StraightMovementModify_mod::SMMData {
                    acc: _,
                    vel: _,
                    last: _,
                }) => {
                self.SMMisReady(::StraightMovementModify_mod::SMMData::from(data.clone()))
            }
            ::Action_mod::Data::Ch8(::StraightMovementTimed_mod::SMTData {
                    vel: _,
                    superAction: _,
                }) => {
                RobotController::SMTisReady(
                    ::StraightMovementTimed_mod::SMTData::from(data.clone()))
            }
            ::Action_mod::Data::Ch9(::TurnMovementPrepare_mod::TMPData {
                    last: _,
                    dt_sps: _,
                    done: _,
                }) => {
                RobotController::TMPisReady(::TurnMovementPrepare_mod::TMPData::from(data.clone()))
            }
        };
    }

    fn AWisReady(action: ::ActiveWait_mod::AWData) -> bool {
        return RobotController::RDAisReady(action.superAction.clone());
    }

    fn CAisReady(data: ::CompositeAction_mod::CAData) -> bool {
        return data.plan.clone() == seq!();
    }

    fn DAisReady(action: ::DelayedAction_mod::DAData) -> bool {
        return action.finished;
    }

    fn RDAisReady(action: ::RelativeDelayedAction_mod::RDAData) -> bool {
        return RobotController::DAisReady(action.superAction.clone());
    }

    fn DAEisReady(&mut self, data: ::DoubleAckermannExecute_mod::DAEData) -> bool {
        if (data.turn.abs() < ::Action_mod::EPS.clone()) {
            for dt in ::Structure_mod::dts.range() {
                self.mctrl.setWheelVelocity(dt.clone(), F64(0.0));
            }
            return true;
        } else {
            return false;
        }
    }

    fn EBisReady(_: ::EmergencyBreak_mod::EBData) -> bool {
        return true;
    }

    fn PTEisReady(&mut self, data: ::PointTurnExecute_mod::PTEData) -> bool {
        if (data.angle.abs() < ::Action_mod::EPS.clone()) {
            for dt in ::Structure_mod::dts.range() {
                self.mctrl.setWheelVelocity(dt.clone(), F64(0.0));
            }
            return true;

        } else {
            return false;
        }
    }
```

```
560
561     fn SMMisReady(&mut self, data: ::StraightMovementModify_mod::SMMData) -> bool {
562         return ::Structure_mod::dts.range().forall(|dt: ::Structure_mod::DriveTrain| -> bool {
563             ((data.vel - self.mctrl.getWheelVelocitySetpoint(dt.clone())).abs() <=
564              ::Action_mod::EPS.clone())
565         });
566     }
567
568     fn SMTisReady(action: ::StraightMovementTimed_mod::SMTData) -> bool {
569         return RobotController::RDAisReady(action.superAction.clone());
570     }
571
572     fn TMPisReady(data: ::TurnMovementPrepare_mod::TMPData) -> bool {
573         if data.done {
574             IO::println(strseq!("steering angle is now acquired"));
575         } else {
576             IO::println(strseq!("waiting for steering angle to stabilize"));
577         }
578
579         return data.done;
580     }
581
582     fn pre_DAstep(data: ::DelayedAction_mod::DAData, pst: F64) -> bool {
583         return data.stop_at != None;
584     }
585
586     fn pre_SMMstep(&self, data: ::StraightMovementModify_mod::SMMData, pts: F64) -> bool {
587         return !(data.last != None) || (data.last.expect("Optional was nil.") < pts);
588     }
589
590     fn pre_compute(&self,
591                    data: ::TurnMovementPrepare_mod::TMPData,
592                    pst: F64,
593                    pmv: F64,
594                    perr: F64)
595                    -> bool {
596         return !(data.last != None) || (pst > data.last.expect("Optional was nil."));
597     }
598 }
599
600 impl Default for RobotController {
601     fn default() -> RobotController {
602         let mctrl: MovementController = MovementController::new();
603         let plan: Option<::Action_mod::Data> = Default::default();
604
605         RobotController {
606             mctrl: mctrl,
607             plan: plan,
608         }
609     }
610 }
```

Listing A.4: RobotController class

A.1.5 Actions

A.1.5.1 Action

