VERIFIED CODE GENERATION FOR A VISUAL MODELING TOOL

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EBRU ÇELEBİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2024

Approval of the thesis:

VERIFIED CODE GENERATION FOR A VISUAL MODELING TOOL

submitted by **EBRU ÇELEBİ** in partial fulfillment of the requirements for the degree of **Master of Science** i**n Computer Engineering, Middle East Technical University** by,

Prof. Dr. Naci Emre Altun
Dean, **Graduate School of Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of the Department, **Computer Engineering** _____

Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering, METU _____

Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU _____

Doç. Dr. Kayhan İmre
Computer Engineering, Hacettepe University _____

Date: 23.08.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name Last name : Ebru Çelebi

Signature :

ABSTRACT

VERIFIED CODE GENERATION FOR A VISUAL MODELING TOOL

Çelebi, Ebru
Master of Science, Computer Engineering
Supervisor : Prof. Dr. Halit Oğuztüzün

August 2024, 83 pages

This thesis introduces a verified code generator for the IMODE tool, which enables users to model a sequential computation visually and to generate executable code from the model that will be part of a safety critical software system. The generated code must carry out exactly the computation modeled without any side effects and must be compliant to the applicable safety critical software standards. The code generator goes through the following stages: (i) It takes its model input from the IMODE XML save files, (ii) checks the input model for well-formedness, (iii) annotates it with logical expressions that capture the input-output relationships effected by the model elements, (iv) generates code in a subset of C, (v) verifies the generated code using Hoare logic, adopting the pre- and post-conditions derived from the annotated model, and, finally, (vi) outputs the verified code. To ensure the correctness of the code generator itself, the Agda programming language, a functional programming language with dependent types, has been used for implementation. Using the Agda's integrated theorem prover, the code generator is proved adhering to the correct-by-construction approach.

Keywords: Model-Based Design Tool, Verified Code Generator, Correct by Construction, Hoare Logic

ÖZ

BİR GÖRSEL MODELLEME ARACI İÇİN DOĞRULANMIŞ KOD ÜRETİMİ

Çelebi, Ebru
Yüksek Lisans, Bilgisayar Mühendisliği
Tez Yöneticisi: Prof. Dr. Halit Oğuztüzün

Ağustos 2024, 83 sayfa

Bu tez, kullanıcıların ardışık bir hesaplamayı görsel olarak modellemelerini ve emniyet kritik yazılım sisteminin parçası olmak üzere modelden kod üretmelerini sağlayan IMODE aracı için doğrulanmış bir kod üreticisi sunmaktadır. Üretilen kod, herhangi bir yan etki olmadan modeldeki hesaplamaları olduğu gibi gerçekleştirmeli ve uygulanacak emniyet kritik standartlarına uygun olmalıdır. Kod üreticisi şu aşamalardan geçer: (i) Model girdisini IMODE XML kayıt dosyaları ile alır; (ii) Girdi modelin bütünlük ve yapısının düzgünlüğünü kontrol eder; (iii) Modeli, model öğeleri tarafından etkilenen giriş-çıkış ilişkilerini yakalayan mantıksal ifadelerle açıklar; (iv) C dilinin alt kümesinde kod üretir; (v) Üretilen kodu, modelin mantıksal ifadeleriyle oluşturulan ön ve son koşullarını kullanarak Hoare mantığı ile doğrular ve son olarak (vi) üretilen kodu çıkarır. Kod üreticisinin kendi doğruluğunu sağlamak için, Agda programlama dili, bağımlı tiplere sahip bir fonksiyonel programlama dili, kullanıldı. Agda'nın bütünleşik teorem ispatlayıcısı kullanılarak kod üreticisi yapısı ile doğru yaklaşımına uygun olarak kanıtlanmıştır.

Anahtar Kelimeler: Model Tabanlı Yazılım Tasarım Aracı, Doğrulanmış Kod Üretimi, Yapısı ile Doğru, Hoare Mantığı

I dedicate this work to my family.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to Prof. Dr. Halit Oğuztüzün for their guidance, advice, criticism, encouragements, patience, and insight throughout the research.

I am also thankful to İbrahim Ardıç and Ümit Aşıkkarip for their support at workload and understanding to use leaves which is also accepted by TUSAŞ. Further thanks to İbrahim Ardıç and Ümit Aşıkkarip for letting me use IMODE and its save files for the purpose of this thesis.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vi

ACKNOWLEDGMENTS viii

TABLE OF CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF ABBREVIATIONS xvi

1 INTRODUCTION 1

1.1 Dijkstra's Weakest Precondition and Hoare Logic 3

1.2 Model-Driven Engineering 3

2 BACKGROUND 5

2.1 IMODE 5

2.2 DO-178 and DO-330 7

2.2.1 DO-178 7

2.2.2 DO-330 8

2.2.3 Stage of Involvement 8

2.3 Qt Node Editor 9

2.4 Hoare Logic 10

2.5 Agda 14

2.5.1 Dependent Types 14

2.5.2 Termination Check and Safety Options 15

2.6 A Constructive Formalization of Hoare Logic within the Interactive Theorem Prover Agda 15
3 VERIFIED CODE GENERATOR 17
3.1 Requirements 17
3.1.1 Program Input 17
3.1.2 Checker 21
3.1.3 Generated Code / Program Output 25
3.1.4 Verification of the Generated Code 29
3.1.5 Quality Requirements 33
3.2 Design and Implementation 34
3.2.1 XML Reader 35
3.2.2 Checker 37
3.2.3 Code Generator 41
3.3 Verifier 44
4 APPLICATION EXAMPLE 51
4.1 Vertical Throw 51
4.1.1 Reader 52
4.1.2 Checker 54
4.1.3 Code Generation 55
4.1.4 Verification 58
5 CONCLUSION 67
5.1 Evaluation of the Work Done 67
5.1.1 Agda as a Programming Language 67
5.2 Future Work 68

REFERENCES 71

6 APPENDICES 75

A. Vertical Throw Save File 75

B. Last Unweakened Postcondition for VerticalThrow 78

C. First Three Step of Hoare Logic for Vertical Throw 83

LIST OF TABLES

TABLES

Table 3.1 IO Requirements for Model Element Types 22
Table 3.2 Model Element IO Types 24
Table 3.3 Model Element Generated Code Structure 28
Table 3.4 Model Element to Annotation 31
Table 3.5 IMODE File Types 35
Table 3.6 Model Element Categories 36

LIST OF FIGURES

FIGURES

Figure 1.1 Example View from IMODE (courtesy of TUSAŞ).....2
Figure 2.1 IMODE Graphical Design Example (courtesy of TUSAŞ) 6
Figure 2.2 IMODE Logical Design Example (courtesy of TUSAŞ).....6
Figure 2.3 Likely Locations for SOIs during Software Development [11] 9
Figure 2.4 An example graph from Qt Node Editor [12]. 10
Figure 2.5 Grammar Snippet [17] 16
Figure 2.6 Hoare Logic Type Declaration in Agda [17].....16
Figure 3.1 Project File Format 18
Figure 3.2 Model File Format 18
Figure 3.3 Model Element File Format 19
Figure 3.4 Connection Model Element File Format.....20
Figure 3.5 Input and Output Model Element File Format 20
Figure 3.6 IO Instance Model Element Additional Format 20
Figure 3.7 Textual Model Element Additional Format 21
Figure 3.8 Numeric Cast Model Element Additional Format.....21
Figure 3.9 Previous Model Element Additional Format.....21
Figure 3.10 Input Structure Template.....25
Figure 3.11 Output Structure Template 26
Figure 3.12 Main Function Declaration Template 26
Figure 3.13 Main Function Template 26
Figure 3.14 Model Element Declaration Template 27
Figure 3.15 Annotation Grammar 30
Figure 3.16 If Model Element in Expression Replacing 32
Figure 3.17 Unweakened Annotation 32
Figure 3.18 Weakened Annotation.....33
Figure 3.19 Flowchart of the Verified Code Generator.....34
Figure 3.20 XML Data Types 37
Figure 3.21 Example Verification for a Requirement 38

Figure 3.22 An Example Model from IMODE Design 39
Figure 3.23 DAG for Figure 3.22 39
Figure 3.24 An Example Model with Previous and Cycle 40
Figure 3.25 DAG 1 for Figure 3.24 40
Figure 3.26 DAG 2 for Figure 3.24 40
Figure 3.27 ModelDAG Type Definition in Agda 41
Figure 3.28 StatementType Definition in Agda 42
Figure 3.29 Example Model Containing If 43
Figure 3.30 Example Generated Main Function 43
Figure 3.31 Annotation from Model 45
Figure 3.32 Hoare Logic for Addition Step 45
Figure 3.33 Hoare Logic for Subtraction Step 45
Figure 3.34 Hoare Logic for Strictly Greater Than Step 46
Figure 3.35 Hoare Logic for If Step 47
Figure 3.36 Hoare Logic for Output Step 48
Figure 3.37 Weakened Postcondition from Generated Code 49
Figure 4.1 Vertical Throw Model in IMODE 51
Figure 4.2 Graph of YPos from IMODE 52
Figure 4.3 Read Model in Agda 53
Figure 4.4 DAGs for Vertical Throw Model 54
Figure 4.5 Header File Definitions and Declarations for Vertical Throw 55
Figure 4.6 Local Variable Definitions in Main Function 56
Figure 4.7 Generated Code from First DAG 57
Figure 4.8 Generated Code from Second and Third DAG 58
Figure 4.9 Annotation of YPos from Model 59
Figure 4.10 Annotation for Velocity from the Model 60
Figure 4.11 Annotation for Previous Memory Variables 61
Figure 4.12 Annotation for Strictly Less Than from Model 62
Figure 4.13 Precondition Consisting of Input and isInitialCycle Variable 63
Figure 4.14 Generated Code for VerticalThrow Model 64

Figure 4.15 Generated Code for VerticalThrow Model (cont.) 65

LIST OF ABBREVIATIONS

ABBREVIATIONS

IMODE: Integrated Model Design and Development Environment

DAL: Design Assurance Level

TQL: Tool Qualification Level

DAG: Directed Acyclic Graph

CHAPTER 1

INTRODUCTION

Verification is a part of high integrity software development processes. In industries like aviation, there are standards that need to be applied to develop software for airborne systems to assure safety. DO-178 is one of them. Because of such standards' demand, software verification processes have much more workload because even one small error can cause life threatening situations. In my experience at a major aviation company, verification of a software are done by covering all possible test cases. This creates intense test sessions after each modification of the software and requires overtime work to meet the deadlines.

To be able reduce this workload on verification, helper tools of various categories are used. One of these categories is code generators. These code generators convert software logic/algorithm designs to verified code so that the main software project does not need to spend time on verification for that algorithm. There are also standards to develop such a tool to support a DO-178 project, which is called DO-330. Like DO-178, DO-330 also requires a detailed verification process for the tool's development, but at least doing it for the tool once relieves the other software projects.

Code generators are used extensively in TUSAŞ. The most well-known one is Ansys SCADE Suite [1]. To be able to use the generated code from SCADE Suite, purchase of the certification package is required. Thus, some of the project's budget always goes to the code generator tools. In 2019, TUSAŞ started a project called IMODE (Integrated Model Design and Development Environment) to reduce the dependence on foreign sources. IMODE is model-based software development tool that generates code from graphic and logic design. I am also working on IMODE, mostly on logic design and code generation part. IMODE is currently in the process of obtaining DO-

330 certification. Thus, verification should show that generated code performs as intended by the designer without any side effects and should comply with the DO-178C standard.

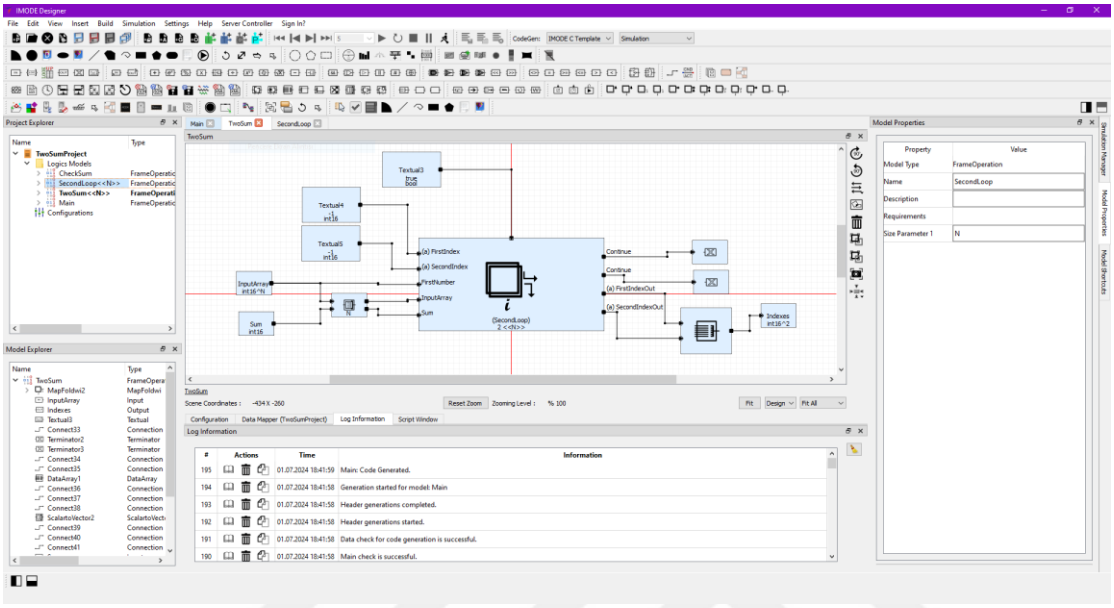


Figure 1.1 Example View from IMODE (courtesy of TUSAŞ)

IMODE logic design has a source to sink data flow method. Data originating from source model elements are operated on with intermediate model elements and finally set to sink model elements. The generated code is in C with restrictions so that it is complied with DO-178.

My subject and motivation for thesis was to verify generated code from the model-based design without writing and testing all possible cases. Instead, the correctness of the program can be proved by itself, proof-by-construction. A programming language that makes the proof-by-construction approach possible is Agda. With help of dependent types and the theorem prover, program correctness can be proved by type checking before compiling.

1.1 Dijkstra's Weakest Precondition and Hoare Logic

Dijkstra's weakest precondition formalism can be used for program verification [2]. To prove a program using weakest precondition, first precondition and postcondition of the program should be defined formally which are logical expressions of the initial and final values of the variables in the program respectively.

Precondition and post condition can also be called predicates over states. Then there is predicate transformer which is a function to take postconditions and compute weakest precondition as outputs [3]. Each statement in the program will have a predicate transformer to calculate the weakest precondition for the statement and post condition.

In model, there is a data flow from source model elements to sink model elements. We can extract source and sink model elements logical expression and annotation from model and show the equivalence of these annotation with precondition and postcondition of the program.

To show that equivalence, we can extract the annotations for sink model elements and show that the generated code also produces the logically equivalent annotations for the variables that correspond to sink model elements. For this approach, Hoare logic is found suitable. Using Hoare logic rules on the generated code, we can extract the last annotations for the variables and compare them with the design intentions.

1.2 Model-Driven Engineering

The code generation process accomplished by our translator is known as model-to-text (M2T) transformation in the MDE (Model-Driven Engineering) jargon. Some recent references that provide context for our work are mentioned in this section.

Domain-specific languages (DSL) or domain-specific modeling (DSM) are common to use for code generators. Such examples are shown in a case study called "Code generation by model transformation" to provide a better modularity in the generated

code [4]. In the case study, techniques are applied on a code generator for a realistic system.

Another mention can be a book by Steven Kelly and Juha-Pekka Tolvanen called “Domain-Specific Modeling: Enabling Full Code Generation” [5]. This book introduces DSM and provides examples from various fields to improve software development with code generation from a DSM.

