Creating a Tool for the Translation of VDM-SL to Isabelle HOL

BSc Computer Science

Newcastle University

Project Supervisor Dr Leonardo J.S Freitas

Jamie Simm

2019

Abstract

This dissertation aims to explain and describe the implementation of and need for a VDM-SL (Vienna Development Method - Specification Language) to Isabelle HOL (Higher Order Logic) translation tool. The tool is applied to a VDM model of the POLAR organ persufflation machine's alarm system. As Isabelle theories can be proven mathematically by the modeller, it is hoped that this tool will be useful in the verification of software by eliminating the need for manual VDM to Isabelle translation.

Declaration

I declare that this dissertation represents my own work except where otherwise stated. The length of this document is only so large for its code snippets and figures, the majority of which take up a considerable amount of the page. To avoid having large code blocks in the main text, relevant code is stored and organised in the Appendices, when the text references code, a footnote will guide the reader to the relevant appendix and section. The exception to this rule is in the Results chapter in which translations are presented. The reader is urged to follow such footnotes in order to make sense of references to computation made in the text. This document contains 14754 words and spans 30 pages without figures or code.

Acknowledgements

I would like to thank my supervisor, Dr Leo Freitas, for guiding me toward the project of developing this tool, which I have thoroughly enjoyed; for introducing me to the world of formal specification, Isabelle and VDM; and for fostering my interest in automated translation and proof tools. Secondly, I would like to thank Casper Thule Hansen of Aarhus University, Denmark, for setting up the environment upon which this tool was built and teaching me how to write transformation visitors. I believe that this project may not have been so successful without the essential and flexible platform upon which it is built.

List of Figures

Contents

Chapter 1

Introduction

1.1 The POLAR machine and persufflation

Organ preservation is a vital factor in the success and availability of allotransplantation[1] as a treatment for illness, conventional methods like Static Cold Storage (SCS) have a number of issues - advancements in this field could serve to transform medicine[Giw+17]. One such issue is that SCS is incapable of sufficient oxygen delivery during preservation[Pap+05]. The POLAR project aims to address this issue by designing a machine to automate another method of organ preservation, persufflation, which gives a far higher oxygen supply per gram of tissue.

1.2 Verifying the correctness of the machine to ensure reliability

To achieve verification of the system we apply formal modelling techniques to its specification in order to derive an unambiguous representation of its components and its operation. For POLAR, this has been done by translating the machine's various alarm conditions into a VDM-SL model. VDM-SL is a formal modelling language, it allows the modeller to abstract the system into a mathematical representation, its syntax rules and semantics are so precisely defined that there is no room for disagreement about the model's properties. This means that the POLAR VDM-SL model can be analysed using mathematical proof. In principal, we can prove that a program, in this case the alarm system, is correct with respect to its specification and prove that the POLAR VDM-SL model embodies a property such as safety - which is critical in a medical system. While VDM-SL has tools for generating proof obligations, it has no support for mathematical proof of those obligations.

One way to prove a VDM-SL model is to translate it into HOL[2] so that it can then be proven using the proof assistant Isabelle, this is how the POLAR model will be proven. Ordinarily, translation is done manually, this can be an arduous process for the modeller/prover who should only be concerned with the proof of the model rather than the process of translation.

Currently, there exists an effort to create a tool that will automate translation so that the modeller/prover can focus their efforts on proof of the system, as this is the pertinent part of the verification process[Gitnt]. In its current state at the time of beginning this dissertation, the tool does not have the functionality to translate the POLAR model and

[1]The transplantation of organs, tissue, or cells from a genetically distinct individual of the same species.

[2]Higher order logic, essentially higher-order simple predicate logic, quantifies over an arbitrary number of nested sets i.e. where first-order logic quantifies only variables that range over individuals and second-order logic quantifies variables that range over both individuals and sets; third-order also quantifies over sets of sets etc. Higher Order Logic is a union of first, second third up to any number of N nested sets.

can only translate integer basic types as well as arithmetic and predicate expressions. With more work, applying the automated translation tool to the POLAR model could save a considerable amount of time in its verification and work on proving the machine's alarm system mathematically could begin sooner and therefore be more thorough. Human errors made during manual translation would be removed, reducing the likelihood that the modeller wastes time proving an incorrect translation.

1.3 Aims and Objectives

1.3.1 Aim

The aim of this dissertation is to develop the current idea on the VDM-SL to Isabelle translation tool so that it can translate more components of the POLAR model.

1.3.2 Objectives

- To learn concepts of advanced software design patterns[Gam+95], namely AST's and visitors.
- To be able to use these concepts in order to learn the Java visitor design pattern.
- Gain a complete understanding of the visitor pattern's implementation with adaptor classes and cases in Overture/VDM.
- Use this knowledge to create Java visitor classes that are able to make the AST transformations.
- Through writing more transformations, create functionality for translation to Isabelle based on recipes understood from both previously studied university modules[3] and a directory of examples.[4]
- Apply this tool to basic real-world whole model examples to test functionality.
- Apply this tool to the VDM-SL POLAR organ persufflation machine model, to output a successful Isabelle HOL translation.

1.4 Document structure

The following document will: provide background by explaining and discussing the area/-domain that the project affects and explores; clarify existing works on the translation tool; present the implementation of my contribution to the tool, with a description of implementation issues and headlines; present the results of the tools development, as well as the results of its application to the POLAR example, and reflect upon the development process; conclude this dissertation with a proposal of future work, description of the impact of this project, a measure of success against proposed aims and objectives and an analysis of the project's complexity.

[3]Software Verification Technologies, Dr. Leonardo J.S Freitas.

[4]Written by Leo J.S Freitas, stored in a private GitLab

Chapter 2

Background

This section attempts to explain and discuss the key concepts that are the foundation of this project: the POLAR organ persufflation machine and its formal model in VDM; mathematically proving a model in Isabelle and implementing a tool to translate a VDM model into Isabelle HOL; an outline of the problems that this dissertation aims to tackle, as well as the proposed solutions to those problems.

2.1 Persufflation

Organ persufflation is a method for organ preservation in which a donor organ is submerged in a cold preservation solution, a catheter[1] is then inserted into the vasculature[2] of the organ and oxygen is pumped into the organ through the catheter, thereby oxygenating[3] the organ, see Fig2.1. Through this method, more oxygen is delivered to the organ than the other most popular methods and by doing so preservation period can be extended by up to 48-72 hours[Sus+13].

However, Persufflation is a delicate process. Oxygen flow, temperature and pressure both inside and outside of the organ must be kept within a precise margin; slightly anomalous variations in any of these variables, could result in damage to the organ. Until now, humans have controlled these variables manually: typically, an alarm will sound if a variable exceeds its boundary and an attendant will adjust the value by hand. This method is not precise enough under such safety critical circumstances as human error may result in damage to the organ pre-transplant or failure of the organ post-transplant, due to poor condition.

Through software, the POLAR project's persufflation machine would enable effective, long term persufflation by removing the human element. By exploiting the speed, precision and accuracy of a computer system, POLAR hopes to be able to control these variables with high precision and within a minuscule margin of error. However, as is the nature of software of such complexity, the alarm system that informs changes to variables may be rife with errors, with high potential for unintended behaviour[CMK17]. In a safety critical system such as this, it is imperative that software components do not malfunction.

2.2 POLAR System Complexity

The POLAR machine is a Cyber-Physical System - collaborating computational elements controlling physical entities, which interact with humans and their environment. For the sake of keeping safety in mind, it is important to note that the software that will be

[1]A flexible tube inserted through a narrow opening into a body cavity.
[2]The arrangement of blood vessels in the body, or within an organ.
[3]Supply, treat, charge, or enrich with oxygen.

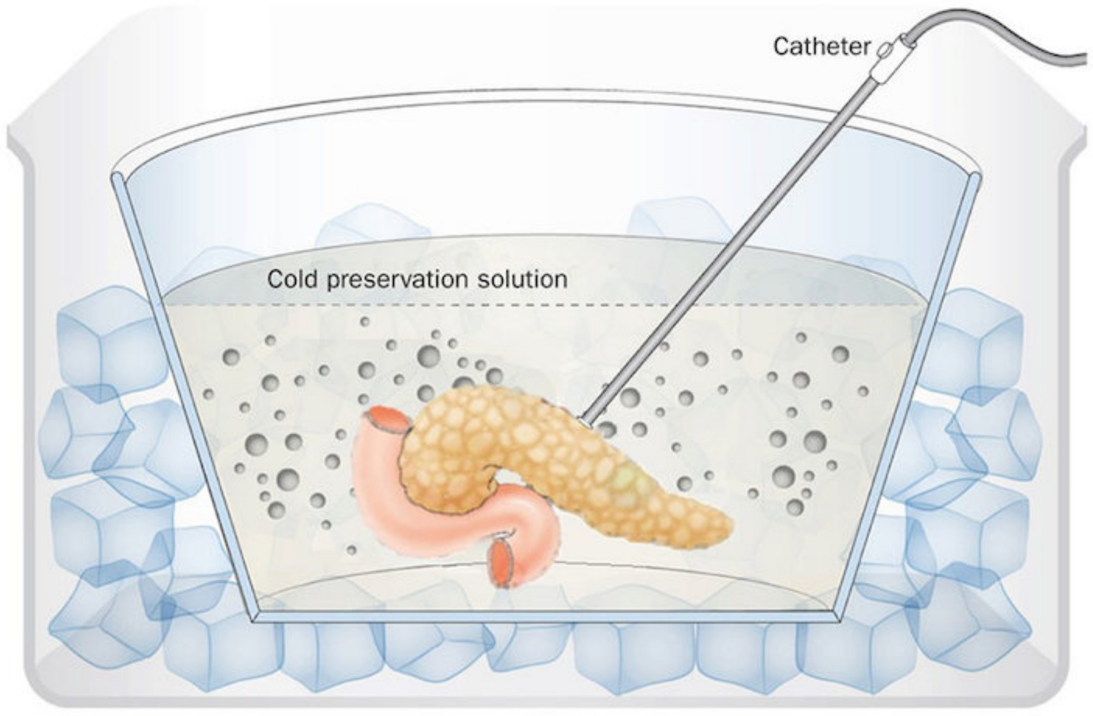

Figure 2.1: Persufflation [Dho+18]

developed for POLAR is part of a system - a combination of interacting elements organized to achieve one or more stated purposes[Wal+07]. As such, the system will have several dimensions to take into account - building complexity. Physical variables such as external and internal pressure, temperature, atmospheric composition, movement and more. Also computational variables such as security protocols, fail safes, varying software; Human variables, how will the machine be handled, used, what errors may humans introduce into the system?

Specifically, human contribution is arguably the central source of error and complexity in a system. Separate discipline-specific development processes result in components being developed in a disjointed manner. In the case of POLAR, software is developed separate to the hardware, a machine engineer is not able to know precisely how their chip's resources will be handled by kernel processes written by the operating system developer, and so they are unable to understand which features might cause error, which limitations the software will have. All of this risks late discovery of defects, namely at the integration stage, when it is highly cumbersome to go back.

2.3 Modelling

The best way to bridge the gap between varying components, development teams, dimensions of the system, is to create a model of the system, an abstraction of all of these variables together in one unambiguous representation so that the variables of the system can be tested and scrutinised as a whole, their interaction monitored and alterations made where error is found. Models allow us to explore a design space before we build, and physically integrate, the components within it. Interactions are modelled as contracts, assumptions and guarantees of what will or should happen rather than how. As an architect models a structure to smaller scale before it is built - so is a software and hardware system modelled before it is integrated and implemented. This provides evidence for trust, reduces risk of bugs and can even identify improvements to the current design. In POLAR, a system that will someday contribute to human well-being, it is easy to see how an optimal design and faster development will be beneficial. For an increasing amount of software, this model takes the form of a formal specification.

2.4 Formal Specification

Formal specifications are used to describe a system, analyse its behaviour and to aid in its design by verifying key properties of interest through rigorous and effective reasoning tools[Hie+][Gau94]. These specifications are formal in the sense that they have a syntax, their semantics fall within one domain, and they are able to be used to infer useful information[Lam00].[19] Ordinarily, a formal specification of a system goes no further than a UML diagram of its components and mostly all projects have some form of it. This is a good first approximation, but it is imprecise. For safety critical systems, a mathematical, unambiguous abstraction of the system, which states the effect of computation, is the only acceptable level of specification. A mathematical representation can be analysed, altered and proven using deduction and reasoning techniques - and therefore, so can the design of the system with respect to its specification. A mathematical specification is encoded in an appropriate formal modelling programming language, for POLAR this is VDM-SL.

2.5 Vienna Development Method Specification Language (VDM-SL)

Though readers of this dissertation might already be familiar with VDM, a brief overview of the language is necessary to provide motivation for this project. VDM is a state-based modelling language which allows the modeller to formally specify structure, behaviour and logical constraints. The Vienna Development Method (VDM) was established in the 1970's originating from IBM Laboratory, Vienna. The language has syntax to represent mathematical representation of a specification. Mathematical constructs including, sets, sequences and integers, to name a small few, define types of data that are maintained and transformed in the system. How this data is represented, what restrictions are placed on the data, represented as invariants on types and values, and what data forms the persistent state of the system, define its data set. Behaviour of the system is encoded as functions that represent functionality, and as operations which modify its persistent state. Pre and post conditions for operations and functions are purposeful in both restricting and checking their operation. Errors in the model identify errors in the specification of the system, the specification can then be altered, and the design improved to eliminate errors before the new specification is written in VDM-SL again so that all of this can be repeated. Many iterations of this - model, check, alter, model again - cycle incrementally identify and eliminate errors until none, or as few as possible within a reasonable threshold, can be found.

2.5.1 VDM-SL Example, Alarm System

Below, VDM code represents the alarm system for a nuclear power plant reactor. Experts are paged when certain variables' values cross safe boundaries. Certain experts are on shift at certain times and paging is decided based on period of time over boundaries as well as other important factors such as an expert's qualification.

```
types
```

```
Schedule = map Period to set of Expert;

Period = token;

Expert :: expertid : ExpertId
quali : set of Qualification
inv ex == ex.quali <> {};

ExpertId = token;
Qualification = <Elec> | <Mech> | <Bio> | <Chem>;
Alarm :: alarmtext : seq of char
quali : Qualification;
```

Listing 2.1: Types of data used in the alarm system in VDM-SL

Above, an invariant on the Expert type is defined to ensure that the expert does not have an empty set of qualifications i.e. the expert is qualified, and their qualifications are recorded.

```
values

  p1:Period = mk_token("Monday day");
  ps : set of Period = {p1,p2,p3,p4,p5};

  eid8:ExpertId = mk_token(190);

  e1:Expert = mk_Expert(eid1,{<Elec>});
  exs : set of Expert = {e1,e2,e3,e4,e5,e6,e7,e8};
```

Listing 2.2: Alarm system's values in VDM-SL

Some of the values in the system are shown above. A set of experts is created, one such expert having a particular id and electrical qualification.

```
functions
  QualificationOK: set of Expert * Qualification -> bool
  QualificationOK(exs2,reqquali) ==
    exists ex in set exs2 & reqquali in set ex.quali;
```

Listing 2.3: An alarm system function in VDM-SL which takes a set of experts a qualification and returns a boolean value.

```
state Plant of
schedule : Schedule
alarms : set of Alarm
```

Listing 2.4: Alarm system's persistent state in VDM-SL t he state is represented as a record type with the important persistent data as fields here schedule and alarms.

```
operations

NumberOfExperts: Period ==> nat
NumberOfExperts(peri) == is not yet specified
```

```
pre peri in set dom schedule;

ExpertIsOnDuty: Expert ==> set of Period

ExpertIsOnDuty(ex) == is not yet specified;

ExpertToPage: Alarm * Period ==> Expert

ExpertToPage(a,peri) == is not yet specified;
```

Listing 2.5: Operations on the state of the alarm system in VDM-SL operations manipulate data stored in the persistent state and mimic the operation of the system.