```
1  /* use declarations omitted */
2
3  // types
4  #[derive(PartialEq, Eq, Clone, Hash)]
5  pub enum Data {
6      Ch0(::ActiveWait_mod::AWData),
7      Ch1(::CompositeAction_mod::CAData),
8      Ch2(::DelayedAction_mod::DAData),
9      Ch3(::DoubleAckermannExecute_mod::DAEData),
10     Ch4(::EmergencyBreak_mod::EBData),
11     Ch5(::PointTurnExecute_mod::PTEData),
12     Ch6(::RelativeDelayedAction_mod::RDAData),
13     Ch7(::StraightMovementModify_mod::SMMData),
14     Ch8(::StraightMovementTimed_mod::SMTData),
15     Ch9(::TurnMovementPrepare_mod::TMPData),
16 }
```

```
impl_union! { Data:
    ::ActiveWait_mod::AWData as Data::Ch0,
    ::CompositeAction_mod::CAData as Data::Ch1,
    ::DelayedAction_mod::DAData as Data::Ch2,
    ::DoubleAckermannExecute_mod::DAEData as Data::Ch3,
    ::EmergencyBreak_mod::EBData as Data::Ch4,
    ::PointTurnExecute_mod::PTEData as Data::Ch5,
    ::RelativeDelayedAction_mod::RDAData as Data::Ch6,
    ::StraightMovementModify_mod::SMMData as Data::Ch7,
    ::StraightMovementTimed_mod::SMTData as Data::Ch8,
    ::TurnMovementPrepare_mod::TMPData as Data::Ch9
}

// values
lazy_static! {
  pub static ref RESOLUTION: u64 = u64::from(F64(1.0E9));
  pub static ref EPS: F64 = F64(0.001);
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct Action;
impl Action {
    // operations
    pub fn new() -> Action {
        let instance: Action = Action::default();
        return instance;
    }
}

impl Default for Action {
    fn default() -> Action {
        Action
    }
}
```

Listing A.5: Action class

A.1.5.2 ActiveWait

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct AWData {
    pub superAction: ::RelativeDelayedAction_mod::RDAData,
}
impl_record! { AWData: superAction as ::RelativeDelayedAction_mod::RDAData }

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct ActiveWait;
impl ActiveWait {
    // operations
    pub fn new() -> ActiveWait {
        let instance: ActiveWait = ActiveWait::default();
        return instance;
    }

    // functions
    pub fn create(pdt: F64) -> AWData {
        AWData::new(RelativeDelayedAction::with_relative_delay(RelativeDelayedAction::create(),
                                                               pdt))
    }
}

impl Default for ActiveWait {
    fn default() -> ActiveWait {
        ActiveWait
    }
}
```

Listing A.6: ActiveWait class

A.1.5.3 CompositeAction

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct CAData {
    pub plan: Seq<::Action_mod::Data>,
}
impl_record! { CAData: plan as Seq<::Action_mod::Data> }

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct CompositeAction;
impl CompositeAction {
    // operations
    pub fn new() -> CompositeAction {
        let instance: CompositeAction = CompositeAction::default();
        return instance;
    }

    // functions
    pub fn create(plan: Seq<::Action_mod::Data>) -> CAData {
        CAData::new(plan.clone())
    }
}

impl Default for CompositeAction {
    fn default() -> CompositeAction {
        CompositeAction
    }
}
```

Listing A.7: CompositeAction class

A.1.5.4 DelayedAction

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct DAData {
    pub stop_at: Option<F64>,
    pub finished: bool,
}
impl_record! { DAData:
    stop_at as Option<F64>,
    finished as bool
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct DelayedAction;
impl DelayedAction {
    // operations
    pub fn new() -> DelayedAction {
        let instance: DelayedAction = DelayedAction::default();
        return instance;

    }


    // functions
    pub fn create() -> DAData {
        DAData::new(None, false)
    }
}



impl Default for DelayedAction {
    fn default() -> DelayedAction {

        DelayedAction
    }
```

```
}
```