For the verification of model transformation in general, elements of a framework for variety of transformation and verification method for each of them are introduced in the paper “A framework for verification of model transformations” [6]. This paper covers a wide range of different verification methods for different model transformation languages including code generators.

In a small paper by Markus Völter and Bernd Kolb, best practices of model-to-text transformations, especially for code generations, are discussed [7].

In the remainder of the thesis, background information combined with previous works with similar concepts are introduced. Then verified code generator project is described in requirements, design, implementation, and verifier details. Simple examples are given through CHAPTER 3 while explaining the implementation. A complete model code generation and verification example is shown in the CHAPTER 4. Lastly in CHAPTER 5, there are reflections on the project and future work that can be done on the project.

CHAPTER 2

BACKGROUND

This chapter contains detailed information about tools, standards and concepts that are mentioned throughout the thesis. Also work done previously about Hoare logic in Agda and a visually similar model-based tool are discussed.

2.1 IMODE

Integrated Model Design and Development Environment, in short IMODE, is a model-based software design tool currently being developed by TUSAŞ Model-based Software Development Team which I am a member of. IMODE currently provides graphical design, logic design, architectural design, automated code generation in C, simulation and debugging, requirement management, verification and interoperability with other development tools and platforms. IMODE is originated from master's thesis of İbrahim Ardıç in 2015 [8].

There are standards for software in airborne systems and equipment to follow which are controlled and certified by the authorities such as FAA and EASA. These standards make sure the software is safe and secure for the airborne systems through giving a checklist for all processes beginning from planning to verifying. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, is the one used on TUSAŞ. DO-178C document is published and distributed by RTCA organization.

There are tools to help reduce these coding and verification processes which also need to follow a standard called DO-330 Software Tool Qualification Considerations. When a tool is certified with DO-330, its outputs can be used to skip some checklists in DO-178C or other applicable standards. Most popular one for the

tools used is SCADE Suite which generates code from the logic design. If the design is verified, the code that SCADE generates does not need to be tested. Because SCADE Suite tool guarantees that generated code is doing the exact calculations the user designed on the tool. Also, DO-178C authorities accept this code if the design is verified because SCADE are certified with DO-330.

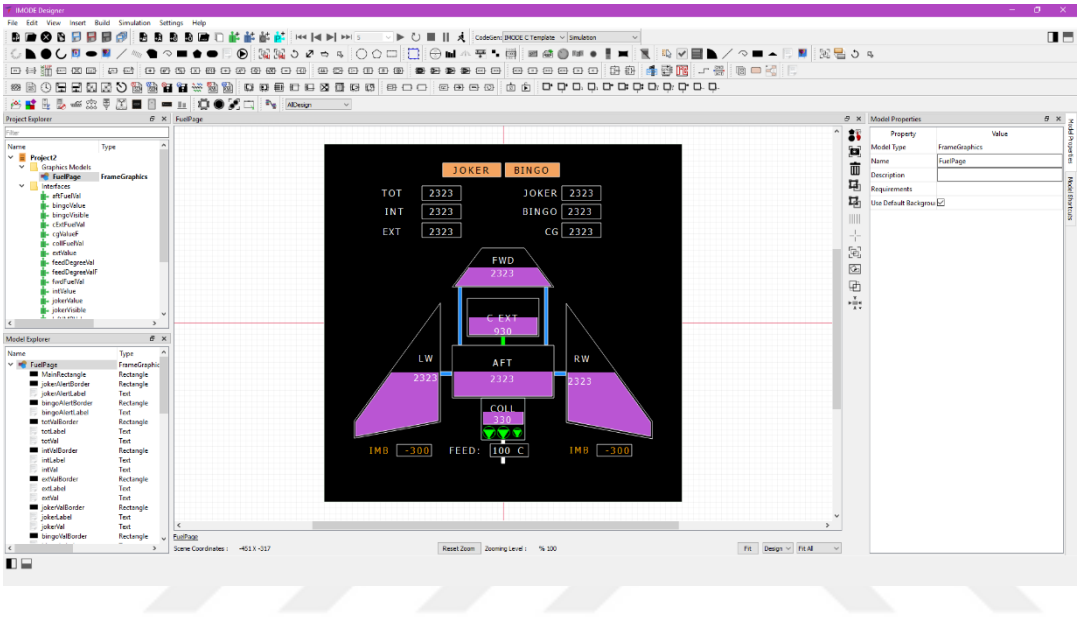


Figure 2.1 IMODE Graphical Design Example (courtesy of TUSAŞ)

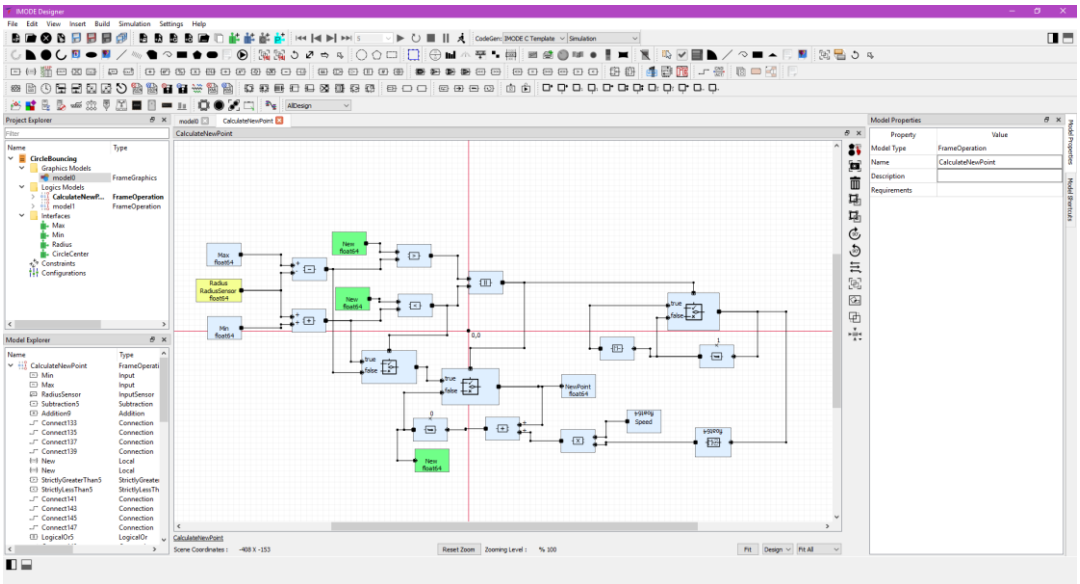


Figure 2.2 IMODE Logical Design Example (courtesy of TUSAŞ)

IMODE aims to become one of the tools used in Türkiye. The tool is already in working state but lacks the DO-330 certification. IMODE is currently finished planning documents for the certification and awaiting the authority control.

In my thesis I worked on the logic part of the IMODE, verifying that generated C code operates the same as logic design on IMODE using Agda and Hoare Logic.

2.2 DO-178 and DO-330

For safety critical software systems, there are standards/guidelines/considerations to comply with. The ones that are most relevant to this work are DO-178 and DO-330.

Additional documents can be found to support DO-178 or DO-330 for other methods for software development such as object-oriented, model-based, or formal methods.

2.2.1 DO-178

The first one is DO-178, Software Considerations in Airborne Systems and Equipment Certification. DO-178 is a guideline, standard, for safety critical avionics software. It is published by RTCA in 1992 as DO-178B. After the updates in 2012, it is replaced with DO-178C.

There are software levels that are defined in the document as DAL, Design Assurance Level [9]. Levels are A to E, A being the most critical. Effects of software failure determine the software level. For example, Software is level A if failure has a possibility to cause deaths. Software is level D if failure can cause minor inconvenience or increase crew workload. In each increasing level, more objectives are required in the document. Also, review, test or document objective can also require independence. For example, review of a design should not be done by the same person who did the design.

Automated tools can be used for the processes of software development to reduce the workload. However, each tool should be specified and accepted by the standard. And for such a tool a DO-330 certification is needed.

2.2.2 DO-330

The second one is DO-330, Software Tool Qualification Considerations. DO-330 is a guideline, standard, for tools that can be used in safety critical processes. DO-330 is not limited to aviation. It can be used in other domains, for example space or automotive. It published by a joint committee of RTCA and EUROCAE as one of four technology supplements with the updated version as known DO-178C. [10]

Like DAL in DO-178C, there are tool qualification levels (TQL) for DO-330. TQLs are 1 to 5, 5 being the most critical. TQL level of a tool is determined by what it does and what possible DO-178C software it can be used in. For example, if the output of the tools will be a part of an airborne system and the airborne system is a level A, then TQL for the tool should be 1. Code generators are in this category. If the tool is used for automating the verification processes or the use of the tool might fail to detect an error in the software, then the tool can be TQL 4 or 5 depending on the DAL. Automated test tools are in this category.

Tools that have DO-330 certification with a suitable TQL for DAL are accepted by the DO-178C and similar standards.

2.2.3 Stage of Involvement

In the development process, there are stages where authority is involved to approve the current situation of the qualification processes. These stages are called Stage of Involvement, SOI in short. Processes and SOIs are shown in Figure 2.3 [11].

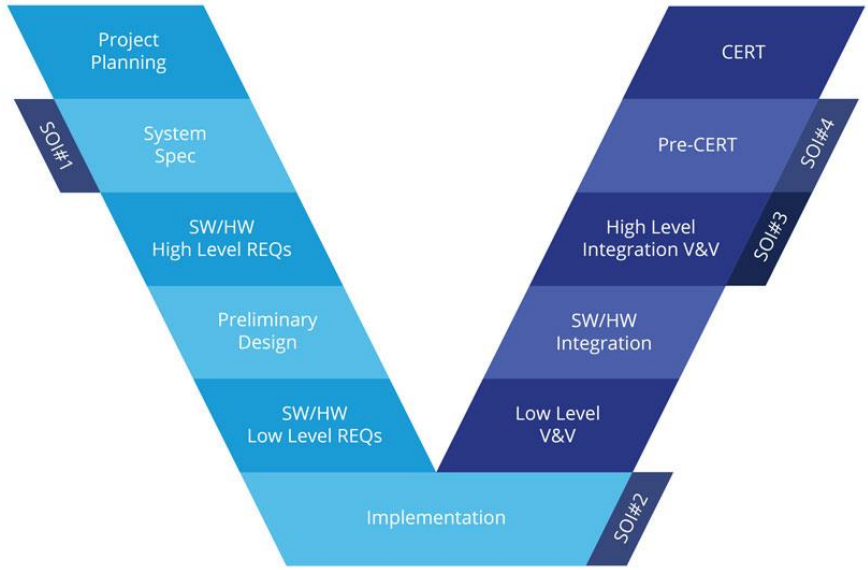


Figure 2.3 Likely Locations for SOIs during Software Development [11]

2.3 Qt Node Editor

Before deciding and getting permission to use IMODE's design files, a similar model-based project named Node Editor on GitHub was considered as a possible replacement to use and add missing operations to [12]. Node Editor is implemented using Qt Framework and used by other flow graph Qt applications according to the information on the GitHub repository.

Node Editor has similar boxes for operations and input outputs connected by lines which the user can design an algorithm with. Instead of generating code, it performs the calculation directly on the model.

I thought using Node Editor models to generate and verify the code but eventually decided to use IMODE's design file directly.

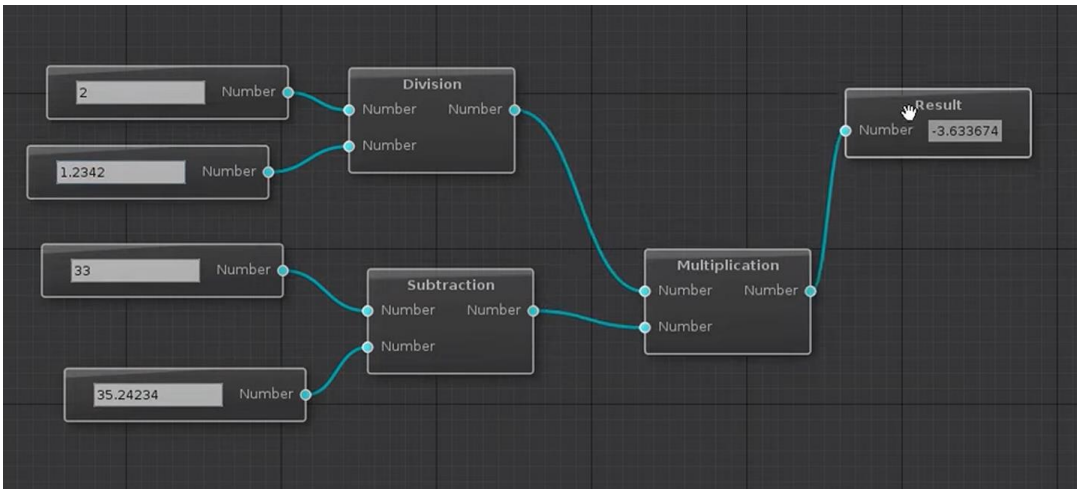


Figure 2.4 An example graph from Qt Node Editor [12].

2.4 Hoare Logic

Hoare logic, which was proposed by Tony Hoare in 1969, is a formal system consisting of inference rules for proving the partial correctness of imperative programs. [13] To check if a program carries out its intended operations, assertions about the values of its variables that will take after the execution of the program can be made. These assertions will be general and not bound the variables to some value, but they will rather describe the properties of the values and relationships between them.

Correctness of a program will depend on the values of the variables before its execution. Assertion for the properties of the variables before program execution is called precondition (P). Then there is the (part of) program to be executed (Q). Then the result of execution depending on the precondition is postcondition (R). These three constituents are combined as a Hoare triplet and shown as:

$$\{P\}\ Q\ \{R\}$$

The P and R assertions represented in predicate logic meaning that if P is true before Q execution, then R will be true after Q terminates.

For an assignment statement "$x :=: e$", here are the axioms and rules of Hoare logic [13]:

D0 Axiom of Assignment:

$$\vdash \{P[\, e \,/\, x \,]\}\, x :=: e\, \{P\}$$

D1 Rules of Consequence:

$$\text{If} \vdash \{P\}\, Q\, \{R\} \text{ and} \vdash R \Rightarrow S \text{ then} \vdash \{P\}\, Q\, \{S\}$$

$$\text{If} \vdash \{P\}\, Q\, \{R\} \text{ and} \vdash S \Rightarrow P \text{ then} \vdash \{S\}\, Q\, \{R\}$$

If R is true after Q execution, then all assertion implied by R is also true after Q execution. Similarly, if precondition P of a program Q ensures R being true, then all assertion implying P preconditioning Q is also ensures R being true.

D2 Rule of Composition:

$$\text{If} \vdash \{P\}\, Q_1\, \{R_1\} \text{ and} \vdash \{R_1\}\, Q_2\, \{R\} \text{ then} \vdash \{P\}\, Q_1;\, Q_2\, \{R\}$$

If the postcondition of Q_1 is identical to precondition of Q_2, then Q_1 and Q_2 can be chained, and Hoare triplets can be combined as shown.

D3 Rule of Iteration:

$$\text{If} \vdash \{P \wedge B\}\, Q\, \{P\} \text{ then} \vdash \{P\}\, \textbf{\textit{while}}\ B\ \textbf{\textit{do}}\ Q\, \{\neg B \wedge P\}$$

If Q is put into a loop, it needs to ensure that precondition P is true in postcondition so that loop can continue running. If while statement with condition B and statement Q terminates, then postcondition needs to include negation B which causes loop to break.

A pair of example programs to prove [14]:

Example 1:

{?}

$z = x + 2;$

$y = z + z;$

$x = z + y;$

$\{x > z \ \wedge y :=: 3\}$

Backtrace from the desired postcondition to check if precondition can be true or not.