2.6 Isabelle Translation

VDM provides an unambiguous mathematical representation of the system, **however it does not provide any support for mathematical proof of the specification**. Mathematically proving or disproving correctness of the VDM model and its intended algorithms, allows us to say without doubt that they are correct with respect to their formal specification. Isabelle allows us to write code contracts in Higher Order Logic statements and provides an automated theorem proving assistant to prove them. Code contracts are assumptions and guarantees in the model, features written as VDM constructs. Contracts include but are not limited to:

- Properties that always hold (i.e. invariants).
- Assumptions (pre conditions) and commitments (post conditions).
- Proof obligations of interest such as satisfiability[4] and reification[5], which if proved would establish the consistency of the model.
- Proof obligations (POs) of what correctness means, verifying that we have built the correct model.
- Sanity checks on functionality, validating that the model has been built correctly.

Isabelle is a functional programming language, constructs are represented as fields and curried functions, all collected together in a **theory**, **.thy**, file. A theory is a named collection of types, functions, and theorems, much like a module in a programming language or a specification in a specification language like VDM. Isabelle contains a theory file **"Main"**, which is a union of all the basic, predefined theories like arithmetic, lists, sets, etc. which are used by the automated theorem prover to inform proof assistants. Below, a module "VDMToolkit"[6] is included in the imports. This is very important in the VDM to Isabelle translation steps as, similar to **"Main"**, it provides type-checked VDM constructs with pre, post conditions and invariants for VDM types like VDMNat1 and VDMSet, as VDM represents these things differently to Isabelle.

To use Isabelle to prove a model, that model must first be translated from VDM into Isabelle HOL. For the above example, translations are below.

```
theory Alarm
```

[4]A logical check to verify that operations are feasible

[5]Do representations of the data in the system agree with one another, are they compatible?

[6]VDMToolkit is written and maintained by Dr Leonardo Jose Simoes Freitas. Newcastle University.

```
|    2| imports "../../lib/VDMToolkit"
|    3| begin
|   42|
|   42|
|    7| type_synonym Period   = VDMToken
|   14| definition
|   15|   inv_Period :: "Period    "
|   16|   where
|   17|   "inv_Period   inv_True"
|   18|
|   19| type_synonym ExpertId = VDMToken
|   20|
|   21| datatype Qualification = Elec | Mech | Bio | Chem
|   22|
|   23| definition
|   24|   inv_Qualification :: "Qualification    "
|   25|   where
|   26|   "inv_Qualification   inv_True"
|   27|
|   28| record Alarm =
|   29|   alarm_alarmtext :: "char VDMSeq"
|   30|   alarm_quali     :: Qualification
|   27|
|   38| definition
|   39|   inv_Alarm :: "Alarm    "
|   40|   where
|   41|   "inv_Alarm   inv_True"
|   42|
|   43| record Expert =
|   44|   expert_expertid  :: ExpertId
|   45|   expert_quali     :: "Qualification VDMSet"
|   46|
|   47| definition
|   48|   inv_Expert :: "Expert    "
|   49|   where
|   50|   "inv_Expert e
|   51|       let eq = (expert_quali e) in
|   52|         inv_SetElems inv_True eq
|   53|         eq   {}"
|   54|
|   74| type_synonym Schedule = "Period   Expert set"
|   75|
|   77| definition
|   78|   inv_Schedule :: "Schedule    "
|   79|   where
|   80|   "inv_Schedule s
|   81|       inv_Map inv_Period (inv_SetElems inv_Expert) s
|   82|
|   83|       (  exs1   rng s .
|   84|           exs1   {}
|   85|           (  ex1   exs1 .   ex2   exs1 .
             ex1   ex2   (expert_quali ex1)   (expert_quali ex2)))"

end
```

The translation 'recipe' detailing the general method for translation will be detailed in the Implementation section of this document later, see 3.5. It is evident that translations might become very intricate and cumbersome for more complex models, human error during repetitive tasks is common, and manual translation takes time. The complexity comes from the level of detail required for each VDM construct's translation: each type must have an invariant checking all of its subsequent types; when it comes to function translations, which we will see later, each function needs a pre and a post condition also; state must be initialised in a separate function which must also have a pre and a post condition and so on. The tool, whose development is detailed in the next chapter, automates translation to leave the modeller free to concentrate on proving the translated model rather than translating it manually - and consequently spending time fixing minor errors and filling in gaps in that translation. [7]

[7]See the Isabelle manual for more information: https://isabelle.in.tum.de/dist/Isabelle2018/doc/tutorial.pdf

Chapter 3

Implementation

This chapter will: describe the existing architecture, namely the technologies involved with the tool such as the VDM AST[1] and the intermediate representation of VDM; describe how the existing tool works; describe the five step methodology that was used and repeated in a development cycle; describe the ways in which I have extended the existing tool and detail the implementation of these extensions and finally outline the translation 'recipe'.

3.1 Existing Architecture Description

3.1.1 The VDM AST and Core Modules

The Abstract Syntax Tree (AST) "is an in-memory representation of the VDM model being worked on"[Cou19a] and is made up of a series of classes which implement an AST for VDM. The AST is generally structured like in Fig3.1. .

Nodes have fields which hold information on their values, as well as fields which hold information about, and provide access to, their children and ancestors. Neighbouring nodes - children and ancestor nodes - can be used to infer the structure of the AST surrounding any given node. Child nodes are fields of their parent nodes, this provides the tree structure of the AST. The highest-level class entity in the AST is the abstract `Node` class which implements, and provides default definitions for, methods in the `INode` interface. This interface defines common behaviour of all nodes in the AST whether they are binary expressions or function patterns. To name but a few, the INode interface contains abstract methods like `parent()`, `getChildren()` and `removeChild()` which all allow for the AST to be manipulated accordingly by visitor classes later. The INode interface also includes abstract `apply()` methods to support the visitor pattern for various adaptors, these concepts are explained in detail in section 3.2.3. The `PGen` interface is implemented by `PGenBase` which extends `Node` and is extended by `SSubGenBase` which implements the `SSubGen` interface. Evidently then, the structure of the AST is intricate but does provide excellent extensibility for tools such as the one that this project will attempt to implement.

A more concrete example of the AST's structure is the `PExp` expressions family of VDM nodes seen in Fig3.2. This example shows the AST hierarchy for a simple numeric binary expression like $a + b$. The concrete classes `APlusNumericBinaryExp` and `ATimesNumericBinaryExpIR` are leaf nodes as `AConcreteSubGen` was in Fig3.1

VDM is made up of a number of core modules, seen in Fig3.3, which all work to build and manipulate the AST. It is worth mentioning that these modules are implemented in Overture, the Eclipse based IDE and VDM-SL platform. This project branches

[1]Abstract Syntax Tree

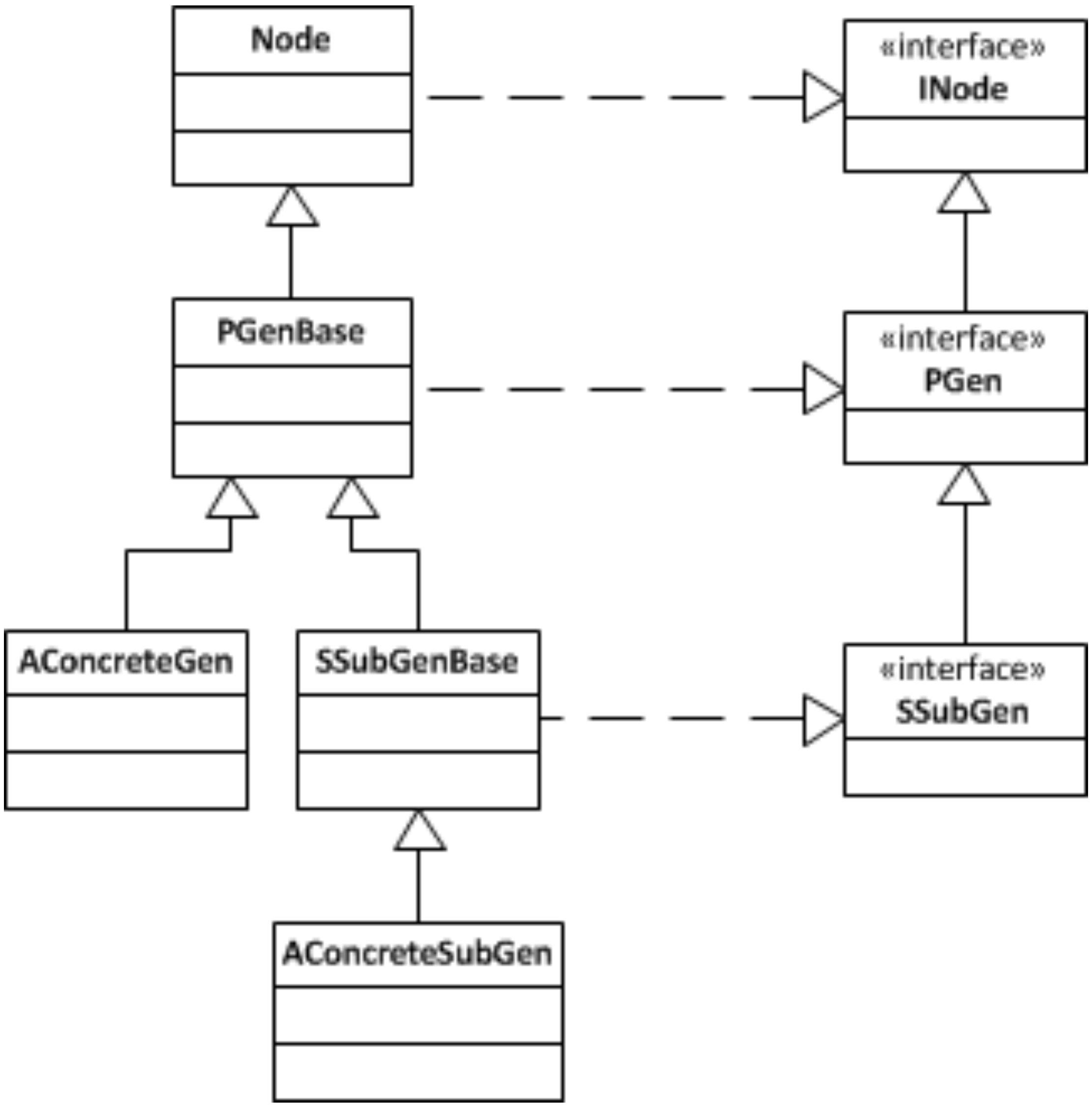

Figure 3.1: The AST's General Architecture [Cou19b]

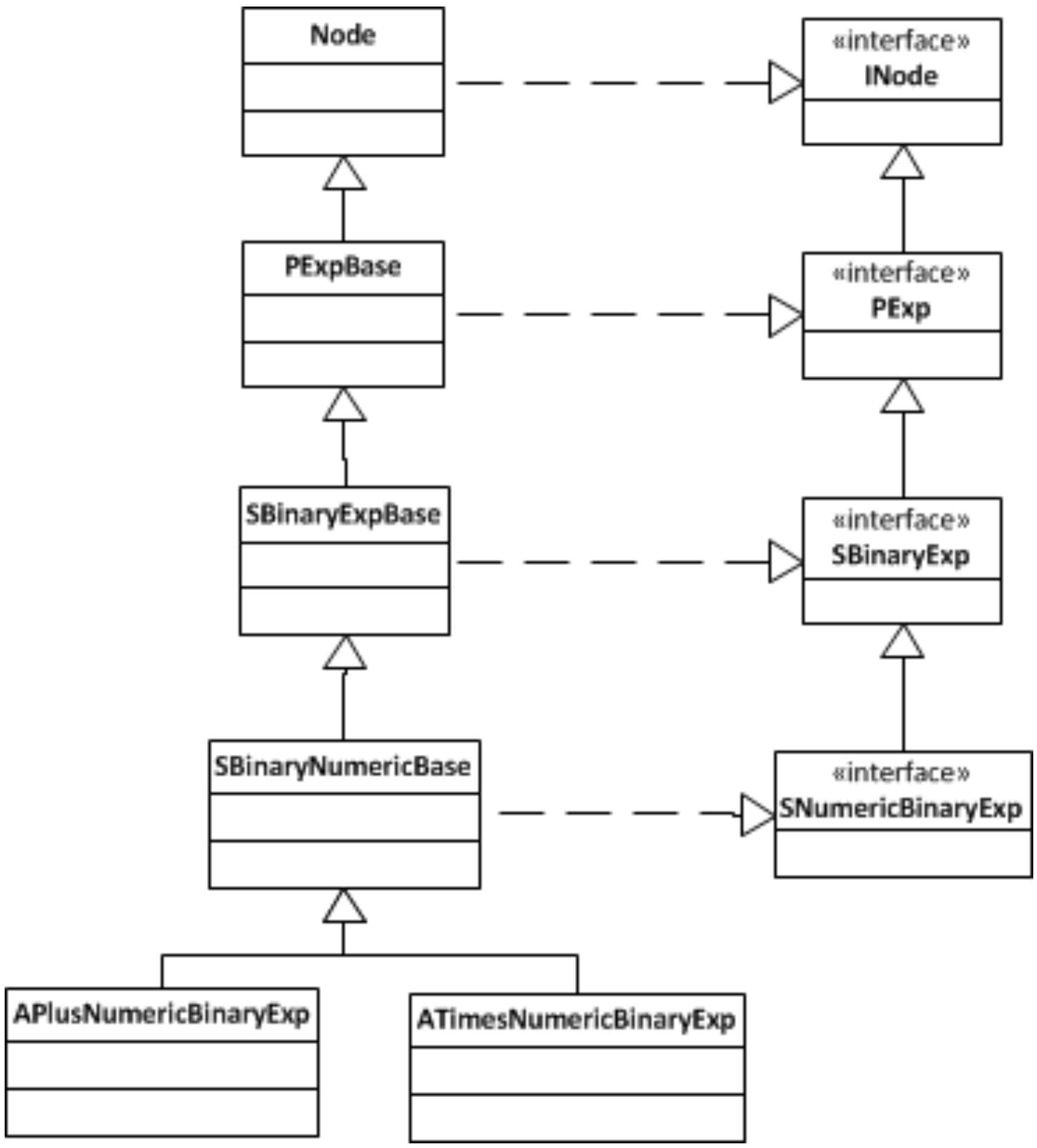

Figure 3.2: The AST hierarchy for a binary expression, in this case plus and times. [Cou19b]