Listing A.8: DelayedAction class

A.1.5.5 DoubleAckermannExecute

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct DAEData {
    pub last: Option<F64>,
    pub radius: ::RobotParameters_mod::dt_data,
    pub omax: F64,
    pub turn: F64,
}
impl_record! { DAEData:
    last as Option<F64>,
    radius as ::RobotParameters_mod::dt_data,
    omax as F64,
    turn as F64
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct DoubleAckermannExecute;
impl DoubleAckermannExecute {
    // operations
    pub fn new() -> DoubleAckermannExecute {
        let instance: DoubleAckermannExecute = DoubleAckermannExecute::default();
        return instance;
    }

    // functions
    pub fn create(ptr: ::RobotParameters_mod::dt_data, pomax: F64, pturn: F64) -> DAEData {
        DAEData::new(None, ptr.clone(), pomax, pturn)
    }
}

impl Default for DoubleAckermannExecute {
    fn default() -> DoubleAckermannExecute {
        DoubleAckermannExecute
    }
}
```

Listing A.9: DoubleAckermannExecute class

A.1.5.6 DoubleAckermannTurn

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct DoubleAckermannTurn;
impl DoubleAckermannTurn {
    // operations
    pub fn maxTurnRadius(ptr: ::RobotParameters_mod::dt_data) -> F64 {
        let mut res: F64 = F64::from(0);
        for r in ptr.range() {
            if (r > res) {
                res = r;
            }
        }

        return res;
    }

    pub fn create(px: F64, pturn: F64) -> ::CompositeAction_mod::CAData {
        let mut pts: ::RobotParameters_mod::dt_data =
```

```
            ::RobotParameters_mod::positions.domain()
                .map_compr(|x: Seq<char>| -> bool { true } ,
                        |x: Seq<char>| -> (Seq<char>, F64) {
                            (x.clone(), if x.get(2) == 'm' {
                                F64::from(0)
                            } else {
                                DoubleAckermannTurn::computeSteeringAngle(px, x.clone())
                            })});
        let mut ptr: ::RobotParameters_mod::dt_data = ::RobotParameters_mod::positions.domain()
            .map_compr(|x: Seq<char>| -> bool { true } ,
                        |x: Seq<char>| -> (Seq<char>, F64) {
                            (x.clone(), DoubleAckermannTurn::computeSteeringRadius(px, x.clone()))
                        });
        let mut omax: F64 = (F64(0.25) / DoubleAckermannTurn::maxTurnRadius(ptr.clone()));

        {
            let  actions: Seq<::Action_mod::Data> = seq!(
                ::Action_mod::Data::from(TurnMovementPrepare::create(pts.clone())),
                ::Action_mod::Data::from(DoubleAckermannExecute::create(ptr.clone(), omax, pturn)),
                ::Action_mod::Data::from(TurnMovementPrepare::create(
                    ::RobotParameters_mod::steering_init.clone()))
            );
            return CompositeAction::create(actions.clone());
        }
    }

    pub fn new() -> DoubleAckermannTurn {
        let instance: DoubleAckermannTurn = DoubleAckermannTurn::default();
        return instance;
    }

    // functions
    pub fn computeSteeringAngle(px: F64, pid: ::RobotParameters_mod::dt_id) -> F64 {
        {
            let (dtx, dty): (F64, F64) = RobotParameters::getXY(pid.clone());
            {
                let pdx: F64 = (px - dtx);
                if (pdx.abs() < ::Action_mod::EPS.clone()) {
                    F64::from(0)
                } else {
                    MATH::atan((dty / pdx))
                }
            }
        }
    }

    pub fn computeSteeringRadius(px: F64, pid: ::RobotParameters_mod::dt_id) -> F64 {
        {
            let (dtx, dty): (F64, F64) = RobotParameters::getXY(pid.clone());
            {
                let dx: F64 = (px - dtx);
                MATH::sqrt(((dx * dx) + (dty * dty)))
            }
        }
    }
}

impl Default for DoubleAckermannTurn {
    fn default() -> DoubleAckermannTurn {
        DoubleAckermannTurn
    }
}
```