Logical expressions can be simplified:

$\{2x = -1\}$

$\{(x + 2) + (x + 2) :=: 3\}$

$z = x + 2;$

$\{z + z :=: 3\}$

$\{true \wedge z + z :=: 3\}$

$\{z + (z + z) > z \ \wedge z + z :=: 3\}$

$y = z + z;$

$\{z + y > z \ \wedge y :=: 3\}$

$x = z + y;$

$\{x > z \ \wedge y :=: 3\}$

Found precondition means that if the variables are integers, then there is no solution for the precondition therefore program fails to be verified.

Example 2:

$\{?\}$

$y = x;$

$y = x + x + y;$

$\{y :=: 3x\}$

Backtrace from the desired postcondition to check if precondition can be true or not:

$\{true\}$

$\{x + x + x :=: 3x\}$

$y = x;$

$\{x + x + y :=: 3x\}$

$y = x + x + y;$

$\{y :=: 3x\}$

Precondition being true means that for all values this program is proved.

Example 3:

$\{?\}$

$if(x - y < 0)$

{

$z = x;$

}

$else$

{

$z = y;$

}

$\{z <= y \wedge z <= x\}$

Backtrace from the desired postcondition for if and else parts:

$\{?\}$

$if(x - y < 0)$

{

$\{x <= y\}$

$\{x <= y \wedge x <= x\}$

$z = x;$

$\{z <= y \wedge z <= x\}$

}

$else$

{

$\{y <= x\}$

$\{y <= y \wedge y <= x\}$

$z = y;$

$\{z <= y \wedge z <= x\}$

}

$\{z <= y \wedge z <= x\}$

Now we need to combine two parts. We need to find precondition ($\{?\}$) P that satisfies:

$\{x - y < 0 \wedge P\} => \{x <= y\}$ and $\{\sim(x - y < 0) \wedge P\} => \{y <= x\}$.

$x - y < 0$ already implies $x <= y$ and $\sim(x - y < 0)$ already implies $y <= x$. Thus, anything works for P including true.

2.5 Agda

Agda is a dependently typed functional programming language which is originally developed by Ulf Norell in his PhD thesis. [15] Because of strong typing and dependent types, Agda also has a proof assistant like Coq. [16] However, unlike Coq, proofs are typed in a functional programming style.

2.5.1 Dependent Types

When a type is indexed by another type thus creating a family of types, it is called dependent type. And the type theory that uses dependent types are called dependent type theory. For example, in Agda standard library vector type definition uses n typed Nat (natural number type) as an index to have fixed length vectors as type, Vec n. Vec 1 or Vec 2 become different types.

Using dependent types expresses properties of the program more clearly and if errors are made, they are fixed much earlier. For example, if we want to define a function that concatenates two vectors, its type would be like $\forall$ {n m} → Vec n → Vec m → Vec (n + m). Type system can check if the definition of the function produces n + m length vector, thus providing a foundation for proving its correctness.

2.5.2 Termination Check and Safety Options

Before compiling the code, there is always a check if there is an inconsistency with types or possible erroneous situations with the program logic. One of that is termination checking. Not all recursions are allowed in Agda. Recursive calls should always decrease in the value of some parameter or types, no input can lead to an infinite loop. For example, while traversing a list, another call for the same function should be made with a smaller part of the list so that it will reach to the end of the list after some time. Agda check for these situations during type check unless it is manually disabled. Termination check can be disabled for all or some portions of the code. However, I made sure it is enabled for the purpose of the project.

In addition to termination checks there are more inconsistency checks in Agda. For example, forcing Agda to accept some unfinished proof is possible. By enabling safe Agda with the "--safe" option, I made sure that Agda gives any inconsistency it can find as an error during type check.

2.6 A Constructive Formalization of Hoare Logic within the Interactive Theorem Prover Agda

In his master's thesis, Brooks creates a constructive formalization of Hoare Logic in Agda language. [17] Because a simple imperative language needs to be constructed and formalized to apply Hoare Logic on, a context-free grammar in Backus-Naur form of an assertion language is created. Assertion language contains operands available in most imperative languages. A snippet of language is shown in Figure 2.5.

```
<Exp>       ::= <Exp> <Op2> <Exp> | <Op1> <Exp> | <terminal>
<Op2>       ::= &&∘ | ||∘ | ==∘ | _∘ | ... | +∘ | -∘ | ... | %∘
<Op1>       ::= ++∘ | −∘ | :∘ | -∘
<terminal>  ::= const <num> | var <id> | true | false
<num>       ::= 1 | 2 | 3 | ...
<id>        ::= x | y | z | ...
```

Figure 2.5 Grammar Snippet [17]

Using this grammar an Assertion type is created in Agda where this Assertion type is used in pre and postconditions of Hoare Logic. Type declaration of the Hoare Logic in the Agda is shown in Figure 2.6.

```
«_»_«_» : Assertion -> C -> Assertion -> Set
```

Figure 2.6 Hoare Logic Type Declaration in Agda [17]

Brooks implements a SWAP program and formalizes its correctness using Agda. I was inspired by usage of Hoare Logic and imperative language in Agda.

CHAPTER 3

VERIFIED CODE GENERATOR

In this chapter, each process of verified code generator is explained. Starting with requirements explaining what program should do, what inputs and outputs it should have. Depending on the requirements, the program is designed and implemented to do the reading, checking, generating code, and verifying the generated code. Illustrative examples throughout the chapter are given. An extensive example can be found in CHAPTER 4.

3.1 Requirements

In this section following requirements are defined:

- Program input requirements, how it shall be formatted and what it shall contain.
- Model check requirements which are about making sure that design is complete and ready to generate code without any problem.
- Code generation requirements describing how the code shall be generated from each model element and what shall be the program output.
- Verification requirements describing how to show the generated code and model are compatible with each other.

3.1.1 Program Input

- Program shall take input IMODE save files from 6701 version which are in XML format: (File descriptions are shown in Table 3.5)
    - One Project File (prjx)

- One or more model files (mdlx)
- One Types File (typx)
- One Interfaces File (intrfx)
- One Constants File (constx)
- A model name to generate code of as a string.

- Project file (prjx) XML format shall contain minimum of following elements and attributes:

```
<project name="projectName" id="projectId">
    <submodels>
        <model path="pathToModel1.mdlx"/>
        <model path="pathToModel2.mdlx"/>
    </submodels>
    <types path="pathToTypes.typx"/>
    <constants path="pathToConstants.constx"/>
    <interfaces path="pathToInterfaces.intrfx"/>
</project>
```

Figure 3.1 Project File Format

- Each model file (mdlx) XML format shall contain minimum of following elements and attributes:

```
<model name="modelName" hash="modelHash" id="modelName">
    <submodels>
      …
    </submodels>
    <startModels>
        <startModel hash="inputModelElement1Id"/>
        <startModel hash="inputModelElement2Id"/>
    </startModels>
    <endModels>
        <endModel hash="outputModelElement1Id"/>
        <endModel hash="outputModelElement2Id"/>
    </endModels>
</model>
```

Figure 3.2 Model File Format

- Each model element other than Connection, Input, Output model element in model file (mdlx) shall be in “submodels” element and contain minimum of following elements and attributes:

```
<model name="modelElementName" hash="modelElementTypeHash"
id="modelElementId">
    <inputConnections>
        <inputConnection portOrder="connectedIndex1"
connectedTo="connectionModelElement1Id"/>
        <inputConnection portOrder="connectedIndex2"
connectedTo="connectionModelElement2Id"/>
    </inputConnections>
    <outputConnections>
        <outputConnection portOrder="connectedIndex1"
connectedTo="connectionModelElement1Id"/>
        <outputConnection portOrder="connectedIndex2"
connectedTo="connectionModelElement2Id"/>
    </outputConnections>
    <conditionConnections>
         <conditionConnection portOrder="connectedIndex"
connectedTo="connectionModelElementId"/>
    </conditionConnections>
    <parameterConnections>
        <parameterConnection portOrder="connectedIndex1"
connectedTo="connectionModelElement1Id"/>
        <parameterConnection portOrder="connectedIndex2"
connectedTo="connectionModelElement2Id"/>
    </parameterConnections>
</model>
```

Figure 3.3 Model Element File Format

- Connection model element in model file (mdlx) shall be in “submodels” element and contain minimum of following elements and attributes:

```
<model name="modelElementName" hash="modelElementTypeHash"
id="modelElementId">
    <startOperation connectedTo="startModelElementId"/>
    <endOperation connectedTo="endModelElementId"/>
</model>
```

Figure 3.4 Connection Model Element File Format

- Input and Output model elements in model file (mdlx) shall be in “submodels” element and contain minimum of following elements and attributes:

```
<model name="modelElementName" hash="modelElementTypeHash"
id="modelElementId">
    <type name="typeOfIOModelElement"/>
    <sourceInstance value="1"/>
</model>
```

Figure 3.5 Input and Output Model Element File Format

- Input Instance and Output Instance model element shall have a text and type elements in addition to Figure 3.3:

```
<copyOf value="sourceInstanceId"/>
```

Figure 3.6 IO Instance Model Element Additional Format

- Textual model element shall have a text and type elements in addition to Figure 3.3:

```
<type name="typeOfTextualModelElement"/>
<text value="text"/>
```

Figure 3.7 Textual Model Element Additional Format

- Numeric Cast model element shall have a cast type element in addition to Figure 3.3:

```
<type name="typeOfNumericCastModelElement"/>
```

Figure 3.8 Numeric Cast Model Element Additional Format

- Previous model element shall have value elements in addition to Figure 3.3:

```
<values>
    <value value="initialValue1"/>
    <value value="initialValue2"/>
</values>
```

Figure 3.9 Previous Model Element Additional Format

- Types, constants, and interfaces file formats are not shown here because they are irrelevant to the current working of the program.

3.1.2 Checker

All requirements in this section are defined to not make any assumption for the design and expect all necessary information from the design to generate a code without a side effect. For example, int64 value is not automatically assigned to an int32 value. If it is necessary to cast one type to another, the user should use Numeric Cast model element.

- Model elements shall have input IO counts as described in Table 3.1.

Table 3.1 IO Requirements for Model Element Types

	Input Count		Output Count		Condition	Parameter
Model Element	**Min**	**Max**	**Min**	**Max**		
Bitwise And Bitwise Or Logical And Logical Or Logical Sharp Addition Multiplication	2	-	1	1	0	0
Bitwise Not Logical Not Numeric Cast Unary Minus	1	1	1	1	0	0
Bitwise Xor Left Shift Right Shift Different Equal Greater Than Equal Less Than Equal Strictly Greater Than Strictly Less Than Logical Xor Modulo Division Subtraction	2	2	1	1	0	0
If	2	2	1	1	1	0
Input Textual	0	0	1	1	0	0
Output	1	1	0	0	0	0
Previous	1	1	1	1	0	1

- Model elements shall only accept types shown in Table 3.2.
  - Numeric: int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, float64
  - Integer: int8, int16, int32, int64, uint8, uint16, uint32, uint64
  - Unsigned: uint8, uint16, uint32, uint64
  - All: all types including bool, char, numeric, array, structure, enumeration.
- Model elements shall not have any unconnected points.
- All outputs in the model shall have definitions.
- There shall not be any cycles of data flow in the model.
- Connection start model element output type shall be the same as end model element input type.

Table 3.2 Model Element IO Types

<table>
<tr><th>Model Element</th><th colspan="2">Input Types</th><th>Output Types</th><th>Condition Type</th><th>Parameter Type</th></tr>
<tr><td>Bitwise And
Bitwise Not
Bitwise Or
Bitwise Xor</td><td>Unsigned</td><td rowspan="11">All inputs shall be the same type.</td><td rowspan="5">Same with input type.</td><td rowspan="3">-</td><td rowspan="4">-</td></tr>
<tr><td>Addition
Multiplication
Numeric Cast
Division
Subtraction
Unary Minus</td><td>Numeric</td></tr>
<tr><td>Modulo</td><td>Integer</td></tr>
<tr><td>If</td><td rowspan="3">All</td><td>bool</td></tr>
<tr><td>Previous</td><td>-</td><td>Same with input type.</td></tr>
<tr><td>Different
Equal</td><td rowspan="4">bool</td><td rowspan="6">-</td><td rowspan="6">-</td></tr>
<tr><td>Greater Than Equal
Less Than Equal
Strictly Greater Than
Strictly Less Than</td><td>Numeric</td></tr>
<tr><td>Logical And
Logical Not
Logical Or
Logical Xor
Logical Sharp</td><td>bool</td></tr>
<tr><td>Left Shift
Right Shift</td><td colspan="2">1: Unsigned
2: Integer</td><td>Same with first input type</td></tr>
<tr><td>Input
Textual</td><td colspan="2">-</td><td>All</td></tr>
<tr><td>Output</td><td colspan="2">All</td><td>-</td></tr>
</table>

3.1.3 Generated Code / Program Output

- Program shall output an error if:
  - XML Reader fails to read the XML files.
  - XML files are missing or garbage information.
  - Model name string given to the program is not in the project.
  - Model check fails.
  - Code generation fails.
  - Verifier fails.
- The generated code shall be in C.
- A header and source file shall be generated for the model.
- In the generated header file:
  - A structure for inputs of the model shall be generated.
  - A structure for outputs and memory variables shall be generated.
  - Declaration of model's Main function shall be generated with parameters of inputs and output structure pointers.
- Input structure shall fit the following template:

```
typedef struct
{
      <input_1_type> <input_1_name>;
      <input_2_type> <input_2_name>;
      …
      <input_n_type> <input_n_name>;
} i_<model_name>;
```

Figure 3.10 Input Structure Template

- Output structure shall fit the following template:

```
typedef struct
{
      bool isInitialCycle;
      <output_1_type> <output_1_name>;
      <output_2_type> <output_2_name>;
      …
      <output_n_type> <output_n_name>;
      <memory_1_type> <memory_1_name>;
      <memory_2_type> <memory_2_name>;
      …
      <memory_n_type> <memory_n_name>;
} o_<model_name>;
```

Figure 3.11 Output Structure Template

- Main function declaration shall fit the following template:

```
void <modelName>Main(i_<modelName>* inputs, o_<modelName>*
outputs);
```

Figure 3.12 Main Function Declaration Template

- In the generated source file, definition of Main function shall be generated.
- Main function shall take two arguments, input structure pointer and output structure pointer.

```
void <modelName>Main(i_<modelName>* inputs, o_<modelName>*
outputs)
{
      …
}
```

Figure 3.13 Main Function Template

- Each model element other than Input and Output shall generate a local variable declaration in Main function in a template of:

```
<model_element_output_type> <model_element_name>;
```

Figure 3.14 Model Element Declaration Template

- Each model element shall generate their corresponding statement to the main function as shown in Table 3.3.
- Main function shall define and calculate the data flow from inputs to the outputs. Each model element shall generate their code in topological order so that no variable is used before its definition.