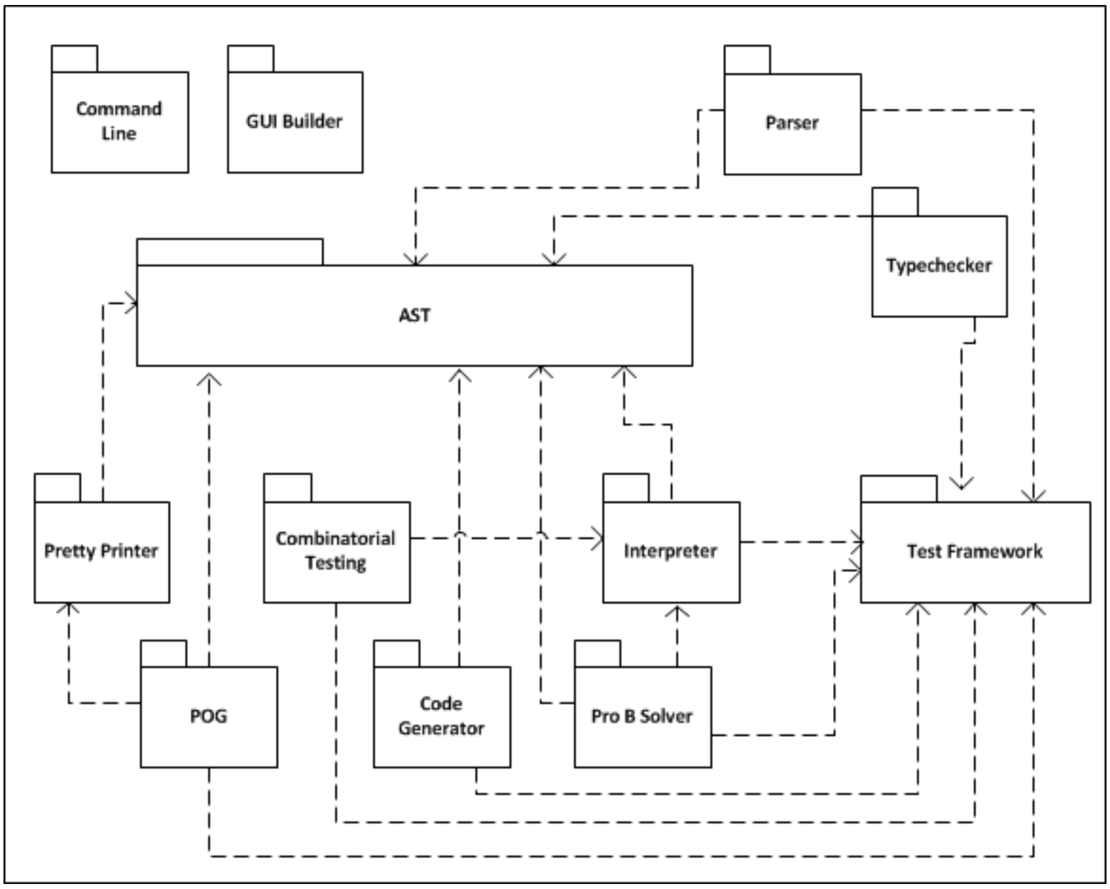

Figure 3.3: The Architecture of Core VDM Modules [Cou19a]

from the Overture GitHub, namely the cth/isagen branch maintained by Casper Thule Hansen,[Gitnt] and the entire development is an extension of the Overture package's `Code Gen` `module.`. Fundamentally, all modules are built around the Abstract Syntax Tree (AST), all modules are written in pure Java. A very brief overview of the most important modules is as follows:

- The `Parser` reads a VDM model and constructs its AST.
- After the operation of the `Parser` , an AST is built. The `Typechecker` module validates this AST and assigns types to each node, one such check might test if a given union type contains a Boolean type. An output of this stage is an Intermediate Representation, see 3.1.2.
- The `Interpreter` module is responsible for executing the AST constructed from VDM Models by the `Parser` , and interacting with the user.

The main module of interest to this project in particular however; is the `Code Generator` , in which lies the base `Code Generator` that generates an Intermediate Representation (IR), and the `IsaGen` package which manipulates that IR and contains all of the code written for the entirety of this project.

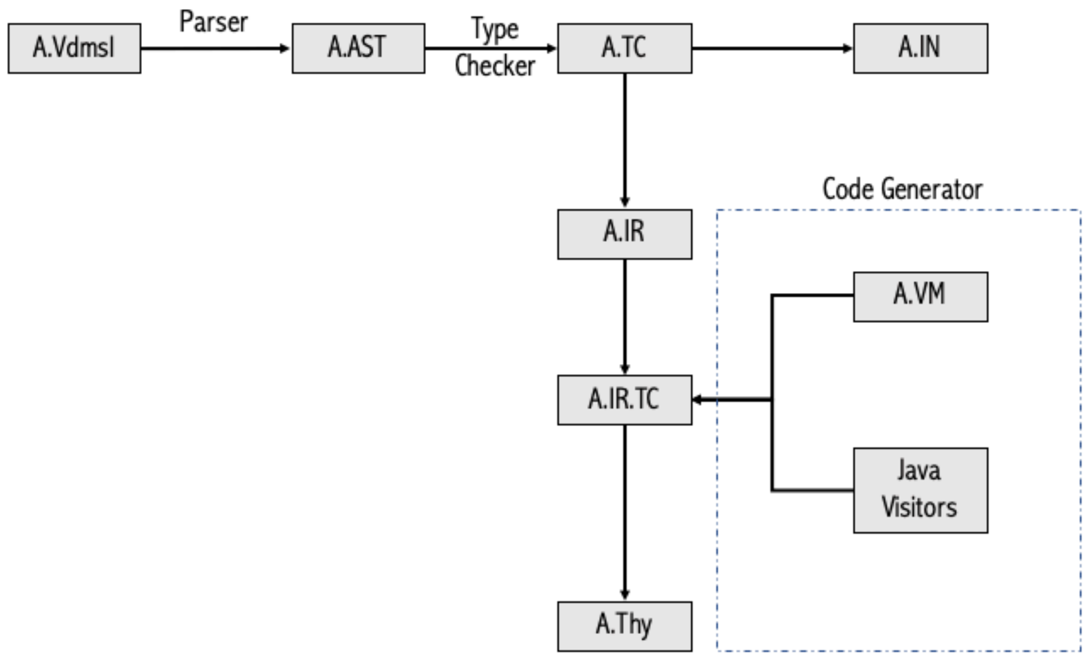

Figure 3.4: The output progression of the A.vdmsl file.

3.1.2 Code Generator & Intermediate Representation

After the `Parser` , `Typechecker` and their subsequent modules have completed operation [2], an Intermediate Representation (IR) is output. An IR is commonly used in programming languages to provide extensibility for Code Generators such as the Isabelle/HOL generator in question. An IR is a representation of a program between the source and target programming languages; after VDM has been compiled into an AST but before it is translated into Java or C++, or HOL here. The IR is independent of its source or target languages and is meant to provide a common ground, an interface of sorts between different tools and languages operating on one AST, similar is the way that Latin is used in the classification of wildlife, `AAndBoolBinaryExpIR` IR node is an `AAndBoolBinaryExpIR` node in both VDM and Isabelle ASTs. The IR allows the translation tool to analyse and manipulate an AST without repeatedly translating to and from VDM and Isabelle and allows the tool to have a pre-type-checked AST to work with. This is beneficial as we do not need to type check again in Isabelle after translation, the translation is broken up into manageable parts. A type-checked binary expression in IR is still a validated and type checked binary expression in Isabelle, or else it would not have been parsed to the IR AST by the `Parser` and `Typechecker` .

The `Code Generator` module contains a base `Code Generator` class which is responsible for providing access to the IR. Sub classing this class gives access to the IR and some valuable AST settings information. This Isabelle/HOL translator, which works within the `Code Generator` module, approximately follows a five-step methodology originally proposed in the 13th Overture Workshop[IG15] and reads like so: [1] set-up, [2] add new nodes, [3] transform the IR, [4] generate syntax, [5] validate the translation.

[2]But before any output is produced by the `Interpreter` or its subsequent modules.

3.2 Methodology

This section will describe the established methodology closely followed by this project as well as explanations of tools and technologies involved in its implementation.

3.2.1 Step 1. Set-up a CGP extension

Have a class which extends the `CodeGenBase` class and create a new template manager which will provide access to the template structure class provided by the `Code Generator` module. Lastly, set up a basic testing facility, Overture, the Eclipse based IDE and VDM-SL interpreter utility in which we write VDM-SL, provides a test framework, this should be used to set up the tests.

3.2.2 Step 2. Add New Nodes

If the target language is significantly different from the first, then new nodes may need to be created and added to the AST. In the case of Isabelle, new nodes were needed quite often. In fact, almost every operation required addition of new Expression, Definition or Pattern nodes to the IR AST. This is because, as mentioned previously, an Isabelle translation requires explicit translation of every construct, for example invariants, pre and post conditions are translated as separate functions - contrary to how VDM handles say, functions, with pre and post conditions attached to the definition as VDM AST node fields.

3.2.3 Step 3. Transform the IR

This step transforms the AST so that the target language can be generated from it in the next stage. Constructs that are not supported in the target language are transformed away and fields and nodes are adjusted, or constructs removed altogether. There are two types of transformation applied to the AST:

- A partial transformation. A partial transformation is trivial to apply but is limited in the way that it can only change the internal structure of the node - like fields, method overrides and so on. A partial transformation does not make large changes to the AST and so does not interfere greatly with its structure. Rarely, only minor adjustments need be made to an ancestor or a child of a partially transformed construct and this is often easily or even automatically achieved with implemented get and set methods.

- A total transformation. Total transformations can change the node itself; the node can be transformed into a different node through type casts for example, removed and replaced with a different node or removed from the tree entirely. Total transformations are difficult to perform and, as I discovered during my implementation of the tool, volatile and vulnerable to anomalous behaviour, sub nodes of the node must on occasion be subsequently changed significantly according to the changes made to their ancestor. The following code snippet from the overture git shows how to apply transformations.

```
     List<ExtIrClassDeclStatus> transformed = new Vector<>();

   for (IRClassDeclStatus status : untransformed)
   {
     // Partial transformation applied directly
     PartialTransformationFoo t1 = new PartialTransformationFoo();
     generator.applyPartialTransformation(status, t1);
```

```
        // Total transformations need wrapping
        ExtIrClassDeclStatus eStatus = new ExtIrClassDeclStatus(status);
        TotalTransformationBar t2 = new TotalTransformationBar();
        generator.applyTotalTransformation(eStatus, t2);
        transformed.add(eStatus);
    }
```

[Tra15]

Total transformations need extra work within the `TotalTransformationBar` to harmonize the tree with the change to its nodes. Transformations themselves take the form of visitor classes, the `PartialTransformationFoo` class above is a visitor class. The visitor pattern is important and so it is given its own section in 3.2.3.

The Visitor Pattern

Manipulating the VDM AST directly is highly discouraged, the system of modules and their dependencies is far too complex to handle direct changes to its central component. The idea of a visitor design pattern is to detach an algorithm from the data that it operates on, in this case to detach the VDM AST from code generation transformations performing potentially erroneous meddling. As mentioned in section 3.1, the `INode` interface that is the highest level of abstraction in the VDM AST enforces `apply()` methods. All Nodes in the AST have `apply()` methods as they enable the visitor pattern. Where before, something like `node.changeParent()`, becomes instead `node.apply(parentChangeVisitor)` where `parentChangeVisitor` is a Java class with methods that change the parent of a node. Almost every time that we interact with the AST this pattern is used. The `apply()` method, implemented by each node, allows each node of the AST to control the way in which it is individually transformed according to its own subtle intricacies that would not be, nor should have to be, strictly adhered to by external classes. The `apply()` method, and by extension the visitor pattern, hence gives control of any interaction with the AST, over to the AST nodes' classes, each of which know precisely how to handle interaction safely in their own unique way, because of this, the AST is never damaged or malformed.

The AST classes will only accept a class in their `apply()` functions if that class is a visitor. A class is identified as a visitor by extending, sub-classing, an adaptor class in the `org.overture.ast.analysis` package, an adaptor class **adapts** AST interaction by visitors into a safe one. More on the various analysis adaptor classes in this package is discussed later in this section. In order for a visitor to be able to interact with different types and families of nodes, they must be equipped with a method containing functionality for the *case* that certain nodes might be encountered.

Cases are a powerful construct within the visitor pattern, they allow the programmer to do away with exceedingly long control blocks used to test for the presence of specific nodes, any large code blocks can be stored in a separate visitor class containing case methods for what to do for each different node family or type. Each class in the `org.overture.ast.analysis` package is filled with case methods for every type of node. The `DepthFirstAnalysisAdaptor` class contains approximately $16,000$ lines of code the majority of which are empty or basic case methods which are overridable so that the sub-classed visitor can add its own functionality for that node case. Each method takes as a parameter, a node of the type that it is the case for. To name but a minute few in this class:

```
caseARealPatternIR(ARealPatternIR)
caseARecordDeclIR(ARecordDeclIR)
```

```
caseARecordModExpIR(ARecordModExpIR)
caseARecordModifierIR(ARecordModifierIR)
caseARecordPatternIR(ARecordPatternIR)
caseARecordTypeIR(ARecordTypeIR)
caseARemNumericBinaryExpIR(ARemNumericBinaryExpIR)
caseARenamedDeclIR(ARenamedDeclIR)
caseARepeatTraceDeclIR(ARepeatTraceDeclIR)
caseAReturnStmIR(AReturnStmIR)
caseAReverseUnaryExpIR(AReverseUnaryExpIR)
caseASameBaseClassExpIR(ASameBaseClassExpIR)
caseASameClassExpIR(ASameClassExpIR)
```

If, for example, the programmer would like to create a visitor that counts the number of fields in `ARecordDeclIR`[3] then they would write a class like so:

```
public class EmptySeqVisitor extends AnalysisAdaptor {

  int nodeCount;

  @Override
  public void caseARecordDeclIR(ARecordDeclIR node)
      throws AnalysisException {

    nodeCount = node.getFields().size();
    //_fields is a NodeList<AFieldDeclIR>() field in ARecordDeclIR. NodeList
        is a VDM class in the AST package implementing list functionality.

  }

}
```

The `org.overture.ast.analysis` package contains additional classes, adaptors, to `AnalysisAdaptor` which can be extended to provide various AST interactions, parameter passing, return values, or both.

1. `AnalysisAdaptor` The easiest way to create a visitor to the AST, it takes no parameters and the return type of its overridden methods are always void. Such a visitor can be used, for example, to count occurrences of a type of node by incrementing a field as the visitor encounters a node of a certain type.

2. `QuestionAdaptor<Q>` Allows the visitor to pass information to the AnalysisAdaptor as a generic type parameter, though this parameter must be the same for all cases in a given visitor. In a question visitor the parameter object `<Q>` is passed as a second parameter to every method in the visitor. It can be used to check the equality of a node or its fields, to set a parameter of a node to a different value and so on.

3. `AnswerAdaptor<A>` Allows the visitor to define a return type and gather data from the AST so long as the data is of type `<A>`. Every method in the visitor has a return type of `<A>`. Extending this class requires the visitor to implement two additional methods `createNewReturnValue(INode node)` and `createNewReturnValue(Object node)`, as something must be returned, these methods are invoked when no other case is matched.

4. `QuestionAnswerAdaptor<Q,A>` As may be evident from the name, a visitor that is a subclass of this class can both pass parameters of type `<Q>` and take a return type of

[3]The IR node for a declaration of a record type.

`<A>`. This class allows for the most flexibility of the three and must implement the same methods as the proceeding classes in this list.