Listing A.10: DoubleAckermannTurn class

A.1.5.7 EmergencyBreak

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct EBData {
    pub dummy: quotes::EMGBreak,
}
impl_record! { EBData: dummy as quotes::EMGBreak }
```

```

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct EmergencyBreak;
impl EmergencyBreak {
    // operations
    pub fn new() -> EmergencyBreak {
        let instance: EmergencyBreak = EmergencyBreak::default();
        return instance;
    }

    // functions
    pub fn create() -> EBData {
        EBData::new(quotes::EMGBreak)
    }
}

impl Default for EmergencyBreak {
    fn default() -> EmergencyBreak {
        EmergencyBreak
    }
}
```

Listing A.11: EmergencyBreak class

A.1.5.8 PointTurn

```
/* use declarations omitted */

// values
lazy_static! {
   static ref pt_angles: ::RobotParameters_mod::dt_data = map!(
     strseq!("lf") => (MATH::pi - PointTurn::getAngle(strseq!("lf"))),
     strseq!("rf") => -PointTurn::getAngle(strseq!("rf")),
     strseq!("lm") => F64(0.0),
     strseq!("rm") => F64(0.0),
     strseq!("lr") => -(MATH::pi - PointTurn::getAngle(strseq!("lr")).abs()),
     strseq!("rr") => -PointTurn::getAngle(strseq!("rr"))
   );
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct PointTurn;
impl PointTurn {
    // operations
    pub fn new() -> PointTurn {
        let instance: PointTurn = PointTurn::default();
        return instance;
    }

    // functions
    fn getDistance(pid: ::RobotParameters_mod::dt_id) -> F64 {
        {
            let (res, _): (F64, F64) = {
                let arg0: ::RobotParameters_mod::dt_id = pid.clone();
                ::RobotParameters_mod::positions.get(arg0.clone())
            };
            res
        }
    }

    fn getAngle(pid: ::RobotParameters_mod::dt_id) -> F64 {
        {
            let (_, res): (F64, F64) = {
                let arg0: ::RobotParameters_mod::dt_id = pid.clone();
                ::RobotParameters_mod::positions.get(arg0.clone())
            };
            res
        }
    }

    pub fn create(pangle: F64) -> ::CompositeAction_mod::CAData {
        {
            let  actions: Seq<::Action_mod::Data> = seq!(
        ::Action_mod::Data::from(TurnMovementPrepare::create(::PointTurn_mod::pt_angles.clone())),
        ::Action_mod::Data::from(PointTurnExecute::create(pangle)),
```

```
        ::Action_mod::Data::from(TurnMovementPrepare::create(
          ::RobotParameters_mod::steering_init.clone()))
      );
            CompositeAction::create(actions.clone())
        }
    }
}

impl Default for PointTurn {
    fn default() -> PointTurn {
        PointTurn
    }
}
```

Listing A.12: PointTurn class

A.1.5.9 PointTurnExecute

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct PTEData {
    pub last: Option<F64>,
    pub angle: F64,
}
impl_record! { PTEData:
    last as Option<F64>,
    angle as F64
}

// values
lazy_static! {
  pub static ref ROT_RATE: F64 = (MATH::pi / F64::from(4));
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct PointTurnExecute;
impl PointTurnExecute {
    // operations
    pub fn new() -> PointTurnExecute {
        let instance: PointTurnExecute = PointTurnExecute::default();
        return instance;
    }

    // functions
    pub fn create(pangle: F64) -> PTEData {
        PTEData::new(None, pangle)
    }
}

impl Default for PointTurnExecute {
    fn default() -> PointTurnExecute {
        PointTurnExecute
    }
}
```