Table 3.3 Model Element Generated Code Structure

Model Element	Statement Template
Bitwise And	`<model_element_name> = <input_1> & <input_2> & … <input_n>;`
Bitwise Not	`<model_element_name> = ~<input>;`
Bitwise Or	`<model_element_name> = <input_1> | <input_2> | … <input_n>;`
Bitwise Xor	`<model_element_name> = <input_1> ^ <input_2>;`
Left Shift	`<model_element_name> = <input_1> << <input_2>;`
Right Shift	`<model_element_name> = <input_1> >> <input_2>;`

If	`if (<condition>){`
	`  <model_element_name> = <input_1>;`
	`}else{`
	`  <model_element_name> = <input_2>;`
	`}`

Different	`<model_element_name> = <input_1> != <input_2>;`
Equal	`<model_element_name> = <input_1> == <input_2>;`
Greater Than Equal	`<model_element_name> = <input_1> >= <input_2>;`
Less Than Equal	`<model_element_name> = <input_1> <= <input_2>;`
Strictly Greater Than	`<model_element_name> = <input_1> > <input_2>;`
Strictly Less Than	`<model_element_name> = <input_1> < <input_2>;`
Input	`Nothing`
Output	`<model_element_name> = <input>;`
Textual	`<model_element_name> = <text>;`
Logical And	`<model_element_name> = <input_1> && <input_2> && … <input_n>;`
Logical Not	`<model_element_name> = !<input>;`
Logical Or	`<model_element_name> = <input_1> || <input_2> || … <input_n>;`
Logical Xor	`<model_element_name> = <input_1> ^ <input_2>;`
Logical Sharp	`<model_element_name> = <input_1> + <input_2> + … <input_n> < 1;`
Addition	`<model_element_name> = <input_1> + <input_2> + … <input_n>;`
Modulo	`<model_element_name> = <input_1> % <input_2>;`
Multiplication	`<model_element_name> = <input_1> * <input_2> * … <input_n>;`
Numeric Cast	`<model_element_name> = (<cast_type>)<input>;`
Division	`<model_element_name> = <input_1> / <input_2>;`
Subtraction	`<model_element_name> = <input_1> - <input_2>;`
Unary Minus	`<model_element_name> = -<input>;`

Previous	`if (outputs->isInitialCycle){`
	`  <model_element_name> = <parameter>;`
	`}else{`
	`  <model_element_name> = <memory_name>;`
	`}`
	At the very end of the main function:
	`<memory_name> = <input>;`

3.1.4 Verification of the Generated Code

- After code generation, following steps shall be done for verification of the generated code in the order:
  - For each Output and memory model element in the model, an annotation (which is a logical expression) shall be calculated in terms of the Input model elements.
  - For each Input model element in the model, a precondition shall be prepared showing that they are defined before the execution of the generated code.
  - Using Hoare logic on generated code with the prepared precondition, post condition shall be derived after all the statements are executed.
  - Post condition shall be checked if it is logically equivalent to output annotation from the model. If they are not, the program shall output the failure of the verifier.
- Annotations shall be expressed as in Figure 3.15 grammar which is a variation of EBNF [18]. Quick explanation of grammar rules:
  - = definition
  - | choice
  - “ ” terminal
  - ; termination of rule
  - ( ) grouping
  - * none or more
  - + once or more
  - ? none or once

```
Letter      = "A"-"Z" | "a"-"z" ;
Digit       = "0" - "9" ;
ID          = ("_" | Letter) ("_" | Letter | Digit)* ;
Variable    = "var" """ ID (-> ID)? """ ;
Constant    = "const" ID ;
BinaryOp    = "+" | "-" | "*" | "/" | "%" | "&&" | "||" | "^" |
"&" | "|" | "<<" | ">>" | "!=" | "==" | ">=" | "<=" | ">" | "<"
;
UnaryOp     = "-" | "!" | "~" ;
Expression  = "true" | "false" | Variable | Constant |
Expression BinaryOp Expression | UnaryOp Expression ;
Annotation  = "true" | "false" | "Defined" Variable | Expression
":=:" Expression | Annotation ∧ Annotation | Annotation ∨
Annotation ;
```

Figure 3.15 Annotation Grammar

- Each model element shall generate their corresponding annotation as explained in the Table 3.4.
    - Right side of ":=:" annotations are called expressions to use in other model element annotations.
    - All model element variables shall be replaced with their corresponding expressions in another model element annotations.
    - Memory variables in other annotations shall not be replaced with their corresponding expressions.
    - Memory variables annotations shall not be replaced with corresponding model element annotations.
    - "If" model element annotation usage in other annotations shall be done as described in Figure 3.16.

Table 3.4 Model Element to Annotation

Model Element	<op>	Annotation
Bitwise And	&	var <model_element_name> :=: <input_1_expression> <op> <input_2_expression> <op> … <input_n_expression>
Bitwise Or	|	
Logical And	&&	
Logical Or	||	
Addition	+	
Multiplication	*	
Bitwise Not	~	var <model_element_name> :=: <op> <input_1_expression>
Logical Not	!	
Unary Minus	-	
Bitwise Xor	^	var <model_element_name> :=: <input_1_expression> <op> <input_2_expression> <op>
Left Shift	<<	
Right Shift	>>	
Different	!=	
Equal	==	
Greater Than Equal	>=	
Less Than Equal	<=	
Strictly Greater Than	>	
Strictly Less Than	<	
Logical Xor	^	
Modulo	%	
Division	/	
Subtraction	-	
If		((<condition_annotation> :=: true) ∧ (var <model_element_name> :=: <input_1_expression>)) ∨ ((<condition_annotation> :=: false) ∧ (var <model_element_name> :=: <input_2_expression>))
Input		To use in other model element annotations as expression: var <model_element_name> To use in precondition: Defined <model_element_name>
Output		var <model_element_name> :=: <input_1_expression>
Textual		var <model_element_name> :=: const <text>
Numeric Cast		var <model_element_name> :=: <input_1_expression>
Previous		To use in other model element annotations as expression: ((var isInitialCycle :=: true) ∧ (var <model_element_name> :=: <parameter_1_expression>)) ∨ ((var isInitialCycle :=: false) ∧ (var <model_element_name> :=: var <memory_name>)) Addition to post condition: var <memory_name> :=: var <input_1_name>

```
var <out> = <expression1> <op1> var "if" <op2> <expression2>

((<condition> :=: true) ∧ (var "if" :=: <expression3>)) ∨ ((<condition>
:=: false) ∧ (var "if" :=: <expression4>))
```

var "if " replaced in first annotation to second annotation:

```
((<condition> :=: true) ∧ (var <out> :=: <expression1> <op1>
<expression3> <op2> <expression2>)) ∨ ((<condition> :=: false) ∧ (var
<out> :=: <expression1> <op1> <expression4> <op2> <expression2>
```

Figure 3.16 If Model Element in Expression Replacing

- While weakening annotations, following actions shall be done:
  - Remove "true" from ∧ clauses if exists.
  - Remove any "var <id> :=:" (except Output model elements and memories) annotations if "var <id>" is not used on other model element annotations. Because these are already replaced and no longer needed in the final postcondition.
  - An example can be seen from Figure 3.17 to Figure 3.18. Textual1 and Previous1 are removed.

```
((((((Defined (var "isInitialCycle")) ∧ true) ∧
     (var "Textual1") :=: (const "1")) ∧
    (((var "isInitialCycle") :=: true ∧
      (var "Previous1") :=: (const "0")) ∨
     ((var "isInitialCycle") :=: false ∧
      (var "Previous1") :=: (var "Previous1_mem")))) ∧
   (((var "isInitialCycle") :=: true ∧
     (var "Addition1") :=: ((const "0") + (const "1"))) ∨
    ((var "isInitialCycle") :=: false ∧
     (var "Addition1") :=: ((var "Previous1_mem") + (const "1"))))) ∧
  (((var "isInitialCycle") :=: true ∧
    (var "outputs->Output") :=: ((const "0") + (const "1"))) ∨
   ((var "isInitialCycle") :=: false ∧
    (var "outputs->Output") :=: ((var "Previous1_mem") + (const "1"))))) ∧
(var "Previous1_mem") :=: (var "Addition1")
```

Figure 3.17 Unweakened Annotation

```
(Defined (var "isInitialCycle")) ∧
 (((var "isInitialCycle") :=: true ∧
   (var "outputs->Output") :=: ((const "0") + (const "1")))
  ∨
  ((var "isInitialCycle") :=: false ∧
   (var "outputs->Output") :=: ((var "Previous1_mem") + (const "1"))))
 ∧
 (((var "isInitialCycle") :=: true ∧
   (var "Addition1") :=: ((const "0") + (const "1")))
  ∨
  ((var "isInitialCycle") :=: false ∧
   (var "Addition1") :=: ((var "Previous1_mem") + (const "1"))))
 ∧ (var "Previous1_mem") :=: (var "Addition1")
```

Figure 3.18 Weakened Annotation

3.1.5 Quality Requirements

- The program shall run on at least Windows 10 machines with command line arguments.
- The program shall never output non verified generated code.
- The program shall be able to be executed in more than one instance.
- The program code shall be comprehensible since it will be subjected to extensive code reviews as a part of the qualification process.
- The program software shall be readable and understandable since it will be maintained along with IMODE.

3.2 Design and Implementation

According to requirements, because operations should be done in sequential order, like a pipeline, a flow chart can be designed as shown in the Figure 3.19.

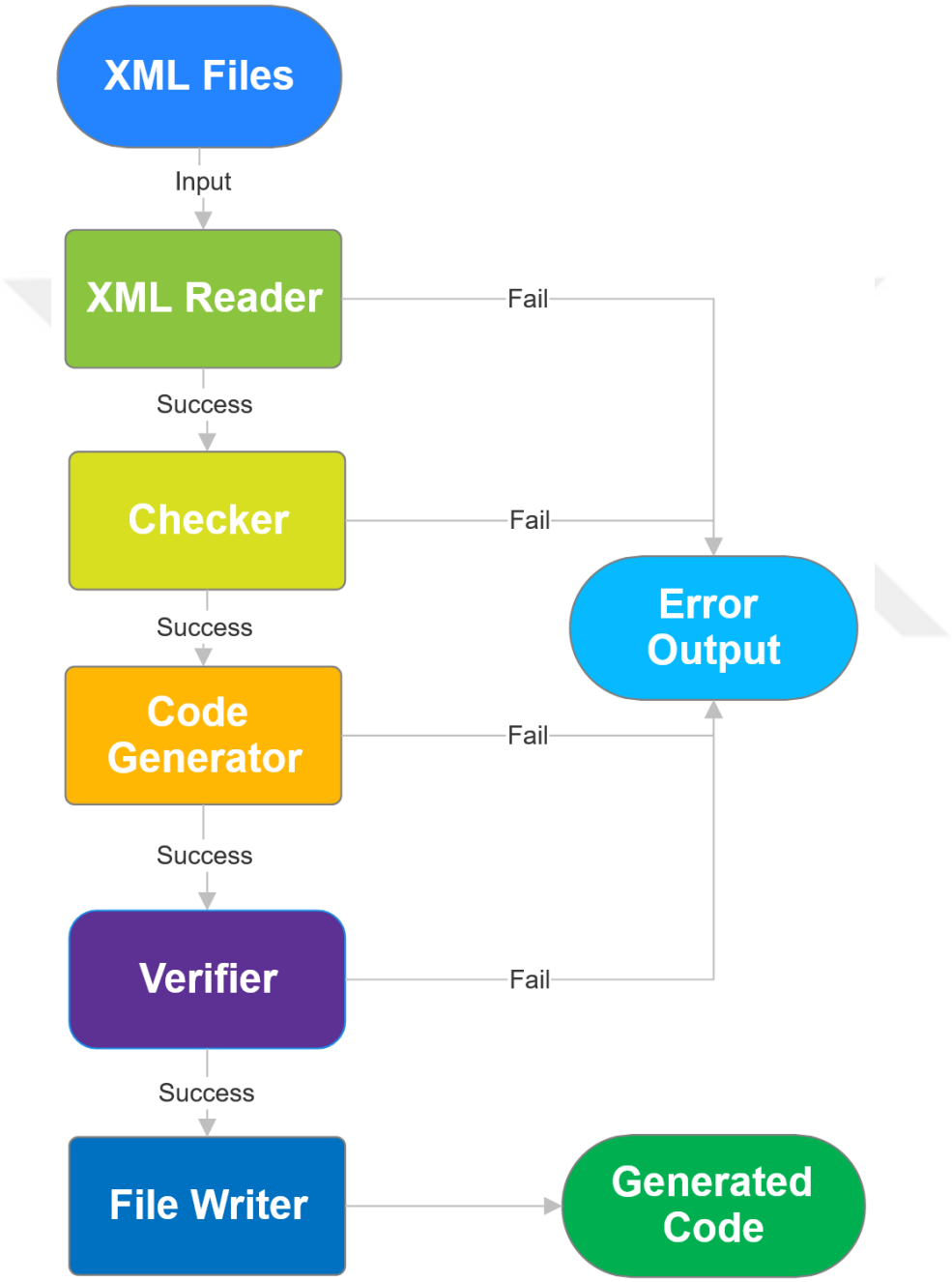


Figure 3.19 Flowchart of the Verified Code Generator

3.2.1 XML Reader

Save files of a project on IMODE is a group of XML files. Details of some XML files are shown in the table below:

Table 3.5 IMODE File Types

Extension	Title	Containing Information
prjx	Project File	• Properties of the project. • Corresponding files for the models, types, constants, packages, and other objects in the project.
mdlx	Model File	• Properties of the model. • Properties of each model element in the model. • Input and output interface of the model. • Connections between model elements. • Visual representation like width, height, position of the box.
typx intrfx constx	Types Interfaces Constants	User defined types/interfaces/constants in the project and their detailed information.

In this project, a single mdlx file, which contains a model, is focused on because it corresponds to a single function in the generated code. A model contains model elements which correspond to operations. These model elements are listed below. Some are obvious by their names, others are explained:

Table 3.6 Model Element Categories

Category	Model Elements	
Bitwise	Bitwise And Bitwise Not Bitwise Or	Bitwise Xor Left Shift Right Shift
Choice	If	
Comparison	Different, Equal Greater Than Equal Less Than Equal	Strictly Greater Than Strictly Less Than
IO	Input Output Textual: For constant values.	
Logical	Logical And Logical Not Logical Or Logical Xor	Logical Sharp: Returns true if inputs contain at most one true, otherwise false.
Mathematical	Addition Modulo Multiplication Numeric Cast	Division Subtraction Unary Minus
Temporal	Previous: Saves the current value of the input and returns previously saved value.	

```
data XmlAttribute : Set where
  Attribute : (name : String) -> String -> XmlAttribute

data XmlElement : Set where
  Element : (name : String) -> List XmlAttribute -> List XmlElement ->
XmlElement
  TextNode : String -> XmlElement
```

Figure 3.20 XML Data Types

To be able to read the information from XML file, I implemented XML Parser in Agda. Used types are shown below:

After parsing the XML, model and model information are interpreted. Because IMODE also needs to show visual expressions for the models, these files contain graphic information as well. Information like width, height, and position of the model element boxes are ignored here, because they do not mean anything for code generation of the logic design.

3.2.2 Checker

After loading the model information, a check is done before code generation on the model so that it is complete and ready for the code generation. This model check is needed because:

- IMODE is a design tool and user can save their unfinished projects to load and continue afterwards. Thus, save files might contain half done designs.
- It is the code generator's responsibility to check compatibility between model elements and integrity of the model, so that user designs according to rules of the model elements and rules of the code generator.

Checked properties are listed below:

- No unconnected input, output, condition, or parameter points in a model element.
- No connections without start or end model element.

- No cycles in the flow of data. This does not mean there are not any cycles in the graph. Previous typed model elements accept cycles because its output does not depend on the current value of the input.
- Input and output counts of the model elements are correct according to their operation.
    - Addition has 2 or more inputs and 1 output.
    - Subtraction has 2 inputs and 1 output.
    - If has 2 inputs, 1 output and 1 condition.
    - …

Each check corresponds to one function in the Agda program and their correctness is done by using Agda's theorem prover. An example is shown below:

```
-- Requirement 3: Addition model elements shall have 2 or more input
points.
req3-1 : ∀ {n id outCons paramCons condCons x y xs} ->
checkInputConnectionCounts (Addition (Properties n id (x :: y :: xs)
outCons paramCons condCons)) ≡ true
req3-1 = refl
```

Figure 3.21 Example Verification for a Requirement

Cycle is prohibited in data flow. Users can connect model elements to each other to create a cycle, but that model should not pass the check. There is an exception to cycle which is Previous model element. As explained before, Previous returns one step before value of the connected input which means Previous' output does not depend on current input's value. This breaks the data flow for that call of the model function and does not create a cycle in the data flow. Cycles that contain any Previous model element shall pass the check.