5. `DepthFirstAnalysisAdaptor` This is used frequently in this project; this adaptor allows a visitor to perform a depth first tree traversal of the AST and perform analysis as it does so. A visitor that is a subclass of this class is given the same methods to implement and override as the `AnalysisAdaptor` class with the addition of `public void setVisitedNodes(Set<INode> value)` and the field `_visitedNodes` which allow sub-classed visitors to access the nodes visited during the depth first tree traversal. This is a particularly useful class for modifying the AST, nodes that need to be worked on can be on wildly different sub-trees due to the differences between Isabelle and VDM.

6. `DepthFirstAnalysisAdaptorAnswer<A>` Performs a depth first analysis of the tree with the addition of answer functionality described in `AnswerAdaptor<A>` above.

7. `DepthFirstAnalysisAdaptorQuestion<Q>` Performs a depth first analysis of the tree with the addition of question functionality described in `QuestionAdaptor<A>` above.

8. `DepthFirstAnalysisAdaptorQuestionAnswer<Q,A>` Performs a depth first analysis of the tree with the addition of question/answer functionality described in `QuestionAnswerAdaptor<A>` above.

3.2.4 Step 4. Generate Target Language Syntax

Now that the AST has been transformed into a state ready to be translated into an AST of its target language, it is passed into the syntax generation framework of the `Code Generator` module. At this stage the `Code Generator` module traverses the IR AST and creates translated target language code as a String for each node. The `Code Generator` detects the node's type and then creates code for it by populating pre-defined Apache Velocity templates, see 3.2.4 for a description of Apache Velocity and its application to this project.

Apache Velocity

Apache Velocity is a Java based template engine in which a `.vm` file is written containing Velocity syntax, the velocity syntax accesses variables defined in Java code and allows the programmer to interweave their values with appropriate syntax, lexicon and document structure to create the desired output string or file. Some example Velocity syntax is:

```
    ($Isa.trans($node.Left) \<and> $Isa.trans($node.Right))
```

This creates essentially place holders for the result of Java computations that will create a value for a Java translation method `trans(INode node)` which will have passed into it the node which is the left hand side value of the binary `and` expression followed by the Isabelle/HOL symbol for `and` `\<and>`[4] followed by the node which is the right hand side of the `and` expression - we will get something like $A \wedge B$ in Isabelle. The and template is a trivial example but templates as complex as entire function definition structures can be achieved.

[4]which will be translated into the HOL symbol $\wedge$

Velocity

Listing 3.1: The Velocity template for function declarations. Below is a translation showing how one such template is used in practice.

```
#macro ( transIdentifiers $node )
#foreach($p in $node.FormalParams)
$Isa.trans($p.pattern)##
#end
#end

definition
#if ("$Isa.transTypeParams($node.MethodType.Params)" == "")
  $node.Name :: "$Isa.trans($node.MethodType.Result)"
#else
  $node.Name :: "$Isa.transTypeParams($node.MethodType.Params) \<Rightarrow>
      $Isa.trans($node.MethodType.Result)"
#end
   where
   "$node.Name #transIdentifiers($node) \<equiv> $Isa.trans($node.Body)"
```

Translation

```
theory A
    imports VDMToolkit
  begin


  definition
    f :: "VDMNat
   \<Rightarrow> VDMNat
   \<Rightarrow> VDMNat
  "
      where
      "f x y  \<equiv> x"


  definition
    pre_f :: "VDMNat
   \<Rightarrow> VDMNat
   \<Rightarrow> \<bool>"
      where
      "pre_f x y  \<equiv> (isa_invVDMNat x \<and> isa_invVDMNat y)"


  definition
    post_f :: "VDMNat
   \<Rightarrow> VDMNat
   \<Rightarrow> VDMNat
   \<Rightarrow> \<bool>"
      where
      "post_f x y RESULT  \<equiv> (isa_invVDMNat x \<and> (isa_invVDMNat
           y \<and> isa_invVDMNat RESULT))"
```

```

     end
```

By this mechanism, the `Code Generator` module traverses the AST and populates the templates while the subsequent steps transform the nodes in the AST by computing and manipulating them in order to provide Velocity with the correct variable values. The purpose of the subsequent steps are to prevent the programmer from having to write large velocity files, a well organised visitor pattern and translator should make their target velocity templates as short as possible, one praise of this project is that the velocity files never exceed any more than 19 lines of code and are approximately 2.67, rounded to 3, lines long on average, this includes white-space. Fig3.5 shows an in detail diagram of these steps' outputs.

3.2.5 Step 5. Validate

The validation step involves comparison of the output with a known correct result file. Result files are kept very simple and each construct is separate, for example we have one file to test `AIntNumericBasicTypeIR` node translation and it is kept as simple as possible. All test files have the `.vdmsl` file extension and have one corresponding `.vdmsl.result` file which contains the expected translation of a construct. For example, the file `Int.vdmsl` has a corresponding result file `Int.vdmsl.result`.

```
types
XType = int;
```

Listing 3.2: Int.vdmsl

```
{"translation":"theory DEFAULT
imports VDMToolkit
begin

type_synonym XType \= \"VDMInt\"




definition
   inv_XType :: \"(XType) \\<Rightarrow> \\<bool> \"
   where
   \"inv_XType x \\<equiv> isa_invTrue x\"

end","errors":false}
```

Listing 3.3: Int.vdmsl.result

A test passes if the output of the first file, after running through all of the core modules and the translation tool, matches the second.

3.3 Previous Tool Implementation Work

3.3.1 Previously Completed Step [1] Set-up A CGP Extension

Before starting this project, Casper Thule Hansen, PhD Researcher of Aarhus University Denmark[Cas], had completed the important and tricky first step of the methodology described in 3.2.1.[Gitnt] . As mentioned in 3.2.1 this is the first step in creating a CGP (Code Generation Platform) extension.3.5 `IsaGen`, extends the `CodeGenBase` class and the `IsaTranslations` class serves as a template manager, this class provides access to the template structure provided by the `Code Generator` module by creating a `MergeVisitor` object as a field as well as a list of callable templates, `TemplateCallable[] templateCallables` from the `org.overture.codegen.merging.TemplateCallable` package in the CGP module, for accessing the various Velocity templates for each node.

The `IsaGen` class overrides the `genVDMToTargetLang` method[5] in `CodeGenBase`. This method is used by `CodeGenBase` and the CGP to generate the target language data. By overriding it, it becomes our main point of interaction with the CGP and so this is where we perform our transformations. We do so by placing visitor classes in this method, taking `status` as an argument.

Once this method has initialised the fields of a `GeneratedModule` object, it is returned to the calling method in `CodeGenBase`, `generate()`, which is used by `CodeGenBase` in a series of steps by the CGP to create the target language output.

The `getInfo()` method provides access to AST settings while the parameter `List<IRStatus<PIR>> statuses` provides access to the AST itself. The method generates the IR using the CGP and Typechecker module through several calls to CGP methods during the lambda function at the top of the method.

As the final step in stage one, a test framework had also been set up. The test that I used was the JUnit class `IsaGenParamTest`. The test strategy was that all tests should pass and before this project one already did, integer, as described later in this section[6]. It is necessary to understand that this test class was set up as an array of skipped tests, the strategy was to activate a test, make it pass and then continue to the next.[7]

3.3.2 Previous Work on Step [2] Add New Nodes & Step [3] Transform The IR

Some parts of the transformation step described in 3.2.3, were also implemented prior to this project. As the tool goes over the AST it applies visitor classes to its nodes. Total transformations are applied to each status, by the `GroupMutRecs` visitor, recursion cycles in the AST's structure are transformed away before translation to Isabelle. The `SortDependencies` visitor is applied to `AModuleDeclIR` nodes, this sets up the theory file, dependencies, imports, name etc. . The `AModuleDeclIR` node is the highest level element of the theory file, it is the file definition that encase all declarations within it, in Isabelle it is marked by the `theory` keyword at the top of a `.thy` file. In this visitor we have the case:

```
@Override
 public void caseAModuleDeclIR(AModuleDeclIR node) throws AnalysisException {
     result = new AModuleDeclIR();
     result.setExports(node.getExports());
     result.setImport(node.getImport());
     result.setIsDLModule(node.getIsDLModule());
     result.setIsFlat(node.getIsFlat());
     result.setMetaData(node.getMetaData());
     result.setName(node.getName());
     result.setSourceNode(node.getSourceNode());
     result.setTag(node.getTag());
     result.setDecls(node.getDecls());
```

[5]See appendix A, section A.2.1to see `genVdmToTargetLang` before development, and appendix B, section B.2.1, for after development

[6]The full class can be seen in appendix C

[7]See appendix C for the full code.

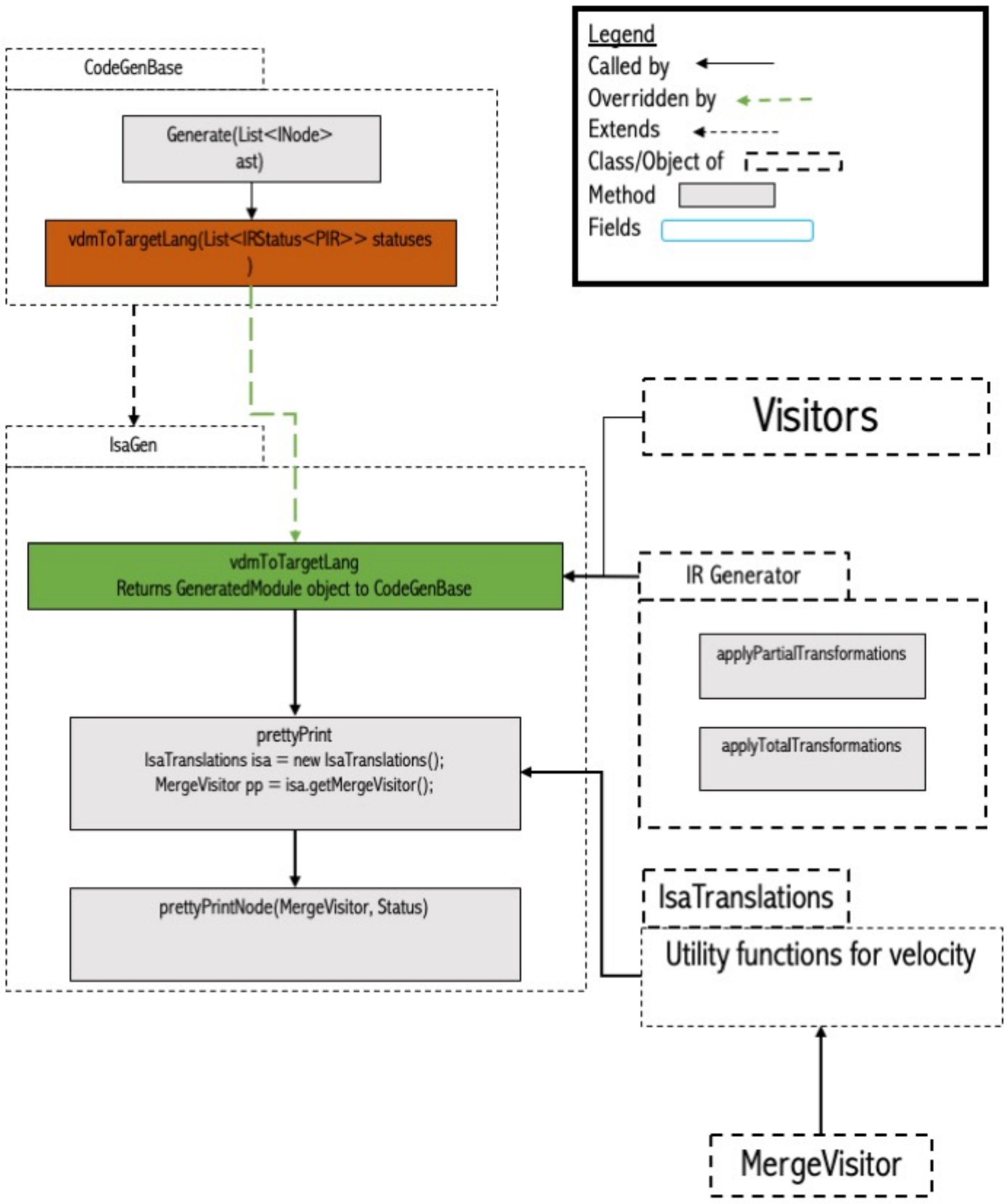
Legend
Called by
Overridden by
Extends
Class/Object of
Method
Fields
CodeGenBase
Generate(List<INode> ast)
vdmToTargetLang(List<IRStatus<PIR>> statuses)
IsaGen
vdmToTargetLang
Returns GeneratedModule object to CodeGenBase
prettyPrint
IsaTranslations isa = new IsaTranslations();
MergeVisitor pp = isa.getMergeVisitor();
prettyPrintNode(MergeVisitor, Status)
Visitors
IR Generator
applyPartialTransformations
applyTotalTransformations
IsaTranslations
Utility functions for velocity
MergeVisitor

Figure 3.5: The tools structure at the set up stage. [Cou19b]

```
        filterFunctions(node.getDecls());
        calcDependencies();

    }
```

This clarifies the above, the module declaration, i.e. the file declaration, is set up by setting the above fields. The `_decls`, declaration, child is particularly important, it holds all declarations in the file as a list. Adding a declaration to the `AModuleDeclIR` 's `_decls` list field adds it to the AST, and so the file, this will be seen later as it is the mechanism used to add to the AST. In short, the `AModuleDeclIR` ancestor of a node is retrieved, the `_decls` field is accessed using a get method, then a declaration is added to it.

With the module declaration set up, visitor classes are then free to be applied to the AST nodes. Previous work had set up one example of how to manipulate the AST to set up classes properly,

```
    IsaBasicTypesConv invConv = new IsaBasicTypesConv(getInfo(),
        this.transAssistant, vdmToolkitModuleIR);
    generator.applyPartialTransformation(status, invConv);
```

The visitor class `IsaBasicTypesConv` contains a case for `caseAIntNumericBasicTypeIR` which translates VDMToolkit's VDMInt types.[8] The constructor of this visitor has a pattern that is commonly repeated throughout the visitors in this project. A lambda function goes through all of the type declarations in the VDMToolkit and maps their name to their IR node. Example:

```
   {isa_VDMNat=privateATypeDeclIRisa_VDMNatint,
   isa_VDMInt=privateATypeDeclIRisa_VDMIntint,
   isa_VDMNat1=privateATypeDeclIRisa_VDMNat1int}
```

A description of the functionality provided by these classes will be present in section 4.1, as this document explains changes made by the development process, so will it describe, by proxy, the similar mechanisms previously implemented here.[9]

Invariant Generation

Before development, the `IsaInvGenTrans` invariant transformation visitor had been established. This visitor generates nodes that will contribute to building the invariant function declaration for a type.[10] This visitor grows significantly over the course of ensuing development as it is used to generate invariants differently per case, further case methods for field declarations[11] are added later. As with `IsaBasicTypesConv`, a map for VDMToolkit types is initialised. Through an identical mechanism, this constructor has the addition of a map initialisation for VDMToolkit function types.

```
   this.isaFuncDeclIRMap = this.vdmToolkitModule.getDecls().stream().filter(d ->
           {
               if (d instanceof AFuncDeclIR)
                   return true;
               else
                   return false;
           }).map(d -> (AFuncDeclIR) d).collect(Collectors.toMap(x ->
               x.getName(), x -> x));
```

[8] See appendix A, section A.1.3
[9] Appendix A, shows all of this code and classes at the start of the project.
[10] See appendix A, section A.1.2 for the before state of `IsaInvGenTrans`
[11] VDM value declarations

A lambda stream filters through all function declarations, of node type `AFuncDeclIR`, in the VDMToolkit IR. These nodes correspond to invariant functions, and so this map is essentially an map of VDMToolkit invariant names to their declaration.