Listing A.13: PointTurnExecute class

A.1.5.10 RelativeDelayedAction

```
/* use declarations omitted */

// types
pub type TimeDelay = F64;
pub fn inv_TimeDelay(td: F64) -> bool {
    (td >= F64::from(0))
}
```

```

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct RDAData {
    pub delay: TimeDelay,
    pub superAction: ::DelayedAction_mod::DAData,
}
impl_record! { RDAData: delay as TimeDelay,  superAction as ::DelayedAction_mod::DAData }

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct RelativeDelayedAction;
impl RelativeDelayedAction {
    // operations
    pub fn new() -> RelativeDelayedAction {
        let instance: RelativeDelayedAction = RelativeDelayedAction::default();
        return instance;
    }

    // functions
    pub fn create() -> RDAData {
        RDAData::new(TimeDelay::from(0), DelayedAction::create())
    }

    pub fn with_relative_delay(d: RDAData, pd: F64) -> RDAData {
        RDAData { delay: (pd * F64::from(::Action_mod::RESOLUTION.clone())), ..d.clone() }
    }
}

impl Default for RelativeDelayedAction {
    fn default() -> RelativeDelayedAction {
        RelativeDelayedAction
    }
}
```

Listing A.14: RelativeDelayedAction class

A.1.5.11 StraightMovement

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct StraightMovement;
impl StraightMovement {
    // operations
    pub fn create(pvmax: F64, pamax: F64, pdist: F64) -> ::CompositeAction_mod::CAData {
        assert!(StraightMovement::pre_create(pvmax, pamax, pdist));
        let mut plan: Seq<::Action_mod::Data> = seq!();
        {
            let tmax: F64 = (pvmax.abs() / pamax);
            {
                let xtmax: F64 = ((pamax * tmax) * tmax);
                {
                    let acc: F64 = if (pvmax > F64::from(0)) {
                        pamax
                    } else {
                        -pamax
                    };
                    if (xtmax < pdist) {
                        plan = plan.conc(seq!(
                            ::Action_mod::Data::from(StraightMovementModify::create(acc, pvmax))));
                        {
                            let tlin: F64 = ((pdist - xtmax) / pvmax.abs());
                            plan = plan.conc(seq!(
                                ::Action_mod::Data::from(StraightMovementTimed::create(tlin, pvmax))));
                        }

                        plan = plan.conc(seq!(
                            ::Action_mod::Data::from(StraightMovementModify::create(-acc,
                                                                                    F64::from(0)))));

                    } else {
                        {
                            let vnom: F64 = (acc * MATH::sqrt((pdist / pamax)));
                            {
                                plan = plan.conc(seq!(
```

```
                                    ::Action_mod::Data::from(
                                        StraightMovementModify::create(acc, vnom)))
                                );
                                plan = plan.conc(seq!(
                                    ::Action_mod::Data::from(
                                        StraightMovementModify::create(-acc, F64::from(0))))
                                );
                            }
                        }
                    }
                }
            }

        return CompositeAction::create(plan.clone());
    }

    pub fn new() -> StraightMovement {
        let instance: StraightMovement = StraightMovement::default();
        return instance;
    }

    fn pre_create(pvmax: F64, pamax: F64, pdist: F64) -> bool {
        return pvmax != F64::from(0) && (pamax > F64::from(0)) && (pdist > F64::from(0));
    }
}

impl Default for StraightMovement {
    fn default() -> StraightMovement {
        StraightMovement
    }
}
```

Listing A.15: StraightMovement class

A.1.5.12 StraightMovementModify

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct SMMData {
    pub acc: F64,
    pub vel: F64,
    pub last: Option<F64>,
}
impl_record! { SMMData:
    acc as F64,
    vel as F64,
    last as Option<F64>
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct StraightMovementModify;
impl StraightMovementModify {
    // operations
    pub fn new() -> StraightMovementModify {
        let instance: StraightMovementModify = StraightMovementModify::default();
        return instance;
    }

    // functions
    pub fn create(acc: F64, vel: F64) -> SMMData {
        SMMData::new(acc, vel, None)
    }
}

impl Default for StraightMovementModify {
    fn default() -> StraightMovementModify {
        StraightMovementModify
    }
}
```