Cycle check is done by creating a directed acyclic graph or graphs from model elements. There can be more than one graph because of Previous. For each Previous, another DAG is created, because they create a separate data flow. In first DAG, Previous behaves like a source. Next one behaves like a sink.

Because our first data is not ordered by anything, we just have information from file about what is connected to what, we need to find the order for the DAG and create it.

An example without a Previous:

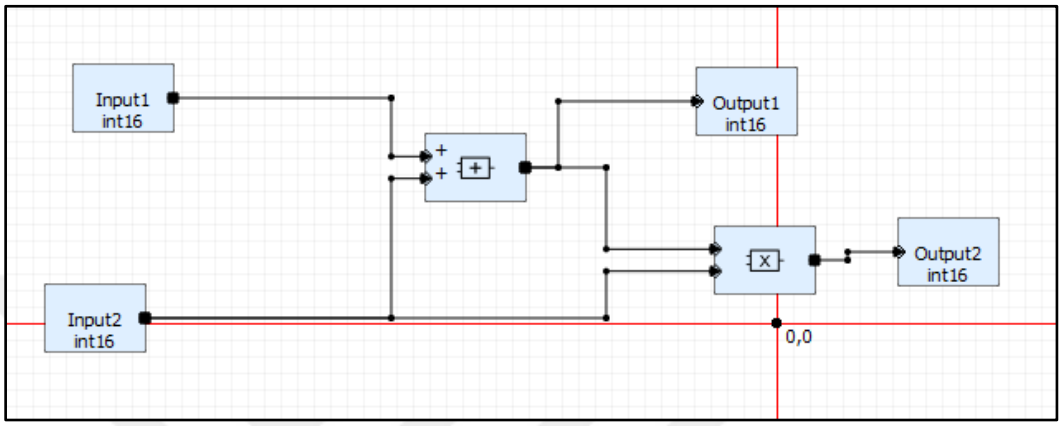


Figure 3.22 An Example Model from IMODE Design

A possible topological order for this model can be:

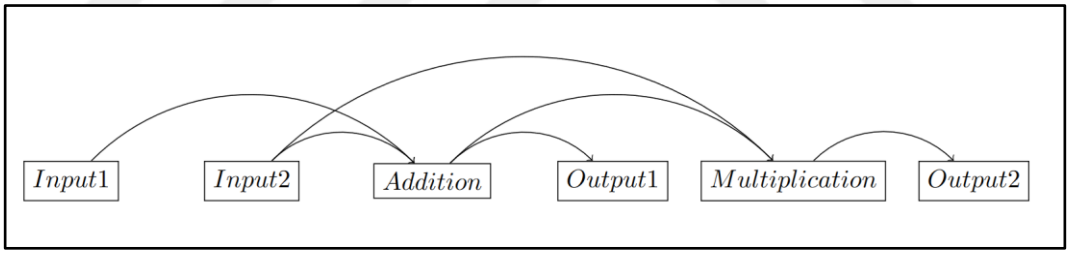


Figure 3.23 DAG for Figure 3.22

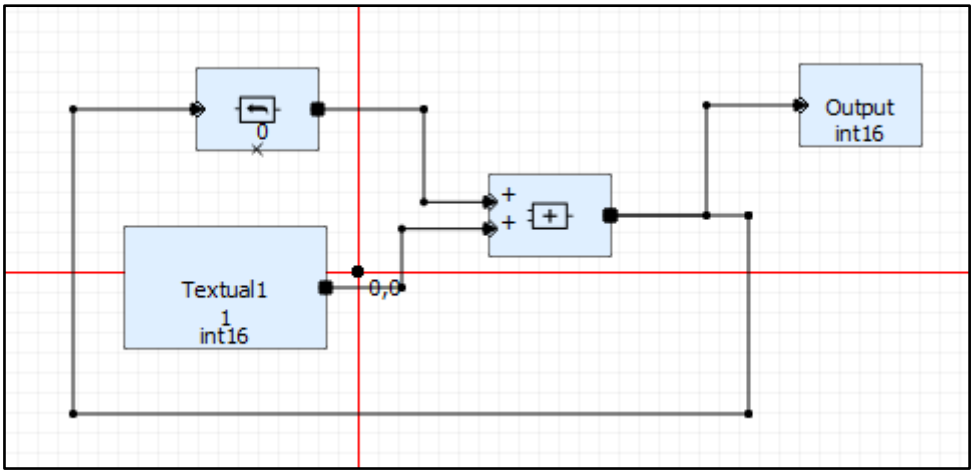


Figure 3.24 An Example Model with Previous and Cycle

DAGs created for the model in Figure 5 are:

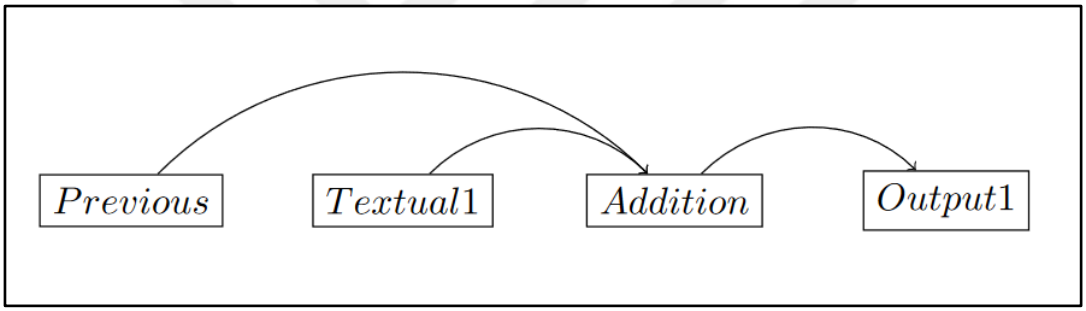


Figure 3.25 DAG 1 for Figure 3.24

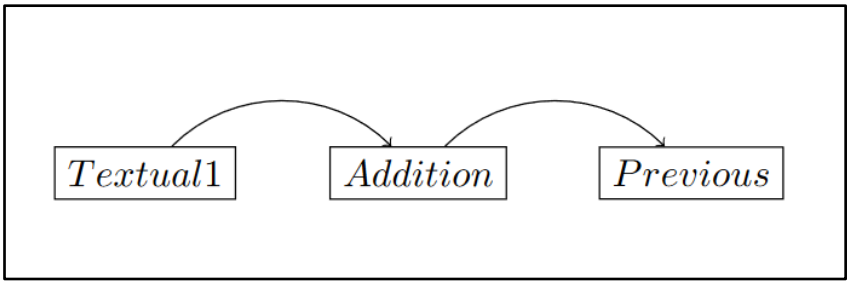


Figure 3.26 DAG 2 for Figure 3.24

Previous is not added again in the second DAG. A DAG should not contain the same model elements multiple times. It looks like Addition has only one input for the second DAG, but this does not affect the code generation for the Addition. The first

DAG manages it. The second one is mostly for the Previous and other model elements if they left unconnected from the Output because of the Previous.

While generating the DAGs if the same model element other than Previous is encountered again, that means there is data flow cycle and model fails the check.

```
data ModelDAG : ℕ → Set where
  ∅   : {n : ℕ} -> ModelDAG n
  _&_ : ∀ {n} (c : Context ModelElement ModelElement n) (g : ModelDAG
n) → ModelDAG (suc n)
```

Figure 3.27 ModelDAG Type Definition in Agda

While arranging the model elements in Agda, Agda already had a DAG Type in their standard library, and I made use of some of the types in the there. However, I created my own DAG type named ModelDAG. Context type is from Agda.

ModelDAG type is bounded by the total model element count in a model which informs the type that this DAG contains at most n number of nodes. This number is needed to ensure the termination check in Agda.

3.2.3 Code Generator

After ensuring everything is okay for model check, code generation is started.

The language of the generated code should be reliant to DO-178 to be used in aviation systems. DO 178 standard defines rules on developed so that it will not any side effects. Dynamic memory should not be allocated. There should not be aliasing, so a subset of C is generated.

The generated code should be consistent with the design done by the user. A main and an initialization function should be generated for the user to run the algorithm that is designed.

While generating a function for the model, every output value shall be calculated. Because DAGs created in the check step are in the data flow dependency order

already, model elements in the DAG just passed one by one to create their own code lines.

```
data StatementType : Set where
    EmptyStatement : StatementType
    Statement : String -> Code -> StatementType
    IfStatement : Code -> List StatementType -> List StatementType
-> StatementType
```

Figure 3.28 StatementType Definition in Agda

Model elements do not directly return the code text/string. Instead, another layer is formed for the generated code. I called it Code Representation. A function will be generated, and functions have statements. So, I have a type for statements:

Code type is for expressions and operations.

After converting model to the List StatementType, List StatementType is converted to text. This process is done with the purpose of another layer to make verification easier. This way string parsing is not needed for verification.

Each model element has its own variables. Input and output variables are given from outside of the main function so that they can be read and written. Memory variables are also contained in output structure so that each time function is called memory variables are also contained from previous call. Other intermediate model elements are defined as local variables inside of the function.

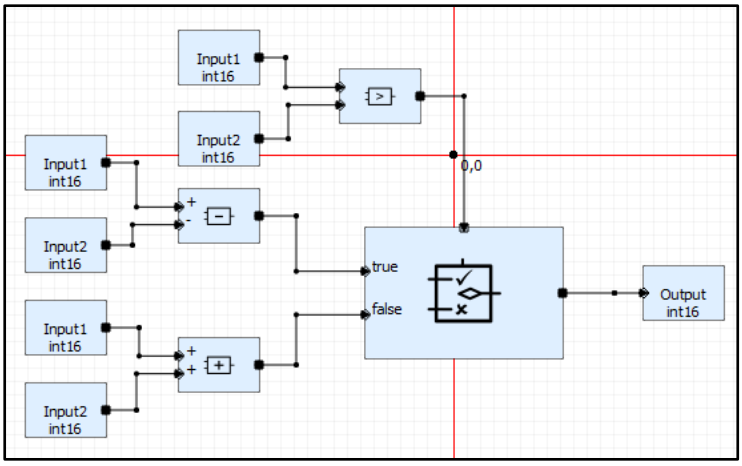


Figure 3.29 Example Model Containing If

If we look at the generated Main function code for the model in Figure 3.29.

```
void ifExampleMain(i_ifExample* inputs, o_ifExample* outputs)
{
    int16 Addition1;
    int16 Subtraction1;
    bool StrictlyGreaterThan1;
    int16 If1;

    Addition1 = inputs->Input1 + inputs->Input2;
    Subtraction1 = inputs->Input1 - inputs->Input2;
    StrictlyGreaterThan1 = inputs->Input1 > inputs->Input2;
    if (StrictlyGreaterThan1)
    {
        If1 = Subtraction1;
    }
    else
    {
        If1 = Addition1;
    }
    outputs->Output = If1;
}
```

Figure 3.30 Example Generated Main Function

“ifExample” is the name of the model in Figure 3.29. “i_ifExample” and “o_ifExample” are structure names that contains input, output, and memory variables (if exist). “int16” corresponds to the short integer which is defined in generated types header with typedef statements. "bool” is also defined as unsigned char.

Generating in DAG order made sure of every model element’s variable is defined before they are ever used in another model element’s variable definition. If model element has else part always so that in all conditions outputs still have a value.

3.3 Verifier

To use Hoare logic in verification, firstly intention of code should be extracted from the model as annotation. Output values should be in terms of inputs or memory variables. These annotations become the postcondition, which is what we want for generated code to do. Precondition can simply be the input variable annotations stating that input values are defined/given to the function as arguments.

The second step is to check if the generated code representation satisfies the wanted postcondition if it starts with the same precondition. Code representation is traversed step by step to calculate middle pre and postconditions which lead to the last postcondition.

An example Hoare logic over Figure 3.29 is shown below. Agda output text is polished about excessive parenthesis and colored for the readability.

For the first step intention of the model, in other means Output value in terms of inputs are extracted.

```
(Defined (var "inputs->Input2")) ∧ (Defined (var "inputs->Input1")) ∧
(((var "inputs->Input1") > (var "inputs->Input2") :=: true ∧ (var
"outputs->Output") :=: (var "inputs->Input1") - (var "inputs->Input2"))
∨
 ((var "inputs->Input1") > (var "inputs->Input2") :=: false ∧ (var
"outputs->Output") :=: (var "inputs->Input1") + (var "inputs->Input2")))
```

Figure 3.31 Annotation from Model

The first precondition is just containing inputs being defined. Each code statement creates a new postcondition to become the next precondition for the next code statement. Variable names are as defined in the model file.

```
⟪ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ⟫
"Addition1 = inputs->Input1 + inputs->Input2;" ∷ []
⟪ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ⟫
∷
```

Figure 3.32 Hoare Logic for Addition Step

```
⟪ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ⟫
"Subtraction1 = inputs->Input1 - inputs->Input2;" ∷ []
⟪ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
⟫
∷
```

Figure 3.33 Hoare Logic for Subtraction Step

```
⦃ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2"))
∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
⦄
"StrictlyGreaterThan1 = inputs->Input1 > inputs->Input2;" ∷ []
⦃ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2"))
∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
∧
(var "StrictlyGreaterThan1") :=: (var "inputs->Input1") > (var "inputs-
>Input2") ⦄
∷
```

Figure 3.34 Hoare Logic for Strictly Greater Than Step

Until this part it was straightforward. When it comes to if statement, if and else part are processed together. No if without an else is generated.

When another variable is encountered other than inputs, they are replaced with their definitions if the definition exists in the precondition. In the next step StrictlyGreaterThan1, Subtraction1, and Addition1 variables are replaced with their definitions in the postcondition.

```
⦃ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
∧
(var "StrictlyGreaterThan1") :=: (var "inputs->Input1") > (var "inputs-
>Input2") ⦄
"if (StrictlyGreaterThan1)" ::
"{" ::
" If1 = Subtraction1;" ::
"}" ::
"else" ::
"{" ::
" If1 = Addition1;" ::
"}" :: []
⦃ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
∧
(var "StrictlyGreaterThan1") :=: (var "inputs->Input1") > (var "inputs-
>Input2") ∧
(((var "inputs->Input1") > (var "inputs->Input2") :=: true ∧ (var "If1")
:=: (var "inputs->Input1") - (var "inputs->Input2")) ∨
 ((var "inputs->Input1") > (var "inputs->Input2") :=: false ∧ (var
"If1") :=: (var "inputs->Input1") + (var "inputs->Input2"))) ⦄
::
```

Figure 3.35 Hoare Logic for If Step

```
⟪ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
∧
(var "StrictlyGreaterThan1") :=: (var "inputs->Input1") > (var "inputs-
>Input2") ∧
(((var "inputs->Input1") > (var "inputs->Input2") :=: true ∧ (var "If1")
:=: (var "inputs->Input1") - (var "inputs->Input2")) ∨
 ((var "inputs->Input1") > (var "inputs->Input2") :=: false ∧ (var
"If1") :=: (var "inputs->Input1") + (var "inputs->Input2"))) ⟫
"outputs->Output = If1;" :: []
⟪ (Defined (var "inputs->Input1")) ∧ (Defined (var "inputs->Input2")) ∧
(var "Addition1") :=: (var "inputs->Input1") + (var "inputs->Input2") ∧
(var "Subtraction1") :=: (var "inputs->Input1") - (var "inputs->Input2")
∧
(var "StrictlyGreaterThan1") :=: (var "inputs->Input1") > (var "inputs-
>Input2") ∧
(((var "inputs->Input1") > (var "inputs->Input2") :=: true ∧ (var "If1")
:=: (var "inputs->Input1") - (var "inputs->Input2")) ∨
 ((var "inputs->Input1") > (var "inputs->Input2") :=: false ∧ (var
"If1") :=: (var "inputs->Input1") + (var "inputs->Input2"))) ∧
(((var "inputs->Input1") > (var "inputs->Input2") :=: true ∧ (var
"outputs->Output") :=: (var "inputs->Input1") - (var "inputs->Input2"))
∨
 ((var "inputs->Input1") > (var "inputs->Input2") :=: false ∧ (var
"outputs->Output") :=: (var "inputs->Input1") + (var "inputs->Input2")))
⟫
::
[]
```

Figure 3.36 Hoare Logic for Output Step

At the end, we reached the Output definition consisting of only inputs. We weaken this last postcondition by removing the local variables that do not affect the output anymore.

```
(Defined (var "inputs->Input2")) ∧
(Defined (var "inputs->Input1")) ∧
(((var "inputs->Input1") > (var "inputs->Input2") :=: true ∧ (var
"outputs->Output") :=: (var "inputs->Input1") - (var "inputs->Input2"))
∨
 ((var "inputs->Input1") > (var "inputs->Input2") :=: false ∧ (var
"outputs->Output") :=: (var "inputs->Input1") + (var "inputs->Input2")))
```

Figure 3.37 Weakened Postcondition from Generated Code

Then we check the output annotation from the model (Figure 3.31) and weakened last postcondition (Figure 3.37) if they are equivalent or not. In this example they are, and using Hoare logic verified that generated code from the model is correct.