A case method for all type declarations was established that would generate invariants for a type declaration node.

```
  @Override
     public void caseATypeDeclIR(ATypeDeclIR node) throws AnalysisException {
```

The state of this case method before development can be found in the appendices, appendix A, section A.1.2. Essentially, the method specified that if no invariant existed, a new function declaration node would be created along with a number of other nodes. These nodes would construct the fields of the function declaration before it is added to the AST, in the body of the generated invariant function, a check that `isa_invTrue x` held of the type declaration. The details of these computations, and the resulting construction of invariant functions, are not detailed here as they are explained thoroughly in 3.4.4.

This method was limited in that it only generated invariants if none were there, where it could have concatenated existing invariants with additional generated necessary checks. At this stage the visitor only supported type declarations with this functionality.

3.4 Work Done in This Project

As step one was entirely set up by Casper Thule Hansen previously, the implementation of each construct starts at step [2] 3.2.3 and goes through to step [5]3.2.5. This section will not discuss every change made to achieve translation for each construct, but instead focus on implementation issues and headlines. This is because some additions or modifications are similar, and represent the same implementation issues and headlines. As such, the following section is structured like so: firstly, a more in depth overview of the development strategy; then, a test representing a VDM construct from `IsaGenParamTest` will be activated, if there are relevant, and as yet uncovered programming issues and headlines they will be described, if not this construct test will be skipped.

3.4.1 Development Strategy

The goal of development was that all `IsaGenParamTest` tests should pass, development only progressed when a test passed, in that sense **development was test driven**. I started with what should be the top of the VDM file, types, so that my approach was structured in a way that complimented the functional programming rules that defined my source and target languages. After, I moved on to values, then functions, then state. At each stage I repeated the five step methodology, previously discussed, in a cycle. I first transformed the IR to translate no more than what was already in the IR AST into Isabelle; next, I generated additional constructs, such as invariant function body expressions, to support the translation and then added them to the AST; I passed the resulting IR AST to the `IsaGen` class and the `applyPartialTransformations` method to generate the Isabelle syntax, at this stage I also added any necessary Velocity code, although this was rare; finally I validated the generated syntax with the relevant test.

3.4.2 No Invariant, Basic Numeric Type Translations

For basic numeric types, generating their translation was the same.[12] All VDMToolkit types were simply added as cases to the `IsaBasicTypesConv` class. The code within them

[12]See appendix B, section B.1.3.

was each time identical to the caseAIntNumericBasiTypeIR, with the exception of their VDMToolkit name. To enable the function of these classes, I added string constants as fields of the class as with the private final static String isa_VDMInt = "isa_VDMInt"; field.

Nat1 Example

private final static String isa_VDMNat1 = "isa_VDMNat1"; was added as a field in IsaBasicTypesConv as well as the case method:

```
  //transform nat1 to isa_VDMNat1
   public void caseANat1NumericBasicTypeIR(ANat1NumericBasicTypeIR x){
       if(x.getNamedInvType() == null)
       {
           // Retrieve isa_VDMNat1 from VDMToolkit
           ATypeDeclIR isa_td =
               isaTypeDeclIRMap.get(IsaBasicTypesConv.isa_VDMNat1);

           x.setNamedInvType((ANamedTypeDeclIR)isa_td.getDecl().clone());
       }

   }
```

Because, IsaBasicTypesConv extends DepthFirstAnalysisIsaAdaptor, this visitor traverses the AST, and as it encounters a ANat1NumericBasicTypeIR node, this case is executed. The line x.setNamedInvType((ANamedTypeDeclIR)isa_td.getDecl().clone()); sets the invariant type of the ANat1NumericBasicTypeIR node x, to the type declaration value in the map created in the constructor, described in 3.3.2. This value is safe copied with the clone() method which must be implemented by all implementing classes of the SDeclIR interface i.e. by all declaration nodes. copy() should return a deep copy of its enclosing class instance. Later in development, I encountered severe and difficult to debug issues spawned from Null Pointer Exceptions that were the result of failing to call copy() on nodes, copy() proved to be arguably the most important method in the entire AST.

The returned object of all of this computation, is cast to ANamedTypeDeclIR for two reasons, this is the type of the _namedInvType field and getDecl() returns SDeclIR, that is the _decl child of the ATypeDeclIR object isa_td. SDeclIR is the interface for all declaration family nodes in the IR AST and it will be used extensively during the following development.

3.4.3 No Invariant, ”Type Type” (Collection) Translations

ASeqSeqTypeIR and ASetSetTypeIR were translated using a new visitor class IsaTypeTypesConv, so called for the nomenclature of the IR nodes involved. There was no obligation to have these two IR nodes' translation functionality in a separate visitor, they could have been written comfortably as cases in the IsaBasicTypesConv class, however I did so in the interest of separating concerns, making the program easier to debug and read. As IsaBasicTypesConv, IsaTypeTypesConv repeats code for each node case method in the class.[13] For that reason, I will show the example of ASetSetTypeIR.

Set Type Example

caseASetSetTypeIR was added to the IsaTypeTypesConv class which has approximately identical constructor and fields to IsaBasicTypesConv, the implementation is the same

[13]To see the whole class please refer to appendix B, section B.1.4.

also. Although there are no new issues or headlines in this case, it was added to show the benefits of the VDMToolkit, a collection can be added in the same way as a basic type such as `AIntNumericBasicTypeIR` as the translation to Isabelle and type-checked IR negates us having to do so ourselves.

```
//transform seq into VDMSeq
public void caseASetSetTypeIR(ASetSetTypeIR x) {
  if(x.getNamedInvType() == null)
    {

       // Retrieve isa_VDMSet from VDMToolkit
       ATypeDeclIR isa_td = isaTypeDeclIRMap.get(IsaTypeTypesConv.isa_VDMSet);

       x.setNamedInvType((ANamedTypeDeclIR)isa_td.getDecl().clone());

    }
}
```

The difference in this implementation is that as it is a separate visitor, a separate transformation, it must be added to the `genVdmToTargetLang` method so that it can be applied to the AST by the `applyPartialTransformation()` method of the `IRGenerator generator` field in the `IsaGen` class.

```
 // Transform Seq and Set types into isa_VDMSeq and isa_VDMSet
  IsaTypeTypesConv invSSConv = new IsaTypeTypesConv(getInfo(),
      this.transAssistant, vdmToolkitModuleIR);
  generator.applyPartialTransformation(status, invSSConv);
```

Applying these transformations uses the exact same code in the `genVdmToTargetLang` class for each transformation visitor for the rest of the development.[14]

3.4.4 Invariant Transformations & Generation

Type declarations need invariants in Isabelle, as does almost every declaration that is not a function declaration. At the beginning of development, a transformation visitor for generating the comprising IR nodes of an invariant had already been established, see 3.3.2. As mentioned in 3.3.2 the case was limited in that it only handled invariants when they were not present, and could only generate `isa_invTrue` invariants. Development of this stage aimed to add support for generating invariant checks that would validate each primitive type constituting the type declaration. Next, translating both a generated invariant and an existing one.[15].

```
 String typeName = IsaInvNameFinder.findName(node.getDecl());
  SDeclIR decl = node.getDecl().clone();

  SDeclIR invFun;
  if (node.getDecl() instanceof ARecordDeclIR)
    invFun = ( (ARecordDeclIR) decl).getInvariant();
  else
    invFun = node.getInv();
  // Invariant function
  AFuncDeclIR invFun_ = new AFuncDeclIR();
  invFun_.setName("inv_" + typeName); //inv_t
```

[14]See appendix B, section B.2.1.

[15]The final case method after consequential development is shown in appendix B, section B.1.7

```
   // Define the type signature
   AMethodTypeIR methodType = new AMethodTypeIR();

   STypeIR t = IsaDeclTypeGen.apply(decl);
   methodType.getParams().add(t.clone());


  methodType.setResult(new ABoolBasicTypeIR());
   invFun_.setMethodType(methodType);
```

This block of code is taken out of the control block and is therefore executed regardless of weather an invariant for a type already exists or not. This is done because in both cases we still want to generate some extra invariant checks on types in order to aid proof in Isabelle later on.

1.
```
{String typeName = IsaInvNameFinder.findName(node.getDecl());}
```

   Assign to `typeName`, the result of the `IsaInvNameFinder` visitor utility class `IsaInvNameFinder`[16]. `IsaInvNameFinder` gets the name of the type declaration node with a call to a few straightforward get methods.

2.
```
SDeclIR decl = node.getDecl().clone();
```

   Assign to `decl`, The `SDeclIR` value of the `_decl` child of the type declaration node and clone it. If the clone method is not used here `decl` evaluates to `null` at later computation. Such an error at one point in the project, took an entire day to track down.

3.
```
SDeclIR invFun = node.getInv();
```

   We declare an `SDeclIR` node and assign to it the `AFuncDeclIR` node that is the `_invFun` child of this type declaration node. If there is not invariant defined then `invFun` will evaluate to `null`.

4.
```
// Invariant function
 AFuncDeclIR invFun_ = new AFuncDeclIR();
 invFun_.setName("inv_" + typeName); //inv_t
```

   Now the invariant function that will become the generated invariant is created and initialised. Its name is set to `"inv_"+ typeName)`, a `String` concatenation of the keyword `inv_` that is the standard nomenclature for Isabelle invariant translations, and `typeName`, retrieved above. For example, for the `type_synonym t`, the invariant function name is `inv_t`.

```
 type_synonym t = "VDMNat"

definition
   inv_t ....
```

```
// Define the type signature
 AMethodTypeIR methodType = new AMethodTypeIR();
```

[16]See appendix A, section A.1.6 for the before development state of this visitor.

```
     STypeIR t = IsaDeclTypeGen.apply(decl);
     methodType.getParams().add(t.clone());


    methodType.setResult(new ABoolBasicTypeIR());
     invFun_.setMethodType(methodType);
```

Each function declaration in the IR AST must have a method type. `AMethodTypeIR` is of the `STypeIR` family of nodes and is eventually set as the `_methodType` child of the `AFuncDeclIR` node. `AMethodTypeIR` nodes are made up of parameters, their `_params` child, and a result, the `_result` child[17]. As such, they make up the signature of a function, its parameters and return type. The `_params` child of the method type takes only type family nodes, those that implement the `STypeIR` interface and therefore a utility visitor, `IsaDeclTypeGen`,[18] is used to create a type node corresponding to the type declaration node for which the invariant function is being generated. `IsaDeclTypeGen` will be detailed further during discussion of later development.

At this point the invariant function declaration has been set up and needs a function body. There are two conditions that were added during development, one for an existing invariant and one for no existing invariant.

Generation for existing invariant functions

`if (invFun != null)` is the condition that is met. Below details my implementation of this condition.

5.

```
    AFuncDeclIR inv = (AFuncDeclIR) invFun;//cast invariant function
        declaration to AFuncDeclIR
      AAndBoolBinaryExpIR multipleInvs = new AAndBoolBinaryExpIR();
```

The exiting invariant function is currently stored as `SDeclIR` and so it is cast to `AFuncDeclIR` before being assigned to a new function declaration `inv` so that the `AFuncDeclIR` fields needed to build the invariant can be accessed. A `AAndBoolBinaryExpIR` object is created, it has two fields of note `_left` and `_right` both of these fields are `SExpIR` nodes, meaning that they can be populated with any expression family node.

2.

```
    for (int i = 0; i < inv.getMethodType().getParams().size(); i++)
      {
        AFormalParamLocalParamIR afplp = new AFormalParamLocalParamIR();
          afplp.setPattern(inv.getFormalParams().get(i).getPattern());
          afplp.setType(inv.getMethodType().getParams().get(i).clone());
          invFun_.getFormalParams().add(afplp);
      }
```

This loops through the parameters of the existing function and creates local parameters for functions. Local parameters are those that are used in the invariant body: `"inv_t t \<equiv> isa_invTrue t"`, `t` here, before `\<equiv>`, is a `AFormalParamLocalParamIR` node. For each type in the method type parameters, a new parameter is created, named with the `setPattern()` method to what it is called in the existing invariant,

[17]To clarify once again, it is important to remember that children are nothing more than fields of a node class.

[18]See the state of this utility visitor before development in appendix A, section A.1.4.

its type is set to the parameter it represents and it is added to the `_formalParams` child of the invariant function.

3.
```
     multipleInvs.setRight(inv.getBody());
```

The right hand side of the `AAndBoolBinaryExpIR` expression defined earlier, is set to the existing invariant body expression.

4.
```
      SExpIR expr = IsaInvExpGen.apply(decl.clone(),
              identifierPattern ,
              methodType.clone(), isaFuncDeclIRMap);
```

The `IsaInvExpGen` class is a utility visitor that generates invariant check expressions for a given declaration and will be discussed later[19].

5.
```
      multipleInvs.setLeft(expr);
        invFun_.setBody(multipleInvs);
```

The left side of the and expression is set and the and expression is set to the body of the invariant.

Velocity development

Successful translation of types required additions and changes to velocity, all changes were very similar and straightforward, the change of an `and` keyword to the correct Isabelle `\<and>`, as well as changes to collection templates being written in incorrect order. `VDMSet VDMNat` was changed to the correct `VDMNat VDMSet`. [20]More velocity templates for mathematical constructs were required and written in, like for example `ASetUnionBinaryExpIR` and many more.

Validation & results

For the VDM type declaration below, it is easy to see how the `<\and>` keyword in the `AAndBoolBinaryExpIR` is used:

```
types

t = int
inv t == t > 0
```

We want to generate a check that the type satisfies VDMTrue from the VDMToolkit, and that the type is more than 0. The tool generates:

```
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMInt
"
```

[19]To see the version of IsaInvExpGen before development, see appendix A, section A.1.5

[20]Changes made to velocity can be compared in the appendices, appendix D section D.1, shows the initial velocity templates, appendix D section D.2, shows the velocity templates after development for any interesting changes only.

```
definition
   inv_t :: "(t) \<Rightarrow> \<bool>"
    where
    "inv_t t \<equiv> (isa_invTrue t \<and> (t > 0))"

end
```

3.4.5 Field/Value Declaration Transformations

No additional visitors were added to enable translation of values, this speaks to the robustness of the visitor system. Keeping the architecture of the tool simple, cases were added to existing visitors with their own complex functionality.[21] The `IsaInvGenTrans` case for value declarations required no noteworthy additional functionality. This set of transformations, however; provide the perfect opportunity to demonstrate the operation of the `IsaInvExpGen` utility visitor.

```
SExpIR expr = IsaInvExpGen.apply(node, identifierPattern, mt.clone(),
    isaFuncDeclIRMap);
```

The above line calls the apply method implemented by `IsaInvExpGen`[22] which it inherited from `AnswerIsaAdaptor<SExpIR>`.[23] The `apply(SDeclIR decl, AIdentifierPatternIR afp, AMethodTypeIR methodType, Map<String, AFuncDeclIR> isaFuncDeclIRMap)` method, applies to the passed in `SDeclIR`, declaration node, the `IsaInvExpGen` visitor. In the case of value declarations, the case `public SExpIR caseAFieldDeclIR(AFieldDeclIR node)throws AnalysisException` is executed.