Listing A.16: StraightMovementModify class

A.1.5.13 StraightMovementTimed

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct SMTData {
    pub vel: F64,
    pub superAction: ::RelativeDelayedAction_mod::RDAData,
}
impl_record! { SMTData:
    vel as F64,
    superAction as ::RelativeDelayedAction_mod::RDAData
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct StraightMovementTimed;
impl StraightMovementTimed {
    // operations
    pub fn new() -> StraightMovementTimed {
        let instance: StraightMovementTimed = StraightMovementTimed::default();
        return instance;
    }

    // functions
    pub fn create(dt: F64, vel: F64) -> SMTData {
        SMTData::new(vel,
                     RelativeDelayedAction::with_relative_delay(RelativeDelayedAction::create(),
                                                                dt))
    }
}

impl Default for StraightMovementTimed {
    fn default() -> StraightMovementTimed {
        StraightMovementTimed
    }
}
```

Listing A.17: StraightMovementTimed class

A.1.5.14 TurnMovementPrepare

```
/* use declarations omitted */

// types
#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct TMPData {
    pub last: Option<F64>,
    pub dt_sps: ::RobotParameters_mod::dt_data,
    pub done: bool,
}
impl_record! { TMPData:
    last as Option<F64>,
    dt_sps as ::RobotParameters_mod::dt_data,
    done as bool
}

// values
lazy_static! {
  pub static ref ONE_DEG_ERR: F64 = (MATH::pi / F64::from(180));
  pub static ref ROTATION_RATE: F64 = (MATH::pi / F64::from(4));
}

#[derive(PartialEq, Eq, Clone, Hash, Debug)]
pub struct TurnMovementPrepare;
impl TurnMovementPrepare {
    // operations
    pub fn new() -> TurnMovementPrepare {
        let instance: TurnMovementPrepare = TurnMovementPrepare::default();
        return instance;
    }

    // functions
```

```
32      pub fn create(dtd: ::RobotParameters_mod::dt_data) -> TMPData {
33          TMPData::new(None, dtd.clone(), false)
34      }
35  }
36
37  impl Default for TurnMovementPrepare {
38      fn default() -> TurnMovementPrepare {
39          TurnMovementPrepare
40      }
41  }
```

Listing A.18: TurnMovementPrepare class

A.1.6 main

```
1   #![allow(non_snake_case, non_upper_case_globals,
2    unused_variables, dead_code, unused_mut,unused_parens,
3   non_camel_case_types,unused_imports)]
4
5   #[macro_use]
6   extern crate lazy_static;
7
8   #[macro_use]
9   extern crate codegen_runtime;
10
11  mod RobotParameters_mod;
12  mod RobotController_mod;
13  mod Structure_mod;
14  mod MovementController_mod;
15  mod ActiveWait_mod;
16  mod CompositeAction_mod;
17  mod DelayedAction_mod;
18  mod DoubleAckermannTurn_mod;
19  mod DoubleAckermannExecute_mod;
20  mod EmergencyBreak_mod;
21  mod PointTurn_mod;
22  mod PointTurnExecute_mod;
23  mod RelativeDelayedAction_mod;
24  mod StraightMovementModify_mod;
25  mod StraightMovementTimed_mod;
26  mod StraightMovement_mod;
27  mod TurnMovementPrepare_mod;
28  mod Action_mod;
29  mod quotes;
30
31  use codegen_runtime::*;
32  use RobotController_mod::RobotController;
33
34  fn main() {
35      // Entry point is not auto generated
36      let mut ctrl = RobotController::new();
37      ctrl.run(16);
38  }
```

Listing A.19: Entry point

A.1.7 Quotes

```
1   impl_quote! { EMGBreak }
2   impl_quote! { append }
3   impl_quote! { start }
```

Listing A.20: Quote definitions

A.1.8 Cargo manifest

```
[package]
name = "Rover"
version = "0.1.0"
authors = ["Overture User <info@overturetool.org>"]

[dependencies]
codegen_runtime = { git = "https://github.com/LasseBP/codegen_runtime.git" }
lazy_static = "0.1.*"
```

Listing A.21: Cargo manifest