CHAPTER 4

APPLICATION EXAMPLE

In this chapter, a complete example extracted from a real application is explained step by step.

4.1 Vertical Throw

We have a model that is designed in IMODE shown in Figure 4.1.

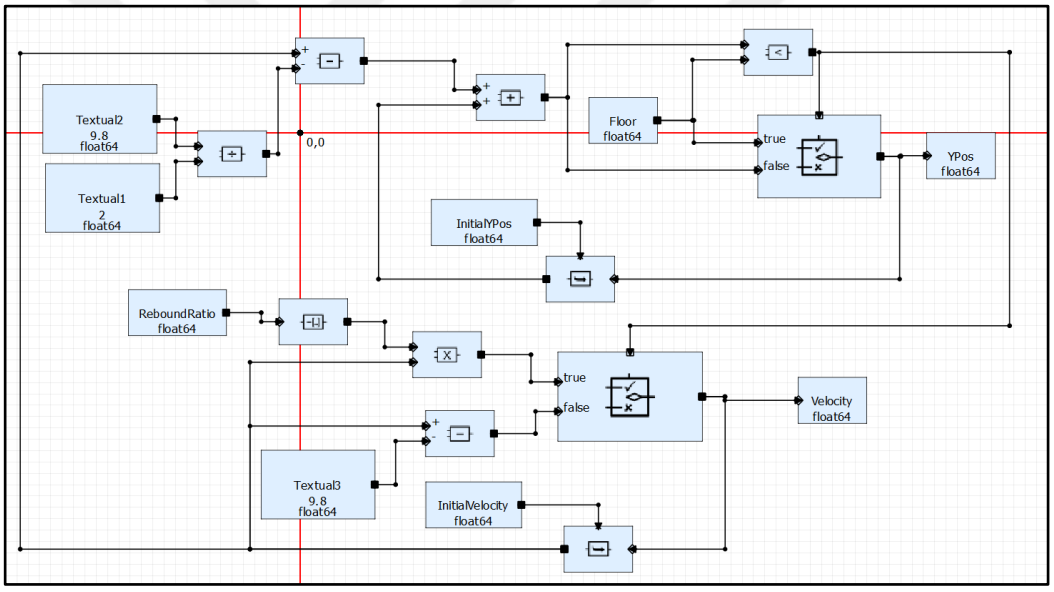


Figure 4.1 Vertical Throw Model in IMODE

Model takes 4 inputs:

- Floor: Floor y coordinate.
- ReboundRatio: ratio of preserved velocity after bouncing from floor.
- InitialYPos: Initial y coordinate of the object.
- InitialVelocity: Initial velocity of the object.

Model gives 2 outputs:

- YPos: New y coordinate for the object after one unit of time.
- Velocity: New velocity for the object after one unit of time.

Object is assumed to have no width or height for simplicity. Model calculates new position and velocity of the object and outputs them. If object hits the floor, it bounces and has a new velocity calculated as $-velocity * rebound\ ratio$. Continuity of the model is done by usage of Previous models. So that model can continue working with previous run's outputs to calculate new outputs. This way we can see object's timeline of y coordinate as its falling to floor, bouncing until it stops. Example graph from IMODE shown in Figure 4.2.

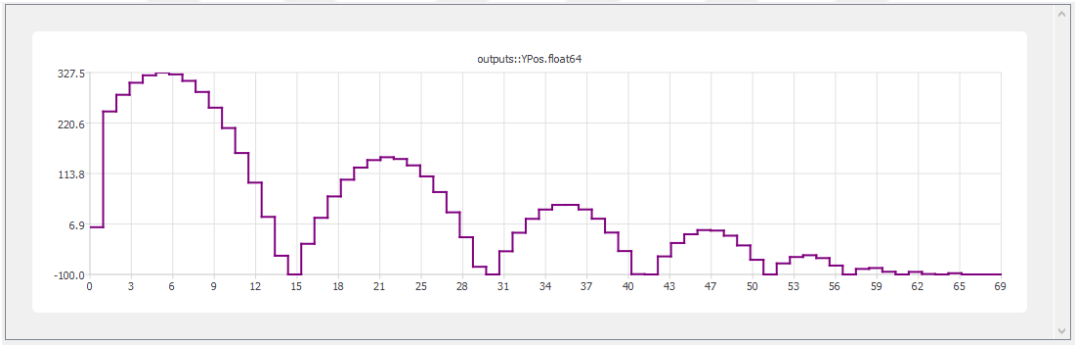


Figure 4.2 Graph of YPos from IMODE

4.1.1 Reader

The model is saved in IMODE and the file "verticalThrowModel.mdlx" is created. File contents can be found in appendix entry, Vertical Throw Save File in page 75.

Necessary information to generate code is read from the file. Beginning of the model value in Agda shown in Figure 4.3. This value is still not parsed to make logical connections between model elements.

```
Operation "verticalThrowModel"
 (Input "Floor" "1720720184416_6" iFloat64 ::
  Input "ReboundRatio" "1720720227382_8" iFloat64 ::
  Input "InitialYPos" "1720720478750_10" iFloat64 ::
  Input "InitialVelocity" "1720720525863_12" iFloat64 :: [])
 (Output "Velocity" "1720720317171_2" iFloat64 ::
  Output "YPos" "1720720400658_4" iFloat64 :: [])
 (PolymorphicDivision
  (Properties "PolymorphicDivision1" "1720720071674_1"
   ("1720720934058_31" :: "1720720089949_6" :: [])
   ((0 , "1720720118192_7") :: []) [] [])  ::
  Textual
  (Properties "Textual1" "1720720085072_1" []
   ((0 , "1720720089949_6") :: []) [] [])
  "2"  ::
  Connection "Connect6" "1720720089949_6" "1720720085072_1"
  "1720720071674_1"  ::
  Subtraction
  (Properties "Subtraction1" "1720720102636_1"
   ("1720720854792_30" :: "1720720118192_7" :: [])
   ((0 , "1720720151894_8") :: []) [] [])  ::
  Connection "Connect7" "1720720118192_7" "1720720071674_1"
  "1720720102636_1"  ::
  Addition
  (Properties "Addition1" "1720720147155_1"
   ("1720720151894_8" :: "1720720449170_23" :: [])
   ((0 , "1720720173739_9") :: (0 , "1720720378545_19") :: []) [] [])  ::
  Connection "Connect8" "1720720151894_8" "1720720102636_1"
  "1720720147155_1"  ::
  Connection "Connect9" "1720720173739_9" "1720720147155_1"
  "1720721413105_1"  ::
…
```

Figure 4.3 Read Model in Agda

Because there is no problem with the xml file or the information that is needed, the program moves on to checker.

4.1.2 Checker

After parsing the model value, created DAGs are visualized in Figure 4.4

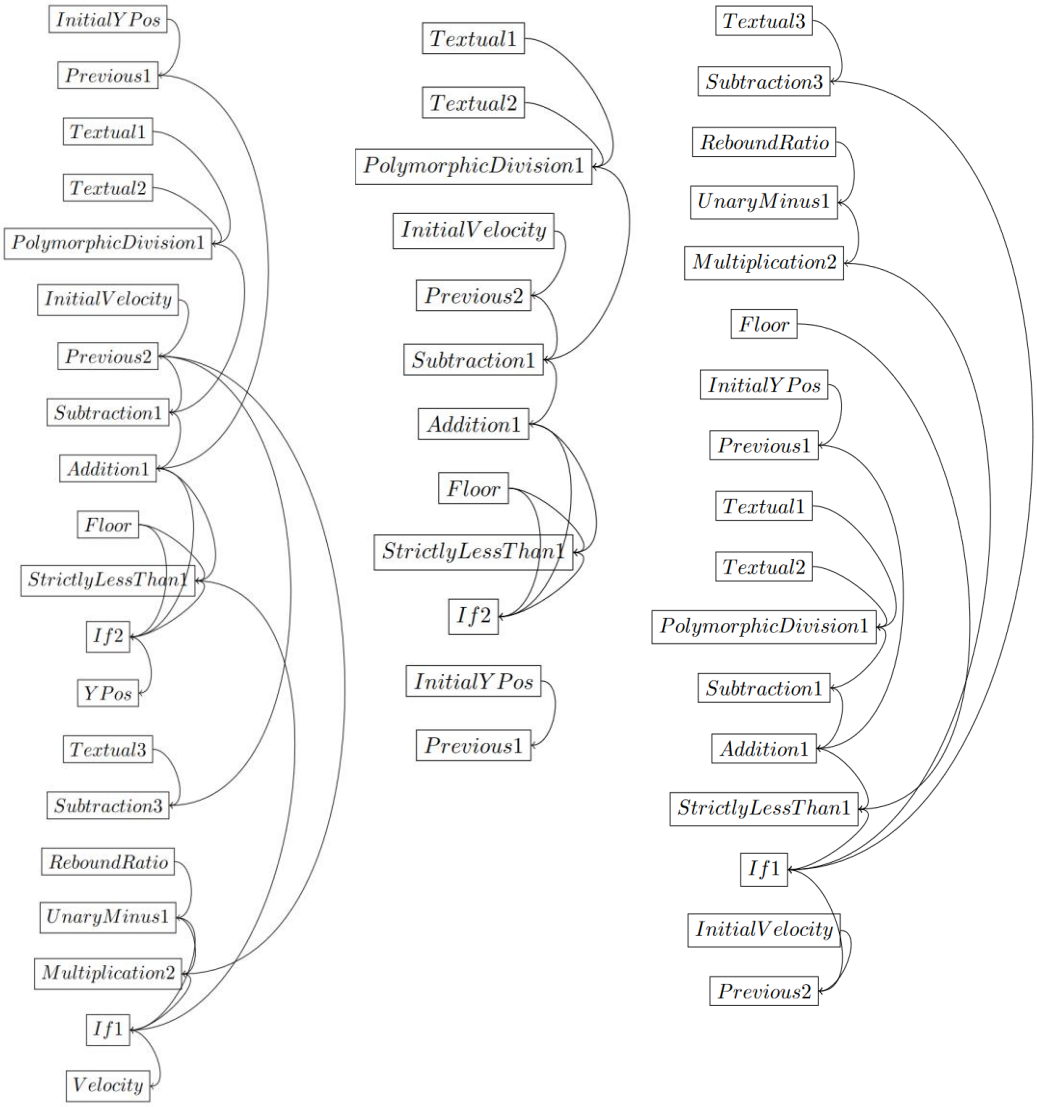


Figure 4.4 DAGs for Vertical Throw Model

The first DAG is created by traversing from Output model elements. In this DAG Previous model elements act as a source, other than their parameter connection, because only input is delayed. Second and third DAGs are Previous model

elements acting as a sink. Now cycles in the model are eliminated by data flows. If there were an actual cycle in the data flow, in separating operation it would be discovered and reported as a check failure.

All types, connections, input output counts are also checked as described in the requirements. Check returns success and program moves on to code generation.

4.1.3 Code Generation

The generator takes previously generated DAGs and starts to generate code. Header file is generated as described in requirements:

```
typedef struct
{
    float64 Floor;
    float64 ReboundRatio;
    float64 InitialYPos;
    float64 InitialVelocity;
} i_verticalThrowModel;

typedef struct
{
    bool isInitialCycle;
    float64 Velocity;
    float64 YPos;
    float64 Previous2_mem;
    float64 Previous1_mem;
} o_verticalThrowModel;

void verticalThrowModel_Main ( i_verticalThrowModel* inputs,
o_verticalThrowModel* outputs );
```

Figure 4.5 Header File Definitions and Declarations for Vertical Throw

For source code Main function is generated. At the start of Main function, local variables are defined.

```
bool StrictlyLessThan1;
float64 If2;
float64 UnaryMinus1;
float64 Multiplication2;
float64 If1;
float64 Previous1;
float64 Previous2;
```

Figure 4.6 Local Variable Definitions in Main Function

Starting from the first DAG all the model elements generated their own code for the Main function in topological order. For the first DAG generated code is shown in Figure 4.7.

```
if (outputs->isInitialCycle)
{
    Previous1 = inputs->InitialYPos;
}
else
{
    Previous1 = Previous1_mem;
}
Textual1 = 2;
Textual2 = 9.8;
PolymorphicDivision1 = Textual2 / Textual1;
if (outputs->isInitialCycle)
{
    Previous2 = inputs->InitialVelocity;
}
else
{
    Previous2 = Previous2_mem;
}
Subtraction1 = Previous2 - PolymorphicDivision1;
Addition1 = Subtraction1 + Previous1;
StrictlyLessThan1 = Addition1 < inputs->Floor;
if (StrictlyLessThan1)
{
    If2 = inputs->Floor;
}
else
{
    If2 = Addition1;
}
outputs->YPos = If2;
Textual3 = 9.8;
Subtraction3 = Previous2 - Textual3;
UnaryMinus1 = - inputs->ReboundRatio;
Multiplication2 = UnaryMinus1 * Previous2;
if (StrictlyLessThan1)
{
    If1 = Multiplication2;
}
else
{
    If1 = Subtraction3;
}
outputs->Velocity = If1;
```

Figure 4.7 Generated Code from First DAG

Because cycles in the model cover the whole model, all needed code is generated in the first DAG. For the second and third DAG, only memory variable definitions of

the Previous model elements are generated. In a case where there are inner cycles with Previous model element, this section would contain another model elements code as well.

```
outputs->Previous1_mem = If2;
outputs->Previous2_mem = If1;
```

Figure 4.8 Generated Code from Second and Third DAG

Generated code from DAGs is not directly converted to strings. It is stored in a CodeRepresentation type and sent to the verifier to check if it really represents the model.

4.1.4 Verification

As described in the requirements, first an annotation is calculated for the Output and memory model elements. Currently no weakening is done to eliminate the same or contradicted condition clauses. This does not cause a problem.