Inside this case method, an expression for all relevant type invariant checks is generated and returned to `IsaInvGenTrans` so that it can be set as the invariant function's body expression.

1.
   ```
   STypeIR t = node.getType().clone();
     AApplyExpIR completeExp = new AApplyExpIR();
   ```

   The type of the value declaration is cloned and stored in a variable `t` so that it is safely cloned for later use. The invariant function of the field[24] must be the invariants of the type of that field, applied to the field itself. For example, for the VDM value:

   ```
   values
   x = 1;
   ```

   The consequent Isabelle translation and invariant should be:

   ```
   abbreviation
    x :: VDMNat1
    where
   "x \<equiv> 1"

   definition
   ```

[21]See appendix B, section B.1.7 to see the `caseAFieldDeclIR` method after development.
[22]The final `IsaInvExpGen` visitor can be found in full in appendix B, section B.1.5
[23]Recall section 3.2.3.
[24]Value and field will be used interchangeably.

```
inv_x :: "\<bool>"
 where
 "inv_x \<equiv> isa_invVDMNat1 x"
```

The invariant for VDMNat1 is applied to the abbreviation/value translation, x. This is easy to achieve when the type is as straightforward as a natural number, however, if we have collections, it becomes difficult. This is because of the way that the IR is structured, for the type `VDMNat1 VDMSet`, the set is an `ASetSetTypeIR` node. This node contains a child `_setOf`, which is a `STypeIR` interface node. This allows the set to contain any type node, including another set. To create an invariant for the type that the set collects we could very simply `getSetOf()` and build the invariant for that type with simple computation. The issue however, is that a set can contain an arbitrary number of nested sets, there is no way of getting the number of these sets efficiently as they are not stored as an iterable, measurable structure in memory. The best way to create an invariant for a type such as `VDMNat VDMSet VDMSet` or `VDMInt VDMSeq VDMSet` is to recursively get the `_setOf` child. We should end up with, for the type `VDMNat VDMSet VDMSet`, an invariant body, `isa_invSetElems isa_invSetElems isa_invVDMNat x` where `x` is the value declaration - so the direction of recursion is also important. The reason that invariant expressions are structured like this, with each nested set type being applied to the one before, is because Isabelle's functions are curried.

Isabelle's curried functions

To properly understand why Isabelle functions are translated in this way we need an understanding of what it means for Isabelle functions to be curried. Without going into too much depth, currying functions means that the evaluation of a function that takes multiple arguments is a sequence of functions rather than a function applied to multiple arguments. For the non-curried function below, applied to multiple arguments:

$$f(x, y, z) \tag{3.1}$$

When curried this application evaluates to:

$$f(x, y, z) = g(x)(y)(z) \tag{3.2}$$

So, for the invariant function:

$$isa\_invSetElems(isa\_invSetElems, isa\_invVDMNat1) \rightarrow X \tag{3.3}$$

The curried function in Isabelle would be the evaluation of $isa\_invSetElems$ applied to $isa\_invSetElems, appliedto \rightarrow isa\_invVDMNat1$, then the evaluation of all of those functions to X, to clarify:

$$((isa\_invSetElems \rightarrow isa\_invSetElems) \rightarrow isa\_invVDMNat1) \rightarrow X \tag{3.4}$$

And this is how it is translated in the tool.

2.
```
    // Crete apply to the inv_ expr e.g inv_x inv_y
      AIdentifierVarExpIR invExp = new AIdentifierVarExpIR();
      invExp.setName(node.getName());
      invExp.setType(this.methodType);
```

`AIdentifierVarExpIR` nodes are those that hold variable values in a function body, functions are applied to these variable nodes using `AApplyExpIR`. Here the `AIdentifierVarExpIR` node will be the field declaration that the curried invariant functions are applied to.

3.
```
this.targetIP = invExp;
```

So that the `AIdentifierVarExpIR` can be used later when building the function in a separate method, it is stored in a field.

4.
```
    completeExp.setType(new ABoolBasicTypeIR());
```

All invariants return a boolean value, and so the type of the invariant expression body is set to a boolean IR node `ABoolBasicTypeIR`.

5.
```
  //Recursively build curried inv function e.g. (inv_VDMSet (inv_VDMSet
      inv_Nat1)) inv_x

completeExp = buildInvForType(t);
return completeExp;
```

Here, the `AApplyExpIR` that was created earlier and set up in subsequent steps, is assigned to the return of the `buildInvForType` method and returned to the `IsaInvGenTrans` class, this method drives all invariant generation for the entire tool and uses recursion to handle nested collections.

The `buildInvForType` method

This section will go through each stage in the `buildInvForType` recursive method.[25]

1.
```
String typeName = IsaInvNameFinder.findName(seqtNode);
```

The `IsaInvNameFinder`[26] is used to get the correct invariant name based on the current type passed into the method - `seqtNode`. If the type is a modeller defined type i.e. a type that is not present as an STypeIR node, the `IsaInvNameFinder` class' `createNewReturnType` method uses another map present as a public field in the `IsaGen` class, `typeGenHistoryMap` . `typeGenHistoryMap` is populated with mappings of previously declared modeller defined type names, for example `Qualification`, and their declaration, that have been added to the AST.

2.
```
AFuncDeclIR fInv;
if (this.isaFuncDeclIRMap.get("isa_inv"+typeName) != null)
{
   fInv = this.isaFuncDeclIRMap.get("isa_inv"+typeName);
}
```

If the type is a node in the IR AST, its invariant is present in the `isaFuncDeclIRMap`, so its invariant is retrieved.

3.
```
else
{
```

[25]See appendix B, section B.1.5
[26]See appendix B, section B.1.1 .

```
        fInv = IsaGen.funcGenHistoryMap.get("inv_"+typeName).clone();
    }
```

If the type is modeller defined, e.g. `Qualification` then its invariant is present in the `funcGenHistoryMap` which is a mapping of all previously generated invariant functions for modeller defined types, value or state declarations.

4.
```
    if (fInv.getMethodType() == null)
    {
        AMethodTypeIR mt = new AMethodTypeIR();
        mt.setResult(new ABoolBasicTypeIR());
        mt.getParams().add(seqtNode);
        fInv.setMethodType(mt.clone());
    }
```

If the invariant retrieved has no method type already, or no existing invariant was retrieved, then one is provided for it. This had to be done here because of an issue with method types being nullified after some previous computation despite cloning or other measures. The best way to combat this was to spare a few lines to reset it, eliminating the NPE.

5.
```
    AIdentifierVarExpIR curriedInv = new AIdentifierVarExpIR();
      curriedInv.setName(fInv.getName());
      curriedInv.setSourceNode(fInv.getSourceNode());
      curriedInv.setType(fInv.getMethodType().clone());//Must always clone
    AApplyExpIR accum = new AApplyExpIR();
    accum.setRoot(curriedInv);
```

This is where the curried invariant function is set up so that it can be applied to its arguments, or in the case of curried functions, subsequent function evaluations. A new `AApplyExpIR` node is created and its root is set to the recently generated invariant function call, `curriedInv`.

6.
```
    //if this type is not the last in the nested types, then keep
        rescursing until we get to the final nested type
    if ( seqtNode instanceof ASetSetTypeIR && ((ASetSetTypeIR)
        seqtNode).getSetOf() != null )
    {
        accum.getArgs().add(buildInvForType(((ASetSetTypeIR)
            seqtNode).getSetOf().clone()));
    }
    else if (seqtNode instanceof ASeqSeqTypeIR && ((ASeqSeqTypeIR)
        seqtNode).getSeqOf() != null)
    {

        accum.getArgs().add(buildInvForType(((ASeqSeqTypeIR)
            seqtNode).getSeqOf().clone()));
    }
```

If the `setOf` or `seqOf` is a collection, and it is not a collection of nothing,[27] then

[27]The last level of nesting in the nested collection could be an empty collection

the argument of the `AApplyExpIR` is set as a recursive call to the `buildInvForType` method, with the `setOf` or `seqOf` type as its parameter, `buildInvForType` will do all of this again recursively for that nested type.

7.
```
else
{
   accum.getArgs().add(targetIP);
}
return accum;
```

The base case of `buildInvForType`, the final function in the curried function is set to the target identifier variable set in the case method that called `buildInvForType` earlier.

The `buildInvForType` method's recursive operation

The operation of `buildInvForType` is not clear to understand from the code alone. As an example, say the type initially passed in by the case method is `VDMNat1 VDMSet VDMSet`. Let S, S, N represent these types, b represent the `buildInvForType` method, AP represent the `AApplyExpIR` generated at each call and i represent the target identifier variable.

$$b(S\ S\ N\ ) \tag{3.5}$$
$$AP(S,\ ) \tag{3.6}$$
$$AP(S,\ AP(S,\ )\ ) \tag{3.7}$$
$$AP(S,\ AP(S,\ AP(N,\ i)\ )\ ) \tag{3.8}$$
$$return:\ AP(S,\ AP(S,\ AP(N,\ i)\ )\ ) \tag{3.9}$$
$$\tag{3.10}$$

3.4.6 Function Transformations

A new function transformation visitor, `IsaFuncDeclConv` was created and added to `IsaGen` to generate pre and post conditions, add them to the AST and handle implicit functions, for which there should only be translated a pre and post condition.[28] A brief explanation of these conditions is below.

The class has a central case handling what is to be generated for certain `AFuncDeclIR` nodes, the case calls methods based on different conditions.

```
if (!x.getName().contains("inv") &&
      !x.getName().contains("post") && !x.getName().contains("pre"))
{
```

This stops pre conditions of pre conditions of pre conditions etc. from being generated when the visitor traverses the AST, as pre, post and invariants are all `AFuncDeclIR` nodes and so would be executed by this case.

```
if (x.getImplicit()) removeFromAST(x);
```

The above removes implicit functions from the AST as only pre and post conditions should exist for implicit functions. Adding and removing from the AST is done by removing or adding an object from or to the `_decls` child of the enclosing `AModuleDeclIR` node described in 3.3.2. This can be seen below:

[28]This section will refer to appendix B, section B.1.2 where you can find the full class.

```
private void removeFromAST(AFuncDeclIR x) {
   // Insert into AST
     AModuleDeclIR encModule = x.getAncestor(AModuleDeclIR.class);
     if(encModule != null)
     {
         encModule.getDecls().remove(x);
     }

}

 private void addToAST(INode node, INode parent) {
   // Insert into AST
     AModuleDeclIR encModule = parent.getAncestor(AModuleDeclIR.class);
     if(encModule != null)
     {
         encModule.getDecls().add((SDeclIR) node);
     }


}
```

Some velocity was required to assist translation of function declarations.

```
#macro ( transIdentifiers $node )
#foreach($p in $node.FormalParams)
$Isa.trans($p.pattern)##
#end
#end

definition
#if ("$Isa.transTypeParams($node.MethodType.Params)" == "")
   $node.Name :: "$Isa.trans($node.MethodType.Result)"
#else
   $node.Name :: "$Isa.transTypeParams($node.MethodType.Params) \<Rightarrow>
       $Isa.trans($node.MethodType.Result)"
#end
    where
    "$node.Name #transIdentifiers($node) \<equiv> $Isa.trans($node.Body)"
```

The structure of this file was changed slightly[29]. To translate only the result for no parameter functions, as it should be in Isabelle, before this change the parameter was translated as empty parentheses.

Pre/Post condition generation

Transforming pre and post conditions of a function are done more or less in the same way. Method types, function declarations, identifier patterns, formal parameters etc. are all set and generated in different but similar ways that do not warrant stepping through their code as we have done before, please refer to appendix B, section B.1.2, for clarification. As the full method is far too large, and its function similar to that which you have seen in the invariant generation, there are few new issues and headlines to discuss.`IsaInvExpGen` is used again for its `buildInvForType` method to generate invariant checks for parameters of a function that will be present in the pre and post conditions. Method types are set using the parameters and results of existing pre and post conditions or generated based on the

[29]See appendix D, section D.2

functions parameters. Pre and post conditions must always be generated regardless of the function and so this visitor always produces something.

```
/* If no post condition is written, none has been generated then
 there are no parameter types to use as invariant checks and no relevant
     checks provided
 by modeller, so post condition is added but left empty as a reminder to the
     modeller to add one later.*/
 else if (node.getPostCond() == null && generatedPost == null)
 {
   //Copy across all pre written properties into final post condition.
   finalPostCondition = new AFuncDeclIR();
   ANotImplementedExpIR n = new ANotImplementedExpIR();
   n.setTag("TODO");
   finalPostCondition.setBody(n);

     AMethodTypeIR mty = new AMethodTypeIR();
     mty.setResult(new ABoolBasicTypeIR());
   mty.getParams().add(mt.getResult());
     finalPostCondition.setMethodType(mty);
   finalPostCondition.setName("unimplemented_post_"+node.getName());
 }
```

Even if the function has no information with which to generate a pre or post condition, one is added to remind the modeller to do so later.

As with invariants, an existing pre and/or post condition expression is always added to the end of the generated one. The `IsaInvExpGen` class is equipped with a method `genAnd`[30] called by many cases including the `caseAFuncDeclIR` case. This serves two purposes, firstly, to generate `and` expressions between generated expressions and existing ones so that they are all satisfied in the pre/post conditions of functions, or in the case of `ARecordDeclIR` invariants between each record field. Secondly, to string together invariant checking expressions for each type in the parameters of a function. For example, for a function:

```
definition
   f :: "VDMNat
 \<Rightarrow> VDMInt
"
    where
    "f x \<equiv> 0"
```

We need to check in the post condition that the invariant for VDMNat holds for the parameter, that the invariant for VDMInt holds for the result and that any existing post condition expression is included, the tool uses `genAnd` to generate the below recursively:

```
definition
   post_f :: "VDMNat
 \<Rightarrow> VDMInt
 \<Rightarrow> \<bool>"
    where
    "post_f x RESULT \<equiv> ((isa_invVDMNat x \<and> isa_invTrue RESULT)
        \<and> true)"
```

The `IsaInvExpGen` method `genAnd`, achieves this through a very similar mechanism to that seen in `buildInvForType`, hence we will not discuss it in detail - it is enough to know that post condition checks are linked together through recursive generation of nested `\<and>`

[30]See B, section B.1.5

expressions.

3.4.7 State Declaration Translation

The state invariant generation in `IsaInvGenTrans` works identically to `ATypeDeclIR` but must be separated for very minor differences, the code of the two cases is almost exactly symmetrical. The functionality in `IsaInvExpGen` contains no new issues or headlines either. A for loop is used to generate invariants for each field in the declaration, this is because states are represented as records in Isabelle.

3.4.8 Final Notes on Implementation

This section has abstracted away any subtle changes between transformations for each construct because it was meant to focus on purely the headlines and issues encountered during development, needless repetition was avoided at all costs. Noteworthy methodology and difficult obstacles alone were detailed, but this development took a significant amount of time and refinement. Methods in this section began three times their size and through continuous development iterations they have been refined and whittled down using logic and Java's powerful object oriented structure. To name but a few ways: interfaces and casting were used to take two similar code blocks in the same family of node, and merge them into one; visitors were used to do away with clunky control blocks that rendered a method longer than most classes; the `buildInvForType` and `genAnd` methods are used widely throughout the tool and use recursion to achieve correct translation for every construct passed to them; `IsaInvExpGen` alone uses these methods successfully for every declaration in the declaration node family. In all, it is worth looking at the appendices of this paper, the code there is far longer and more complex than can be impressed in this relatively short description and may give an appreciation for how each these classes utilise the functionality of the other.

3.5 Isabelle Translation Recipe

Translation from VDM to Isabelle, at a high level, is relatively straightforward, expressions tend to translate one to one with different symbol representations, as we have seen with `\<any operator>`. A few expressions do not follow this rule, like cardinality, which is handled differently in Isabelle as it is to VDM, for this reason we use the VDMToolkit's definition of cardinality `VDMCard`. This is the same for `len`, to get the length of a sequence and a few more. The VDMToolkit, as I have said before, is the most powerful tool in translation; where tricky hacks around differences between VDM cardinality and Isabelle cardinality might have been time consuming and difficult to write, VDMToolkit gives us a solution of a pre-written, pre-proven cardinality function.

3.5.1 Type Translation Recipe

Translation of types is similar for all types, and we will avoid going into too great a depth about the cases in which it is different. For a type:

```
types

t = nat
```

The type `t` would be translated using the `type_synonym` Isabelle construct into:

```
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMNat"
```

The Isabelle `type_synonym` keyword replaces, in a sense, the types keyword for VDM-SL, however in Isabelle types need to be defined one at a time, rather than in a block of type definitions like in VDM-SL.

A block of type definitions below will demonstrate both the idea of one to one translation and the translation of a large number of VDM types:

```
types
x = char
t = nat
y = set of nat1
A :: b : seq of char
      c : int
  Q = <D> | <C> | <E> | <G>;

...
```

Each type would be translated as:

```
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMNat"

type_synonym x = "\<char>"

type_synonym y = "VDMNat1 VDMSet"

record A =
  a_b :: "char VDMSeq"
  a_c :: int

datatype Q = D | C | E | G
```

Collections are written in the opposite direction in Isabelle as what they collect is applied to their type declaration. Records are translated in almost the same way as they are written in VDM-SL, the `record` keyword denotes a record and its name is written directly after it. Fields are separated by white-space and their naming convention is to use the lower case of the record name, followed by an underscore, followed by the name of the field, though this does not have to be strictly followed. Two colons denote the type of the field. Union types are translated very similarly as well, making the types translation recipe relatively straightforward, the keyword `datatype` proceeds the name of the union, and then all that is done is to remove the $<$ $>$ symbols.

3.5.2 Value Translation Recipe

VDM values are almost always translated into the `abbreviation` construct.

```
values
```

```
x : nat = 1;
```

Should translate to:

```
abbreviation
 x :: VDMNat where
 "x ≡ 1 "
```

As with types, VDM values under the values block are translated one by one with the `abbreviation` pattern for each. `abbreviation` is followed by the name of the value, as with fields two colons denote type; the keyword `where` tells Isabelle that the abbreviation is about to be assigned to a value; inside the speech marks is written the name of the value; the Isabelle translation of equivalence ≡ followed by the actual value of the abbreviation.

```
values
  a: nat1 = 10;
  b: set of nat = {0,...,a};
```

We translate sets and give them value using the normal mathematical set symbol, this is the same with sequences. To initialise a set between a range of values the syntax is the lower bound, followed by an ellipsis, followed by the upper bound. So long as another value has already been defined, as are the rules of functional programming, it can be used in following computation by the interpreter.

```
abbreviation a :: VDMNat where
  "a ≡ 10"
abbreviation b :: "VDMNat VDMSet" where
  "b ≡ {0 .. a}"
```

Where ≡ is what the `\<equiv>` statement is encoded as.

3.5.3 Invariant Translation Recipe

Invariants are translated into Isabelle as functions, they have no language-defined distinction from other functions and are identified solely by their nomenclature, all invariant functions must be proceeded by the word `inv_` followed by the name of the type or value that it belongs to, e.g. `inv_y`. Invariants are slightly more complex translations but no more difficult.

Functions in Isabelle follow the `definition` pattern.

```
types

t = int
inv t == t > 0
```

This type and its invariant would be translated to:

```
type_synonym t = "VDMInt"

definition
   inv_t :: "t ⟶ \<bool>"
    where
    "inv_t t ≡ isa_invTrue t \<and> (t > 0)"
```

Ideally, we also check the invariant of the type that makes up the type declaration, here we use isa_invTrue for demonstration. The definition keyword, just as all other constructs we have seen, proceeds the function name, two colons again denotes type. A functions ”type” is not as straightforward as a single basic type like with a type declaration. Functions have parameters and a return type, they can be empty. A functions ”type” is its method type which is the parameters and return type, so after the double colon we place the functions parameters, the invariant t takes a type t and returns a boolean as all invariants do, an invariant is satisfied and evaluates to true, or is broken and evaluates to false, this is boolean. Multiple parameters for a function are curried, as explained in 1, and as will be clarified soon also. The where keyword, again denotes assigning of a value to the definition. The name of the function, ≡ symbol and function body go inside the speech marks. inv_t t here is an example of a local variable parameter, where inv t in VDM-SL takes type t and evaluates it against the greater than operator, functions can be applied by placing such a variable after the function name, as seen with isa_invTrue t.

3.5.4 Function Translation Recipe

For further translations of functions outside of invariants, we discuss pre and post conditions, as well as implicit function translation. Functions in VDM are translated into Isabelle in the exact same way as with invariants, but with extra functionality. Functions are again, translated one to one, each with the definition pattern.

```
functions
f : nat * nat -> int
f (x,y) == 0
pre true
post true
```

This function, with multiple parameters stored as a tuple, would be written in Isabelle with curried functions, as described before, the * of the tuple would be removed and replaced with further right arrows, as below. Also, pre and post conditions would be translated in the exact same way as invariant functions, but would need extra local parameter variables for the additional parameters, x and y.

```
definition
   f :: "VDMNat
 → VDMNat
 → VDMInt
"
    where
    "f x y \<equiv> 0"


definition
   pre_f :: "VDMNat
 → VDMNat
 → \<bool>"
    where
    "pre_f x y ≡ ((isa_invVDMNat x \<and> isa_invVDMNat y) \<and> true)"


definition
   post_f :: "VDMNat
 → VDMNat
 → VDMInt
 → \<bool>"
   where
   "post_f x y RESULT ≡ ((isa_invVDMNat x \<and> (isa_invVDMNat y \<and>
       isa_invTrue RESULT)) \<and> true)"
```

Pre conditions only check parameters and return boolean, whereas post conditions check that parameters produce particular results and so take as parameters every type in the function signature, and returns boolean. As mentioned, the numerous parameters are curried, where numerous parameters are tuples in VDM-SL, in Isabelle they are separated by → symbols, applied to one another, this is currying.

Implicit functions are those without a function body, they specify parameters and a return type, or just a return type.

```
functions

f (x:int, y:int) r: int
pre y <> 0
post x/y =r
```

As these functions have no implementation to translate, they are defined by their pre and post conditions, in this case only pre and post conditions are translated.

```
definition
pre_f :\: "VDMInt
 → VDMInt
 → \<bool>"
   where
   "pre_f x y ≡ y <> 0"


definition
   post_f :: "VDMInt
 → VDMInt
 → VDMInt
 → \<bool>"
   where
   "post_f x y r \<equiv> (x / y) = r"
```

3.5.5 State Translation Recipe

In Isabelle, state is broken into three separate parts: the state declaration, stored as a record type; the state invariant and the state initialisation.

```
state S of
  x : nat
  y : nat
  init s == s = mk_S(0)
end
```

State is stored as a record, the initialisation of the state is a definition as is its invariant.

```
record S =
  x :: VDMNat
  y :: VDMNat
```

```
definition
  inv_S :: "S  \<bool>"
where
  "inv_S x y  inv_VDMnat1 (x st) \<and> inv_VDMNat1 (y st)"

 definition
  init_S :: "S"
where
  "init_S  (| x = 1, y = 2 |)"
```

For the invariant, we check the type of each field, as we do for all record types. In Isabelle, record fields are accessed like so `(field_name record_name)`. The initialisation of the state is done with another function, the third and final part of state translation. As explained before, functions follow the `definition` pattern. In VDM-SL, fields are initialised individually, either with a predicate, which is easily translated as we have seen in invariant expressions in previous sections, or with a make expression like so `mk_stateName(field1_value, field2_value, fieldN_value)`. In Isabelle make expressions are translated as corresponding fields being assigned a value between what are known informally as ”banana brackets”, ⦇ ⦈. For a make expression like `mk_stateName(field1_value, field2_value, ..., fieldN_value)`, the Isabelle translation would be of this structure ⦇`field1 = 1, field2 = 2, ..., fieldN = X`⦈.

Chapter 4

Results

The following section will describe the outcome of development, it will discuss, among other things: a measure of what work has been done, for example which constructs were translated; metrics of the development, how long development took, how many lines of code were written; limitations of the current state of the tool.

4.1 Work Done

As mentioned, the purpose of development was to create IR transformation functionality, that would facilitate translation from the IR into Isabelle, for as many VDM constructs as possible. The test framework provided a large list of tests for the translation of VDM constructs to Isabelle, the test strategy was that they should all pass, there were tests for each construct, as well as tests for a combination of constructs. By the end of development every one of these tests passed successfully, each construct test and construct combination test matched a manual correct translation.

```
types

t = map int to char
```

Listing 4.1: The VDM-SL test file MapIntChar.vdmsl

This test file tests that the translation of a VDM-SL map, and a combination of int and char constructs, matches a previously manually translated correct translation, seen below in MapIntChar.vdmsl.result. If the translation matches then the tool successfully translates this construct.

Listing 4.2: The MapIntChar.vdmsl.result file specifies that the below is the correct translation and the output of the tool applied to MapIntChar.vdmsl should match it.

```
 --- Expected: ---
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMInt \<rightharpoonup> char"




definition
    inv_t :: "(t) \<Rightarrow> \<bool>"
    where
    "inv_t t \<equiv> isa_invTrue t"

end
 --- Got: ---
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMInt
 \<rightharpoonup> char"



definition
   inv_t :: "(t) \<Rightarrow> \<bool>"
    where
    "inv_t t \<equiv> isa_invTrue t"

end
```

The .vdmsl.result file specifies that not only should the type be translated but it should also have an invariant generated for it. The output, under the word "Got:" matches, and so this test passes, this construct is successfully translated.

Additional to the pre-defined tests, I created some further tests that should pass, testing things like nested collection translation and translation of multiple parameter functions. A partial list of tests and their contents is in the appendices along with their result files, appendix C, section C.1. The collective code is far too long to include each test. The test files covered important aspects of each construct, and each "family" of construct had a large number of tests, totalling just over 50. Function tests tested combination of parameter types, different numbers of parameters, implicit functions, function application expressions and pre and post conditions. Type tests tested each basic type construct, and combinations of them, with modeller defined invariants and without them. Value expressions tested record values, combination of types, initialising those values with expressions or predicate. State tests tested the tool for its ability to generate and translate state initialisation and invariants.

4.1.1 Reflection on Difficulty at Each Stage

This means that every VDM construct was successfully translated, with the exception of a few special cases, discussed in 4.1.3. Basic constructs such as `Int`, `char`, `set`, essentially the type construct, were translated without much hassle, relative to function or invariant translations. Successful translation of basic constructs presented a nice quirk of the development process that I decided to exploit early on, if visitors and their methods are built flexibly, translations almost pass like dominoes when one of these tests pass. Translation of the basic types and their invariants means that any consequent combination of types is translated successfully. For example, successful translation of `Map`, `Int` and `char`, powered by a flexible visitor for generating invariants from any type, means that any combination of them also translate successfully - see C.1.19. This feature means that translation of most type constructs were relatively trivial after one good general transformation visitor was written, the truly difficult part of translation was engineering this flexibility.

This was not true for all types however, and more tricky transformations were required for record types due to the uniqueness of their field structure, however even then, due to the fact that a flexible invariant and type generation visitor was set up, transformation functionality was already majorly established for each field and only required some massaging. Due to the way that invariant generation visitors had been written, different invariants for different fields were generated, the difficult part here was to combine them into one invariant field.

```
types

RecType :: x : char
        y: real
```

Listing 4.3: Rec2FieldsDiffTypes.vdmsl two fields needed to have invariant checks in the invariant but were generated in separate invariant functions.

Listing 4.4: After removing malformed individual invariants from the AST one was generated and combined no additional code had to be written to translate each field's type as this had already been done when translating type constructs.

```
  imports VDMToolkit
begin

record RecType =
      recType_x :: char
      recType_y :: \<real>




definition
  inv_RecType :: "RecType \<Rightarrow> \<bool>"
   where
   "inv_RecType r \<equiv> (isa_invTrue (recType_x r) \<and> isa_invTrue
      (recType_y r))"

end
```

The more difficult translations were for functions - invariant or otherwise. Functions have many more layers of complexity to be concerned with during translation, the tool had to worry about more than just one name, one type and keywords generated by velocity. Now there were special cases for implicit functions, multiple parameter functions, no parameter functions, functions with existing pre and post conditions with their own identifier variables. For implicit functions, the malformed function body needed to be removed from the AST leaving only correctly translated pre and post conditions. Pre conditions carried the issue of having to traverse the AST and pull out only the parameters of a function for their method type. Post conditions required more fiddling by adding the result of their parent function to their parameters. Most of all though, the most difficult thing in the development was generating expressions. Generating expressions was so sophisticated for the fact that they were needed to be generated by almost every construct in the AST, and as such, the methods involved needed to be highly flexible. Figuring out the recursion of methods such as `genAnd` and `buildInvForType` was a difficult task with so many different inputs to consider, and took up the majority of my time with the project. Again though, once this infrastructure had been set up, all invariant check expression generation was successful for functions and invariants regardless of their type, structure or parameters.

Briefly, translation of state declaration followed the same formula as other transformations before it, and all that needed be done was to create a new function that would hold initialisation of state.

To summarise, thoughtful generation of flexible transformations from the start, made it trivial to translate what should have been highly complex constructs later in the development, and this I think is what makes the tool successful. Creating transformations was difficult overall because development was aimed toward making the tool as flexible and extensible as possible. Translating individual constructs would have been trivial but rigid, and not extendible to successful translation of further tests on whole models.

4.1.2 Complexity of This Project

This project had a high level of complexity because of the numerous, and varied, frameworks and environments that drive it. For the Java side alone, there was an AST to work with and efficiently traverse and manipulate. I had to learn the structure of the AST, the families of nodes within it and their interaction. I also had to learn the visitor design pattern and get to grips with the various adaptors, each one with its own subtlety and aberration. Familiarity with a number of frameworks was also essential, the code generation platform, overture, and the ability to write and use Apache Velocity. Before development could begin, it was necessary to be able to write, use and understand two formal specification languages, both Isabelle and VDM-SL. Finally, and perhaps most importantly, the tool required me to learn and be able to perform translation from VDM into HOL and Isabelle so that the tool produced the correct translation.