Annotations for YPos and Velocity outputs are shown in Figure 4.9 and Figure 4.10 respectively. Also, there are two memory variables from two Previous model elements. Because they need to use previous cycle's values, three local variables need to be added to logical expression, If1, If2, and StrictlyLessThan1 in Figure 4.11 and Figure 4.12.

```
(((var "StrictlyLessThan1") :=: true ∧ (var "outputs->YPos") :=: (var "inputs->Floor"))
 ∨
 ((var "StrictlyLessThan1") :=: false ∧
  (((var "outputs->isInitialCycle") :=: true ∧
   (((var "outputs->isInitialCycle") :=: true ∧
     (var "outputs->YPos") :=: (var "inputs->InitialVelocity") - (const "9.8") / (const
"2") + (var "inputs->InitialYPos"))
     ∨
    ((var "outputs->isInitialCycle") :=: false ∧
     (var "outputs->YPos") :=: (var "inputs->InitialVelocity") - (const "9.8") / (const
"2") + (var "Previous1_mem"))))
    ∨
   ((var "outputs->isInitialCycle") :=: false ∧
    (((var "outputs->isInitialCycle") :=: true ∧
     (var "outputs->YPos") :=: (var "Previous2_mem") - (const "9.8") / (const "2") + (var
"inputs->InitialYPos")))
     ∨
    ((var "outputs->isInitialCycle") :=: false ∧
     (var "outputs->YPos") :=: (var "Previous2_mem") - (const "9.8") / (const "2") + (var
"Previous1_mem"))))))
```

Figure 4.9 Annotation of YPos from Model

```
(((var "StrictlyLessThan1") :=: true ∧
 (((var "outputs->isInitialCycle") :=: true ∧
  (var "outputs->Velocity") :=: - (var "inputs->ReboundRatio") * (var "inputs-
>InitialVelocity"))
  ∨
  ((var "outputs->isInitialCycle") :=: false ∧
   (var "outputs->Velocity") :=: - (var "inputs->ReboundRatio") * (var "Previous2_mem"))))
  ∨
 ((var "StrictlyLessThan1") :=: false ∧
  (((var "outputs->isInitialCycle") :=: true ∧
   (var "outputs->Velocity") :=: (var "inputs->InitialVelocity") - (const "9.8"))
   ∨
   ((var "outputs->isInitialCycle") :=: false ∧
    (var "outputs->Velocity") :=: (var "Previous2_mem") - (const "9.8")))))
```

Figure 4.10 Annotation for Velocity from the Model

```
(((var "StrictlyLessThan1") :=: true ∧ (var "If2") :=: (var "inputs->Floor"))
 ∨
 ((var "StrictlyLessThan1") :=: false ∧
  (((var "outputs->isInitialCycle") :=: true ∧
   (((var "outputs->isInitialCycle") :=: true ∧
    (var "If2") :=: (var "inputs->InitialVelocity") - (const "9.8") / (const "2") +
(var "inputs->InitialYPos"))
    ∨
    ((var "outputs->isInitialCycle") :=: false ∧
     (var "If2") :=: (var "inputs->InitialVelocity") - (const "9.8") / (const "2") +
(var "Previous1_mem"))))
   ∨
   ((var "outputs->isInitialCycle") :=: false ∧
    (((var "outputs->isInitialCycle") :=: true ∧
     (var "If2") :=: (var "Previous2_mem") - (const "9.8") / (const "2") + (var
"inputs->InitialYPos"))
    ∨
    ((var "outputs->isInitialCycle") :=: false ∧
     (var "If2") :=: (var "Previous2_mem") - (const "9.8") / (const "2") + (var
"Previous1_mem")))))))
 ∧
(var "Previous1_mem") :=: (var "If2")
(((var "StrictlyLessThan1") :=: true ∧
 (((var "outputs->isInitialCycle") :=: true ∧
  (var "If1") :=: - (var "inputs->ReboundRatio") * (var "inputs->InitialVelocity"))
  ∨
 ((var "outputs->isInitialCycle") :=: false ∧
  (var "If1") :=: - (var "inputs->ReboundRatio") * (var "Previous2_mem"))))
 ∨
((var "StrictlyLessThan1") :=: false ∧
 (((var "outputs->isInitialCycle") :=: true ∧
  (var "If1") :=: (var "inputs->InitialVelocity") - (const "9.8"))
  ∨
 ((var "outputs->isInitialCycle") :=: false ∧
  (var "If1") :=: (var "Previous2_mem") - (const "9.8")))))
∧
(var "Previous2_mem") :=: (var "If1")
```

Figure 4.11 Annotation for Previous Memory Variables

```
(((var "outputs->isInitialCycle") :=: true ∧
 (((var "outputs->isInitialCycle") :=: true ∧
  (var "StrictlyLessThan1") :=: (var "inputs->InitialVelocity") - (const "9.8")
/ (const "2") + (var "inputs->InitialYPos") < (var "inputs->Floor"))
  ∨
 ((var "outputs->isInitialCycle") :=: false ∧
  (var "StrictlyLessThan1") :=: (var "inputs->InitialVelocity") - (const "9.8")
/ (const "2") + (var "Previous1_mem") < (var "inputs->Floor"))))
∨
((var "outputs->isInitialCycle") :=: false ∧
 (((var "outputs->isInitialCycle") :=: true ∧
  (var "StrictlyLessThan1") :=: (var "Previous2_mem") - (const "9.8") / (const
"2") + (var "inputs->InitialYPos") < (var "inputs->Floor"))
  ∨
 ((var "outputs->isInitialCycle") :=: false ∧
  (var "StrictlyLessThan1") :=: (var "Previous2_mem") - (const "9.8") / (const
"2") + (var "Previous1_mem") < (var "inputs->Floor")))))
```

Figure 4.12 Annotation for Strictly Less Than from Model

Next step is tom prepare a precondition consisting of Input model elements. Because we have a Previous model element, isInitialCycle is also a memory variable added to keep track of if it is the beginning of the calls. We also tell that it is defined with inputs.

```
(Defined (var "outputs->isInitialCycle")) ∧
(Defined (var "inputs->ReboundRatio")) ∧
(Defined (var "inputs->Floor")) ∧
(Defined (var "inputs->InitialYPos")) ∧
(Defined (var "inputs->InitialVelocity"))
```

Figure 4.13 Precondition Consisting of Input and isInitialCycle Variable

Now this precondition will be given to Hoare logic to work on generated code to see if annotation from the model can be seen in Hoare logic post condition as well.

Generated code has 18 statements, counting if else as one statement. Step by step Hoare logic output from Agda takes 3346 lines. So, I cannot show it here. Last

unweakened post condition can be found in appendix page 78. First three steps of the Hoare logic, again can be found in appendix page 83.

In last precondition, we can found logically equivalent (for this example exactly same) expressions with Figure 4.9, Figure 4.10, Figure 4.11 and Figure 4.12. This equivalence proves that generated code does the same operations on inputs to reach the same annotation for outputs, which defines verification for the code generator.

Because there is no verifier failure, the program returns generated files to the user. Total source file looks like following:

```
#include "verticalThrowModel.h"

void verticalThrowModelMain(i_verticalThrowModel* inputs, o_verticalThrowModel* outputs)
{
    bool StrictlyLessThan1;
    float64 If2;
    float64 UnaryMinus1;
    float64 Multiplication2;
    float64 If1;
    float64 Previous1;
    float64 Previous2;

    if (outputs->isInitialCycle)
    {
        Previous1 = inputs->InitialYPos;
    }
    else
    {
        Previous1 = Previous1_mem;
    }
    Textual1 = 2;
    Textual2 = 9.8;
    PolymorphicDivision1 = Textual2 / Textual1;
    if (outputs->isInitialCycle)
    {
        Previous2 = inputs->InitialVelocity;
    }
```

Figure 4.14 Generated Code for VerticalThrow Model

```
    else
    {
        Previous2 = Previous2_mem;
    }
    Subtraction1 = Previous2 - PolymorphicDivision1;
    Addition1 = Subtraction1 + Previous1;
    StrictlyLessThan1 = Addition1 < inputs->Floor;
    if (StrictlyLessThan1)
    {
        If2 = inputs->Floor;
    }
    else
    {
        If2 = Addition1;
    }
    outputs->YPos = If2;
    Textual3 = 9.8;
    Subtraction3 = Previous2 - Textual3;
    UnaryMinus1 = - inputs->ReboundRatio;
    Multiplication2 = UnaryMinus1 * Previous2;
    if (StrictlyLessThan1)
    {
        If1 = Multiplication2;
    }
    else
    {
        If1 = Subtraction3;
    }
    outputs->Velocity = If1;
    Previous1_mem = If2;
    Previous2_mem = If1;
}
```

Figure 4.15 Generated Code for VerticalThrow Model (cont.)

CHAPTER 5

CONCLUSION

5.1 Evaluation of the Work Done

With usage of Agda theorem prover and writing proofs according to requirements, one can be sure of requirements are satisfied and functions are working as intended even before Agda code is compiled and an executable is created. This is called proof by construction. Proof by construction saves us from generating all possible cases and testing them each time the code is changed.

In addition to the proofs, using Hoare logic provided a layer of verifying the generated code if it is working as it is designed in the model.

5.1.1 Agda as a Programming Language

Using Agda as a programming language was a challenge for me because I was using mostly C and C++ for my work and projects. It took time and patience to learn and do the operations I wanted because of the unpopularity of the language. Emacs was used as an IDE for Agda, but I used Visual Studio Code because of unconventional shortcuts and environment in Emacs. Support for unique shortcuts for Agda in Visual Studio Code that is used in Emacs helped a lot.

Using unicode in source code was visually appealing. Emacs and Visual Studio also support inserting unicode characters, this made typing easy.

Safe options in Agda such as termination check prevents possible infinite loop situations in the code. Even though I know that function will definitely end in some steps, convincing Agda for that was challenging.

I tried to get help through communities and used some generative AI tools, but the generated answer was not compatible with Agda. One reason might be lack of Agda code and discussions in public repositories.

5.2 Future Work

The current situation of code generator is primitive. IMODE is much more complicated. User defined types of interface and constants are not generated in Agda code generator. There are also model elements that can manipulate complex types such as array and structures. Examples are:

- **Projection:** To access an element in a complex type statically.
- **Dynamic Projection:** To access an element in a complex type dynamically including a default value.
- **Assign of a Structured Element:** To change an element in a complex type.
- **Reverse:** To reverse the elements of an array.
- **Split:** To take a portion from an array.

There are model elements that can do loops in a single step. Examples are:

- **Map:** To call a function on an array of values.
- **Fold:** To accumulate a value from an array of values.

A model can be used in another model as reference. The model's inputs and outputs can be connected in another model with other model elements and generate a call of referenced model's main function in code generation.

Templates of models can be defined to be able to call them with different types or different sized arrays. This is like template in C++. Because C does not have template functionality, in code generation all needed functions need to be generated.

To be able to use this code generator for a DO-178C related project, authority needs to be convinced about Agda and processes that are done to verify the code generator.

Top level DO-178C certification wants proper proofs for all the tools, programming languages, operating system calls, and compilers.

REFERENCES

[1] «Ansys SCADE Suite Model-Based Development Environment for Critical Embedded Software,» [Online]. Available: https://www.ansys.com/products/embedded-software/ansys-scade-suite. [Accessed 26 Aug 2024].

[2] E. W. Dijkstra, «A Discipline of Programming,» *Prentice-Hall,* 1976.

[3] G. D. McClellan, "Weakest Pre-Condition and Data Flow Testing," 1995.

[4] Z. Hemel, L. C. L. Kats, D. M. Groenewegen and E. Visser, «Code generation by model transformation: a case study in transformation modularity,» *Springer Link,* vol. 9, pp. 375-402, 2009.

[5] S. Kelly and J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2007.

[6] K. Lano, T. Clark and S. Kolahdouz-Rahimi, «A framework for model transformation verification,» *Formal Aspects of Computing,* vol. 27, no. 1, pp. 193-235, 2015.

[7] M. Völter and B. Kolb, «Best Practices for Model-to-Text Transformations,» in *Modeling Symposium at Eclipse Summit,* 2006.

[8] İ. Ardıç, *Model Based Software Design and Development Tool for Embedded Systems,* 2015.

[9] «RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification,» 2012, p. 116.

[10] "DO-330 Software tool qualification considerations: When, where, and how it applies," [Online]. Available: https://ldra.com/do-330/#DO1. [Accessed 15 July 2024].

[11] «DO-178C Guidance: Introduction to RTCA DO-178 certification | Rapita Systems,» [Online]. Available: https://www.rapitasystems.com/do178#safetylevels. [Accessed 18 Aug 2024].

[12] D. Pinaev, "Qt Nodes," 2022. [Online]. Available: https://github.com/paceholder/nodeeditor. [Accessed 8 June 2024].

[13] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM,* vol. 12, no. 10, pp. 576-583, 1969.

[14] D. Walker, *Hoare Examples & Proof Theory [Lecture Slides],* 2011.

[15] U. Norell, *Towards a practical programming language based on dependent type theory, PhD Thesis,* Chalmers University of Technology, 2007.

[16] "What is Agda?," [Online]. Available: https://agda.readthedocs.io/en/latest/getting-started/what-is-agda.html. [Accessed 25 June 2024].

[17] F. Brooks, *A Constructive Formalisation of Hoare Logic within the Interactive Theorem Prover Agda,* 2021.

[18] W3C, «Extensible Markup Language (XML) 1.0 (Fifth Edition) Section 6,» [Online]. Available: https://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation. [Accessed 17 Aug 2024].

[19] "Entegre Model Tabanlı Tasarım ve Geliştirme Ortamı," [Online]. Available: https://www.numesys.com.tr/scade-suite/. [Accessed 8 June 2024].

[20] R. Harper, "Dynamics," in *Practical Foundations for Programming Languages*, Cambridge University Press, 2016, pp. 39-40.