4.1.3 Limitations of the Tool

Due to time restrictions, a few translations are not sophisticated, the tool simply prints out the translations of the node that it can cope with, and for the most part this is acceptable, however for more intricate Isabelle, the tool might struggle to produce an error free translation. These include the template type, and the `ARecordPatternIR` pattern template, as well as a number more IR node constructs which form a list of roughly 12. The biggest limitation of the tool is that it does not translate VDM-SL operations, thought it is common for the modeller to not translate operations anyway, there would have been more satisfaction in including operation translations also. With that said, the tool goes above and beyond in other ways, it can translate far more than just the POLAR model, as we will see in . Despite this success, the tool still cannot translate a number of constructs, this is not because the tool cannot achieve this, but because they were simply not attempted due to time restrictions. Further work on the project would continue to add support for new constructs.

4.2 Development Process Metrics

The entire development process lasted for 46 days in total, and each day averaged successful translation of two constructs, this is because later in the project development slowed down to debug null pointer exceptions and develop utility methods for more complex function translations. To the merit of the tool, Velocity templates were kept as small as possible, at 2.6, rounded to 3, lines on average. It is worth noting that the code in the appendices and in this project were at three times their size on the first development iteration, but have been refined and cut down for efficiency. Given more time with the project, it would be rewarding to revise the code further, some control flow blocks could be transformed into more complex separate state methods and would therefore become better programs for it. In all, the functionality of the tool is split across only 14 visitor classes, development contributed 2,072 additions, in lines of code, and 171 deletions.

4.3 Application to POLAR and More

In this section, a number of examples will be displayed to show the tool working on individual constructs. Next, the tool will be applied to the POLAR model. It came to be that the tool worked well on the POLAR model, and therefore it is applied further to the FSM3 payment system model and a model of a nuclear plant Alarm system. Also discussed, will be any bugs and limitations of the tool present in translations. This section is thick with code, but is kept out of the appendices as it is important to have it all in one locale so that it can be commented upon.

4.3.1 Examples of Application to Individual Types

Some function translations

```
module A

definitions

functions
f : int * int * int -> int
f (x,y,z) == 0;

values
x = f(1,2,3);

end A
```

```
--- Got: ---
theory A
  imports VDMToolkit
begin


definition
   f :: "VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> VDMInt
"
    where
    "f x y z \<equiv> 0"

abbreviation
 x :: VDMInt
 where
"x \<equiv> f 1 2 3"


definition
   inv_x :: "\<bool>"
    where
    "inv_x \<equiv> isa_invTrue x"


definition
   pre_f :: "VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> \<bool>"
   where
   "pre_f x y z \<equiv> (isa_invTrue x \<and> (isa_invTrue y \<and>
       isa_invTrue z))"


definition
   post_f :: "VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> \<bool>"
   where
   "post_f xyzRESULT \<equiv> (isa_invTrue x \<and> (isa_invTrue y \<and>
       (isa_invTrue z \<and> isa_invTrue RESULT)))"

end
```

```
functions

f (x:int, y:int) r: int
pre y <> 0
post x/y =r
```

```
--- Got: ---
theory DEFAULT
  imports VDMToolkit
begin


definition
   pre_f :: "VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> \<bool>"
   where
   "pre_f x y \<equiv> ((isa_invTrue x \<and> isa_invTrue y) \<and> (y <> 0))"


definition
   post_f :: "VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> VDMInt
 \<Rightarrow> \<bool>"
   where
   "post_f x y r \<equiv> ((isa_invTrue x \<and> (isa_invTrue y \<and>
       isa_invTrue r)) \<and> ((x / y) = r))"
```

Some value translations

```
values
a = 10;
b = 20;
```

```
c = 30;
```

```
 --- Got: ---
theory DEFAULT
  imports VDMToolkit
begin

abbreviation
 x :: VDMNat1
 where
"x \<equiv> 1"

abbreviation
 y :: VDMNat1
 where
"y \<equiv> x"


definition
   inv_x :: "\<bool>"
    where
    "inv_x \<equiv> isa_invVDMNat1 x"


definition
   inv_y :: "\<bool>"
    where
    "inv_y \<equiv> isa_invVDMNat1 y"

end
```

Some type translations

```
types

t = set of int
inv t == t <> {}
```

```
 --- Got: ---
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMInt
 VDMSet
"




definition
   inv_t :: "(t) \<Rightarrow> \<bool>"
    where
    "inv_t t \<equiv> (isa_invTrue t \<and> (t <> {}))"
```

```
types

t = nat1
```

```
 --- Got: ---
theory DEFAULT
  imports VDMToolkit
begin

type_synonym t = "VDMNat1
"



definition
   inv_t :: "(t) \<Rightarrow> \<bool>"
    where
    "inv_t t \<equiv> isa_invTrue t"

end
```

```
types

RecType :: x : char
          y: real
```

```
 --- Got: ---
theory DEFAULT
  imports VDMToolkit
begin

record RecType =
        recType_x :: char
        recType_y :: \<real>




definition
   inv_RecType :: "RecType \<Rightarrow> \<bool>"
    where
    "inv_RecType r \<equiv> (isa_invTrue (recType_x r) \<and> isa_invTrue
        (recType_y r))"

end
```

```
types

t = map int to int
```

```
--- Got: ---
theory DEFAULT
```

```
  imports VDMToolkit
begin

type_synonym t = "VDMInt
 \<rightharpoonup> VDMInt
"



definition
   inv_t :: "(t) \<Rightarrow> \<bool>"
    where
    "inv_t t \<equiv> isa_invTrue t"

end
```

State translations

```
state S of
  x : nat
  init s == s = mk_S(0)
end
```

```
 --- Got: ---
theory DEFAULT
  imports VDMToolkit
begin

record S =
        s_x :: VDMNat



definition
   inv_S :: "S \<Rightarrow> \<bool>"
    where
    "inv_S s \<equiv> isa_invVDMNat (s_x s)"


definition
   init_S :: "S \<Rightarrow> \<bool>"
    where
    "init_S s \<equiv> (s = (| x = 0 |))"

end
```

4.3.2 Comments

As is evident from the above, the tool successfully translates every construct presented to it by the test framework. For a full list of its translations for all of the types mentioned in 4.1, see appendix E, section E.0.1.

4.3.3 Application to FSM3

FSM3 is a specification of low level C code for the CANDO brain chip[Pol], the function of the tool on this model speaks to this projects success.[1] There are a few places in which the tool does not successfully translate, this is only because it has a handful of velocity templates missing, notice, at one point, an expression operator `\<ADomainResByBinaryExp>` is one of them. Places in which a velocity template is missing are noticed by the pattern `$node.left \<AMissingExp>$node.right`. The tools limitations become visible in large models with a large number of constructs like this one.

4.3.4 Application to Alarm.vdmsl

[2]As previously mentioned, this tool is not without its bugs, one such example is `sch<peri>` above, as well as `Plant_(sch, -)`, this is because the tool handles velocity associated with `sch` incorrectly. This is a good opportunity however, to commend the tool on its robustness. The `post_ExpertToPage` post condition is extremely complex and the tool translates the entire thing correctly.

4.3.5 Application to POLAR

Law prevents inclusion of the POLAR model and its translation, however, the tool translates it without additional trouble to what is previously mentioned above, making this project a success.

[1]See appendix E, section E.0.3
[2]See appendix E, section E.0.2

Chapter 5

Conclusion

To conclude, the issues laid out in the introduction and background chapters are eliminated by this tool, the aims and objectives of this dissertation achieved. As explained in the previous section; the development of the tool was difficult due the vast complexity of its environment. Considering this, and considering that the tool works correctly, the project should be considered a success, this conclusion will attempt justify this statement.

5.1 Measuring The Success Of This Project

From the aims and objectives defined in the Introduction chapter, all were achieved. The aim of this dissertation was to extend the current VDM to Isabelle translation tool so that it could translate more components of the POLAR model, this went unchanged and has been achieved and exceeded as more models than just POLAR can be translated successfully, as demonstrated in 4.3. The objectives of this project were augmented as development and research into the field progressed. The objectives before this project were:

- Understand the Java visitor design pattern.
- Understand the existing tool architecture.
- Create Velocity templates and Java visitors for more VDM constructs.
- Apply the translator to the POLAR model.

These objectives were not enough to cover the work required for a project of such complexity, and in reflection I believe that I underestimated the task with these objectives. The objectives were changed to those seen in 1.3.2, it was no where near enough to understand solely the Java visitor design pattern. A thorough understanding of not only this pattern, but also its intricate implementation with cases and various adaptors, was required to create the functionality of the tool. Likewise, understanding the existing tool architecture did not come close to satisfying the depth of knowledge required to safely navigate and manipulate the AST - it is for this reason that it became an objective to understand the AST and the VDM modules which create and alter its structure. Though, as this paper has shown, I had created a number of Velocity templates and modified many more, it was not necessary, and in fact would have been irresponsible, to create too much Velocity because a good template is just that, a template, it should be abstract, short, and minimal. Working with the existing velocity at the start of the project, and making small modifications to it, improved the overall quality of the tool, and for this reason the third objective was removed. As the results revealed that the tool could translate many more models, the objective was added to apply it to more than just POLAR.

As was hopefully demonstrated in 3.2.3, I fully understand the visitor design pattern and its implementation with adaptor classes in Overture/VDM, 4.1 should have provided an argument that this knowledge enabled me to create transformations to the AST. 3.1 shows how I successfully learnt a large amount about the AST's architecture and the VDM modules which create and manipulate it - again, subsequent sections in 4.1 should demonstrate a well learned, clear ability to work with the AST correctly. Application to the POLAR model was a success and the tool even went further to translate Alarm.vdmsl, FSM3.vdmsl and by extension could translate countless more models. Translation is successful and efficient with the exception of some small foibles already covered, caused by a few kinks in the tool.

5.2 Future Development

This tool lays the foundations for more sophisticated prototypes to be developed on top of it. Flexible methods and translation of basic VDM constructs have made the tool extensible, but in some areas, like the way that symbols are reorganised to their correct positions in set and sequence initialisation, the tool uses a "hacky" method to iterate through where they should have been in their correct place to begin with.[1] Though they present no practical obstacle, and the tool works well for the tests that have been run, given more time, these inelegant methods could be transformed or removed entirely after some thought so that they do not present potential problems down the line for larger models. For actual additional functionality however, it would be highly beneficial if this tool could also generate lemmas, the units of auxiliary information used by Isabelle to prove a model. This would be done in an similar way to the way that it has been done for invariant generations during the development of this tool, although they would have to be generated *after* the generation of the translated model. The reason for this, is that the lemmas are based on other constructs in the Isabelle file as well as theories present in the "Main" Isabelle theory directory mentioned previously.

Should lemmas be generated successfully, the tool would become more than a translation engine and code generator. The tool would then, with the assistance of the Isabelle automated theorem assistant, become an automated proof tool that could have vastly beneficial implementations in proving the specification of safety critical systems.

5.3 Impact of This Project

The introduction explains that before the development of this tool, the modeller was required to manually translate an VDM-SL model into Isabelle. This is cumbersome for two reasons, the first is that the modeller is human and makes mistakes, despite the high intelligence true of those working in the field of mathematical formal specification, human error is ever present. Small errors made in translation could result in faults with proof, e.g. proving the wrong thing. This means that the soundness of proof of an Isabelle proven specification could come into question. The second reason is the massive additional time required to translate a model into Isabelle. Each construct must be manually typed and considered, this means that it simply is not feasible to translate large Isabelle models, like the one for POLAR into Isabelle. The important thing to note is that the man hours required to translate a model could not only be spent proving it, which is the difficult part, but if we are to use Isabelle to prove all safety critical systems, as we should to ensure the utmost safety assurance, then a potentially life saving/changing technology could be delayed while such an automatable step is performed. This tool remedies both, for the

[1]This quirk of the tool is a side effect of reverse recursion to generate symbols.

first, automation means that we only have to prove that a translation is sound once, when we create the translation recipe that the tool follows, then any subsequent translations done by the tool are always likely to be correct, practically eliminating the concern of modeller error. The second encumbrance of time is removed entirely thanks to the speed of the computer, automated translation could speed up the development of safety critical systems with the beneficial side effect of leaving the modeller more time to focus on proof.

Bibliography

2019. URL: https://en.wikipedia.org/wiki/Formal_specification#cite_note-a9-hierons-1.

Casper Thule, Hansen. URL: https://pure.au.dk/portal/en/persons/id(306c27de-bac0-409e-9dfc-a6ab1e4db86f).html.

Coiera, Enrico, Farah Magrabi, and Mi Ok Kim. "Problems with health information technology and their effects on care delivery and patient outcomes: a systematic review". In: *Journal of the American Medical Informatics Association* 24.2 (Feb. 2017), pp. 246–250. ISSN: 1067-5027. DOI: 10.1093/jamia/ocw154. eprint: http://oup.prod.sis.lan/jamia/article-pdf/24/2/246/10436950/ocw154.pdf. URL: https://doi.org/10.1093/jamia/ocw154.

Couto, Luis Diogo. 2019. URL: https://github.com/overturetool/overture/wiki/core-modules.

— *overturetool/overture*. 2019. URL: https://github.com/overturetool/overture/wiki/AST-Guide.

Dholakia, Sham et al. "Preserving and Perfusing the Allograft Pancreas: Past, Present, and Future". In: *Transplantation Reviews* 32 (Mar. 2018). DOI: 10.1016/j.trre.2018.02.001.

Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

Gaudel, M. -. "Formal specification techniques". In: *Proceedings of 16th International Conference on Software Engineering.* May 1994, pp. 223–227. DOI: 10.1109/ICSE.1994.296781.

Git, Github Open Source. *Isagen.* 2015-Present. URL: https://github.com/overturetool/overture/tree/cth/isagen.

Giwa, Sebastian et al. "The promise of organ and tissue preservation to transform medicine". In: *Nature Biotechnology* 35 (June 2017), 530 EP -. URL: https://doi.org/10.1038/nbt.3889.

Hierons, R. M. et al. *Using Formal Specifications to Support Testing.*

Ishikawa, Fuyuki and Peter Gorm Larsen. "GRACE TECHNICAL REPORTS". In: *13th Overture Proceedings.* Grace Center, 2015. URL: http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf.

Lamsweerde, Axel van. "Formal Specification: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering.* ICSE '00. Limerick, Ireland: ACM, 2000, pp. 147–159. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336546. URL: http://doi.acm.org/10.1145/336512.336546.

Papas, KK et al. "Pancreas oxygenation is limited during preservation with the two-layer method". In: *Transplantation proceedings* 37.8 (Oct. 2005), pp. 3501–3504. ISSN: 0041-1345. DOI: 10.1016/j.transproceed.2005.09.085. URL: https://doi.org/10.1016/j.transproceed.2005.09.085.

Pollitt, Allastair. "Verifying the CANDO Project Optrode Command Interface in eCv". PhD thesis. School of Computing Science, University of Newcastle, U. K.

Suszynski, Thomas M. et al. "Persufflation (gaseous oxygen perfusion) as a method of heart preservation". In: *Journal of Cardiothoracic Surgery* 8.1 (Apr. 2013), p. 105. ISSN: 1749-8090. DOI: `10.1186/1749-8090-8-105`. URL: `https://doi.org/10.1186/1749-8090-8-105`.

Tran-Jørgensen, Peter. *overturetool/overture*. 2015. URL: `https://github.com/overturetool/overture/wiki/CGP-Tutorial`.

Walden, David D et al. *Systems engineering handbook*. 3rd ed. INCOSE, 2007, p. 19.