APPENDICES

A. Vertical Throw Save File

I cannot put the whole file here because it contains so much more information for IMODE to use. Here is a snippet.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE mdlx>
<model             name="verticalThrowModel"             hash="a67070cb51e7193b57ee8ad63c72b3f5"
projectFileName="VerticalThrow.prjx" id="1720720037722_1" description="" tracedRequirements="">
    <submodels>
        <model          visibility="1"          enable="1"          name="PolymorphicDivision1"
hash="32d7b1f51ebe3b5ef0526278e223fcc9"  projectFileName=""  id="1720720071674_1"  description=""
tracedRequirements="">
            <submodels/>
            <ioDirection value="0"/>
            <center xCoord="-59.99362814826121" yCoord="18.515929629347013"/>
            <size width="60" height="40"/>
            <inputConnections>
                <inputConnection portOrder="0" connectedTo="1720720934058_31"/>
                <inputConnection portOrder="1" connectedTo="1720720089949_6"/>
            </inputConnections>
            <outputConnections>
                <outputConnection portOrder="0" connectedTo="1720720118192_7"/>
            </outputConnections>
            <conditionConnections/>
            <parameterConnections/>
        </model>
        <model visibility="1" enable="1" name="Textual1" hash="e0a31b19b95d5690f29a7d9880e8b078"
projectFileName="" id="1720720085072_1" description="" tracedRequirements="">
            <submodels/>
            <ioDirection value="0"/>
            <center xCoord="-174.3304089354219" yCoord="57.421184027751284"/>
            <size width="100" height="60"/>
            <inputConnections/>
            <outputConnections>
                <outputConnection portOrder="0" connectedTo="1720720089949_6"/>
            </outputConnections>
            <conditionConnections/>
            <parameterConnections/>
            <type name="float64"/>
            <text value="2"/>
        </model>
        <model visibility="1" enable="1" name="Connect6" hash="c2459d3d1ef8a0b20f3e7125bae74582"
projectFileName="" id="1720720089949_6" description="" tracedRequirements="">
            <submodels/>
```

```
            <startOperation      connectedTo="1720720085072_1"      xCoord="-124.3304089354219"
yCoord="57.42118402751284"/>
            <endOperation       connectedTo="1720720071674_1"       xCoord="-89.99362814826121"
yCoord="25.18259629601368"/>
            <points>
                <point xCoord="-124.3304089354219" yCoord="57.42118402751284"/>
                <point xCoord="-109.7054208330154" yCoord="57.42118402751284"/>
                <point xCoord="-109.7054208330154" yCoord="25.18259629601368"/>
                <point xCoord="-89.99362814826121" yCoord="25.18259629601368"/>
            </points>
            <type name="float64"/>
        </model>
        <model             visibility="1"             enable="1"             name="Subtraction1"
hash="8738acafef8eef7ddb3f91485d3ef88a"  projectFileName=""  id="1720720102636_1"  description=""
tracedRequirements="">
            <submodels/>
            <ioDirection value="0"/>
            <center xCoord="26.153661435231108" yCoord="-63.389324768818796"/>
            <size width="60" height="40"/>
            <inputConnections>
                <inputConnection portOrder="0" connectedTo="1720720854792_30"/>
                <inputConnection portOrder="1" connectedTo="1720720118192_7"/>
            </inputConnections>
            <outputConnections>
                <outputConnection portOrder="0" connectedTo="1720720151894_8"/>
            </outputConnections>
            <conditionConnections/>
            <parameterConnections/>
        </model>
...
        <model            visibility="1"            enable="1"            name="InitialVelocity"
hash="47652e68b75f740d7c4228759d31a8f5" projectFileName="" id="1720720525863_12" description=""
tracedRequirements="">
            <submodels/>
            <type name="float64"/>
            <sourceInstance value="1"/>
            <last value=""/>
            <default value=""/>
            <value value="0"/>
        </model>
        <model visibility="1" enable="1" name="Velocity" hash="1deb5a48a4655393a18760b265134ef3"
projectFileName="" id="1720720317171_2" description="" tracedRequirements="">
            <submodels/>
            <type name="float64"/>
            <sourceInstance value="1"/>
            <last value=""/>
            <default value=""/>
        </model>
        <model  visibility="1"  enable="1"  name="YPos"  hash="1deb5a48a4655393a18760b265134ef3"
projectFileName="" id="1720720400658_4" description="" tracedRequirements="">
            <submodels/>
            <type name="float64"/>
            <sourceInstance value="1"/>
```

```
            <last value=""/>
            <default value=""/>
        </model>
    </submodels>
    <genericTypes/>
    <sizeParameters/>
    <startModels>
        <startModel hash="1720720184416_6"/>
        <startModel hash="1720720227382_8"/>
        <startModel hash="1720720478750_10"/>
        <startModel hash="1720720525863_12"/>
    </startModels>
    <endModels>
        <endModel hash="1720720317171_2"/>
        <endModel hash="1720720400658_4"/>
    </endModels>
    <inputSensors/>
    <outputSensors/>
    <locals/>
    <lookups/>
</model>
```

B. Last Unweakened Postcondition for VerticalThrow

```
((((((((((((((((((((Defined (var "isInitialCycle")) ∧
                    (Defined (var "inputs->Floor")) ∧
                    (Defined (var "inputs->ReboundRatio")) ∧
                    (Defined (var "inputs->InitialYPos")) ∧
                    (Defined (var "inputs->InitialVelocity")))
                   ∧
                   (((var "isInitialCycle") :=: true ∧
                     (var "Previous1") :=: (var "inputs->InitialYPos"))
                    ∨
                    ((var "isInitialCycle") :=: false ∧
                     (var "Previous1") :=: (var "Previous1_mem"))))
                  ∧ (var "Textual1") :=: (const "2"))
                 ∧ (var "Textual2") :=: (const "9.8"))
                ∧ (var "PolymorphicDivision1") :=: ((const "9.8") / (const "2")))
               ∧
               (((var "isInitialCycle") :=: true ∧
                 (var "Previous2") :=: (var "inputs->InitialVelocity"))
                ∨
                ((var "isInitialCycle") :=: false ∧
                 (var "Previous2") :=: (var "Previous2_mem"))))
              ∧
              (((var "isInitialCycle") :=: true ∧
                (var "Subtraction1") :=:
                ((var "inputs->InitialVelocity") - ((const "9.8") / (const "2"))))
               ∨
               ((var "isInitialCycle") :=: false ∧
                (var "Subtraction1") :=:
                ((var "Previous2_mem") - ((const "9.8") / (const "2"))))))
             ∧
             (((var "isInitialCycle") :=: true ∧
               (((var "isInitialCycle") :=: true ∧
                 (var "Addition1") :=:
                 (((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
                  + (var "inputs->InitialYPos")))
                ∨
                ((var "isInitialCycle") :=: false ∧
                 (var "Addition1") :=:
                 (((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
                  + (var "Previous1_mem")))))
              ∨
              ((var "isInitialCycle") :=: false ∧
               (((var "isInitialCycle") :=: true ∧
                 (var "Addition1") :=:
                 (((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
                  (var "inputs->InitialYPos")))
                ∨
```

```
    ((var "isInitialCycle") :=: false ∧
     (var "Addition1") :=:
     (((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
      (var "Previous1_mem")))))))
∧
(((var "isInitialCycle") :=: true ∧
  (((var "isInitialCycle") :=: true ∧
    (var "StrictlyLessThan1") :=:
    ((((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
      + (var "inputs->InitialYPos"))
     < (var "inputs->Floor")))
   ∨
   ((var "isInitialCycle") :=: false ∧
    (var "StrictlyLessThan1") :=:
    ((((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
      + (var "Previous1_mem"))
     < (var "inputs->Floor")))))
 ∨
 ((var "isInitialCycle") :=: false ∧
  (((var "isInitialCycle") :=: true ∧
    (var "StrictlyLessThan1") :=:
    ((((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
      (var "inputs->InitialYPos"))
     < (var "inputs->Floor")))
   ∨
   ((var "isInitialCycle") :=: false ∧
    (var "StrictlyLessThan1") :=:
    ((((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
      (var "Previous1_mem"))
     < (var "inputs->Floor")))))))
∧
(((var "StrictlyLessThan1") :=: true ∧
  (var "If2") :=: (var "inputs->Floor"))
 ∨
 ((var "StrictlyLessThan1") :=: false ∧
  (((var "isInitialCycle") :=: true ∧
    (((var "isInitialCycle") :=: true ∧
      (var "If2") :=:
      (((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
       + (var "inputs->InitialYPos")))
     ∨
     ((var "isInitialCycle") :=: false ∧
      (var "If2") :=:
      (((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
       + (var "Previous1_mem")))))
   ∨
   ((var "isInitialCycle") :=: false ∧
    (((var "isInitialCycle") :=: true ∧
      (var "If2") :=:
      (((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
       (var "inputs->InitialYPos")))
```

```
          ∨
          ((var "isInitialCycle") :=: false ∧
           (var "If2") :=:
           (((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
            (var "Previous1_mem")))))))))
    ∧
    (((var "StrictlyLessThan1") :=: true ∧
      (var "outputs->YPos") :=: (var "inputs->Floor"))
     ∨
     ((var "StrictlyLessThan1") :=: false ∧
      (((var "isInitialCycle") :=: true ∧
        (((var "isInitialCycle") :=: true ∧
          (var "outputs->YPos") :=:
          (((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
           + (var "inputs->InitialYPos")))
         ∨
          ((var "isInitialCycle") :=: false ∧
           (var "outputs->YPos") :=:
           (((var "inputs->InitialVelocity") - ((const "9.8") / (const "2")))
            + (var "Previous1_mem")))))
        ∨
        ((var "isInitialCycle") :=: false ∧
         (((var "isInitialCycle") :=: true ∧
           (var "outputs->YPos") :=:
           (((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
            (var "inputs->InitialYPos")))
          ∨
          ((var "isInitialCycle") :=: false ∧
           (var "outputs->YPos") :=:
           (((var "Previous2_mem") - ((const "9.8") / (const "2"))) +
            (var "Previous1_mem")))))))))
   ∧ (var "Textual3") :=: (const "9.8"))
  ∧
  (((var "isInitialCycle") :=: true ∧
    (var "Subtraction3") :=:
    ((var "inputs->InitialVelocity") - (const "9.8")))
   ∨
   ((var "isInitialCycle") :=: false ∧
    (var "Subtraction3") :=: ((var "Previous2_mem") - (const "9.8")))))
 ∧ (var "UnaryMinus1") :=: (- (var "inputs->ReboundRatio")))
∧
(((var "isInitialCycle") :=: true ∧
  (var "Multiplication2") :=:
  ((- (var "inputs->ReboundRatio")) *
   (var "inputs->InitialVelocity")))
 ∨
 ((var "isInitialCycle") :=: false ∧
  (var "Multiplication2") :=:
  ((- (var "inputs->ReboundRatio")) * (var "Previous2_mem")))))
∧
(((var "StrictlyLessThan1") :=: true ∧
```

```
    (((var "isInitialCycle") :=: true ∧
      (var "If1") :=:
      ((- (var "inputs->ReboundRatio")) *
       (var "inputs->InitialVelocity")))
     ∨
     ((var "isInitialCycle") :=: false ∧
      (var "If1") :=:
      ((- (var "inputs->ReboundRatio")) * (var "Previous2_mem")))))
   ∨
   ((var "StrictlyLessThan1") :=: false ∧
    (((var "isInitialCycle") :=: true ∧
      (var "If1") :=: ((var "inputs->InitialVelocity") - (const "9.8")))
     ∨
     ((var "isInitialCycle") :=: false ∧
      (var "If1") :=: ((var "Previous2_mem") - (const "9.8")))))))
 ∧
 (((var "StrictlyLessThan1") :=: true ∧
   (((var "isInitialCycle") :=: true ∧
     (var "outputs->Velocity") :=:
     ((- (var "inputs->ReboundRatio")) *
      (var "inputs->InitialVelocity")))
    ∨
    ((var "isInitialCycle") :=: false ∧
     (var "outputs->Velocity") :=:
     ((- (var "inputs->ReboundRatio")) * (var "Previous2_mem")))))
  ∨
  ((var "StrictlyLessThan1") :=: false ∧
   (((var "isInitialCycle") :=: true ∧
     (var "outputs->Velocity") :=:
     ((var "inputs->InitialVelocity") - (const "9.8")))
    ∨
    ((var "isInitialCycle") :=: false ∧
     (var "outputs->Velocity") :=:
     ((var "Previous2_mem") - (const "9.8")))))))
∧ (var "Previous1_mem") :=: (var "If2"))
∧ (var "Previous2_mem") :=: (var "If1")
```

C. First Three Step of Hoare Logic for Vertical Throw

```
《
 (Defined (var "isInitialCycle")) ∧
 (Defined (var "inputs->Floor")) ∧
 (Defined (var "inputs->ReboundRatio")) ∧
 (Defined (var "inputs->InitialYPos")) ∧
 (Defined (var "inputs->InitialVelocity"))
 》
 "if (isInitialCycle)" ::
 "{" ::
 "    Previous1 = inputs->InitialYPos;" ::
 "}" ::
 "else" ::
 "{" ::
 "    Previous1 = Previous1_mem;" ::
 "}" :: []
 《
 ((Defined (var "isInitialCycle")) ∧
  (Defined (var "inputs->Floor")) ∧
  (Defined (var "inputs->ReboundRatio")) ∧
  (Defined (var "inputs->InitialYPos")) ∧
  (Defined (var "inputs->InitialVelocity")))
 ∧
 (((var "isInitialCycle") :=: true ∧
   (var "Previous1") :=: (var "inputs->InitialYPos"))
  ∨
  ((var "isInitialCycle") :=: false ∧
   (var "Previous1") :=: (var "Previous1_mem")))
 》
 ::
 《
 ((Defined (var "isInitialCycle")) ∧
  (Defined (var "inputs->Floor")) ∧
  (Defined (var "inputs->ReboundRatio")) ∧
  (Defined (var "inputs->InitialYPos")) ∧
  (Defined (var "inputs->InitialVelocity")))
 ∧
 (((var "isInitialCycle") :=: true ∧
   (var "Previous1") :=: (var "inputs->InitialYPos"))
  ∨
  ((var "isInitialCycle") :=: false ∧
```

```
  (var "Previous1") :=: (var "Previous1_mem")))
⟫
"Textual1 = 2;" :: []
⟪
(((Defined (var "isInitialCycle")) ∧
  (Defined (var "inputs->Floor")) ∧
  (Defined (var "inputs->ReboundRatio")) ∧
  (Defined (var "inputs->InitialYPos")) ∧
  (Defined (var "inputs->InitialVelocity")))
 ∧
 (((var "isInitialCycle") :=: true ∧
   (var "Previous1") :=: (var "inputs->InitialYPos"))
  ∨
  ((var "isInitialCycle") :=: false ∧
   (var "Previous1") :=: (var "Previous1_mem"))))
∧ (var "Textual1") :=: (const "2")
⟫
::
⟪
(((Defined (var "isInitialCycle")) ∧
  (Defined (var "inputs->Floor")) ∧
  (Defined (var "inputs->ReboundRatio")) ∧
  (Defined (var "inputs->InitialYPos")) ∧
  (Defined (var "inputs->InitialVelocity")))
 ∧
 (((var "isInitialCycle") :=: true ∧
   (var "Previous1") :=: (var "inputs->InitialYPos"))
  ∨
  ((var "isInitialCycle") :=: false ∧
   (var "Previous1") :=: (var "Previous1_mem"))))
∧ (var "Textual1") :=: (const "2")
⟫
"Textual2 = 9.8;" :: []
⟪
((((Defined (var "isInitialCycle")) ∧
   (Defined (var "inputs->Floor")) ∧
   (Defined (var "inputs->ReboundRatio")) ∧
   (Defined (var "inputs->InitialYPos")) ∧
   (Defined (var "inputs->InitialVelocity")))
  ∧
  (((var "isInitialCycle") :=: true ∧
    (var "Previous1") :=: (var "inputs->InitialYPos"))
   ∨
   ((var "isInitialCycle") :=: false ∧
```

```
    (var "Previous1") :=: (var "Previous1_mem"))))
 ∧ (var "Textual1") :=: (const "2"))
∧ (var "Textual2") :=: (const "9.8")
⟫
∷
⟪
((((Defined (var "isInitialCycle")) ∧
   (Defined (var "inputs->Floor")) ∧
   (Defined (var "inputs->ReboundRatio")) ∧
   (Defined (var "inputs->InitialYPos")) ∧
   (Defined (var "inputs->InitialVelocity")))
  ∧
  (((var "isInitialCycle") :=: true ∧
    (var "Previous1") :=: (var "inputs->InitialYPos"))
   ∨
   ((var "isInitialCycle") :=: false ∧
    (var "Previous1") :=: (var "Previous1_mem"))))
 ∧ (var "Textual1") :=: (const "2"))
∧ (var "Textual2") :=: (const "9.8")
⟫
"PolymorphicDivision1 = Textual2 / Textual1;" ∷ []
⟪
(((((Defined (var "isInitialCycle")) ∧
    (Defined (var "inputs->Floor")) ∧
    (Defined (var "inputs->ReboundRatio")) ∧
    (Defined (var "inputs->InitialYPos")) ∧
    (Defined (var "inputs->InitialVelocity")))
   ∧
   (((var "isInitialCycle") :=: true ∧
     (var "Previous1") :=: (var "inputs->InitialYPos"))
    ∨
    ((var "isInitialCycle") :=: false ∧
     (var "Previous1") :=: (var "Previous1_mem"))))
  ∧ (var "Textual1") :=: (const "2"))
 ∧ (var "Textual2") :=: (const "9.8"))
∧ (var "PolymorphicDivision1") :=: ((const "9.8") / (const "2"))
⟫
```