Rajiv Murali
H00022712
MSc Artificial Intelligence
Supervisor: Professor Andrew Ireland

An Investigation into the Automated Generation of provably Correct Code from Formally Verified Designs

School of Mathematical & Computer Sciences
Heriot-Watt University
August 18, 2011
Declaration

I, .................................................... confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: ....................................................

Date: .......................................................
Acknowledgement

I would like to sincerely thank my project supervisor, Prof. Andrew Ireland for his guidance through the course of this project, for sharing his knowledge on the subject matter, and for the references to many interesting materials that aided this paper. In addition, I would like to thank Maria Teresa for an introduction into the Rodin Tool-kit and finally my parents for their loving support.
Abstract

Industries involved in the development of safety critical systems are faced with conflicting demands of meeting both high assurance and productivity within the development of software systems. Auto-code generation has been an effective tool in bridging the stages of specification and implementation, thus improving productivity within software development. However, standards for safety-critical systems dictate a constant degree of verification through both these stages, and most commercially available auto-code generation tools have only targeted code generation and not the verification of the generated code.

This project aims to investigate a novel approach to how properties defined in the specification of a system could be automatically translated and effectively utilized to provably correct code. To this end, we have focused upon the Event-B formal modelling notation, and the SPARK programming language. With this, the SPARK Examiner provides formal program analysis that helps eliminate security vulnerabilities and promote logical soundness within functionality of the system. Hence, by investigating this automatic generation of verifiable program code, an approach is taken towards meeting the standards of high assurance required in the development of safety-critical systems. The outcome of this project was the development of ESpark, an Event-B plug-in for automatic code generation to SPARK Ada. A series of successful experiments of the automatic code generation activity is detailed in the results of this paper. At this stage ESpark covers the sequential subset of Event-B system. Future development of the auto coder could delve into new domains such as concurrent systems and could also look into targeting other program languages such as Java and C.
## Contents

| Declaration | ................................................................. | ii |
| Acknowledgement | ............................................................ | iii |
| Abstract | ............................................................. | iv |
| Contents | ............................................................... | v |

### 1 Introduction

#### 1.1 Motivation

#### 1.2 Formal Methods: Safety Critical Systems

#### 1.3 Formal Methods: Safety Critical Systems

#### 1.4 Dissertation Overview

### 2 Background

#### 2.1 Formal Modelling

##### 2.1.1 Z Notations

##### 2.1.2 The Vienna Development Method

##### 2.1.3 The B-Method

##### 2.1.4 Event-B

##### 2.1.4 Event-B Tool Comparison

#### 2.2 Formal Program Analysis

##### 2.2.1 SPARK/Ada

##### 2.2.2 ESC/Java

##### 2.2.3 Spec#

#### 2.3 Automatic Code Generation

### 3 Requirements Analysis

#### 3.1 Aims

#### 3.2 Sequential Systems: Model vs. Program Code

#### 3.3 Functional Requirements

#### 3.4 Evaluation Requirements

### 4 System Design

#### 4.1 Overview

#### 4.2 Code Generation for Event-B Systems

#### 4.3 Translation Approach
4.3.1 Overview .................................................................................. 28
4.3.2 Translation of Event-B Notations ............................................. 29
4.3.3 Design Example for Code Generation ...................................... 32
4.3.4 Context Translation Phase ....................................................... 35
4.3.5 Machine Translation Phase ...................................................... 39
5 System Implementation .................................................................. 44
  5.1 Overview .................................................................................. 44
  5.2 ESpark: Plug-in Architecture .................................................... 44
  5.3 ESpark: Implementation of Minimum Search Problem .............. 46
    5.3.1 Implementation ..................................................................... 46
    5.3.2 Evaluation ........................................................................... 48
6 Evaluation & Conclusion ................................................................. 33
  6.1 Summary of Strengths and Achievements .................................. 50
  6.2 Limitations .............................................................................. 53
  6.3 Future Work ............................................................................ 53
    6.3.1 Short-term ........................................................................... 53
    6.3.2 Long-term ........................................................................... 54
  6.4 Conclusion ............................................................................... 54
7 Bibliography .................................................................................... 56
A ESpark RODIN Plug-in ................................................................. 59
  A.1 ESpark Installation Guide .......................................................... 59
  A.2 ESpark – How to Activate Auto Code Generator .................... 63
B Event-B Specification ..................................................................... 65
  B.1 Division Algorithm .................................................................... 67
  B.2 Sorted Array Algorithm ............................................................. 70
  B.3 Filter Program .......................................................................... 53
  B.4 Simple Sorting Algorithm ........................................................ 72
  B.5 Summation Program .................................................................. 75
  B.6 Least Common Multiple ............................................................ 77
  B.7 Minimum Search ....................................................................... 78
  B.8 Power Evaluation Program ........................................................ 81
  B.9 Polynomial Evaluation ............................................................. 82
B Event-B Specification ..................................................................... 33
B.1 Division Algorithm ................................................................. 67
B.2 Sorted Array Algorithm .......................................................... 70
B.3 Filter Program ........................................................................ 53
B.4 Simple Sorting Algorithm ......................................................... 72
B.5 Summation Program ................................................................. 75
B.6 Least Common Multiple ............................................................ 77
B.7 Minimum Search ................................................................. 78
B.8 Power Evaluation Program ...................................................... 81
B.9 Polynomial Evaluation ............................................................ 82

C Generated Code ........................................................................ 85
C.1 Division Algorithm ................................................................. 85
C.2 Sorted Array Algorithm .......................................................... 86
C.3 Filter Program ........................................................................ 87
C.4 Simple Sorting Algorithm ......................................................... 88
C.5 Summation Program ................................................................. 89
C.6 Least Common Multiple ............................................................ 90
C.7 Minimum Search ................................................................. 91
C.8 Power Evaluation Program ...................................................... 92
C.9 Polynomial Evaluation ............................................................ 93

D Semantic Analysis Summary ...................................................... 94
D.1 Division Algorithm ................................................................. 94
D.2 Sorted Array Algorithm .......................................................... 95
D.3 Filter Program ........................................................................ 96
D.4 Simple Sorting Algorithm ......................................................... 97
D.5 Summation Program ................................................................. 98
D.6 Least Common Multiple ............................................................ 99
D.7 Minimum Search ................................................................. 100
D.8 Power Evaluation Program ...................................................... 101
D.9 Polynomial Evaluation ............................................................ 102
Chapter 1

Introduction

1.1 Motivation

“Our civilisation runs on software”

- Once an observation by Bjarne Stroustrup, the inventor of C++.

This has become a growing reality as the presence of software is seen as ubiquitous and deeply rooted within modern society. Particularly, it has become a necessity for business and government organisation where elements of reliability and robustness are paramount within most of its operations. Subsequently, industries that are faced with safety, business and security critical concerns have come under great pressure to certify the reliability of their software (Bowen and Stavridou 1993). Minor faults within such safety-critical systems have resulted in disastrous consequences. The Korean Flight 801 is one such example, which resulted from a minute error within the aircraft’s Minimum Safe Altitude Warning (MSAW) system (Greenwell, Strunk et al. 2004). Apart from errors within software development, complex systems are required to operate with no un-expected behaviour in system’s functionality. Government organisations have been prone to threats of complex cyber-attacks, such as the case of Iran’s nuclear program, where a main system controller within the nuclear plant was disrupted by the malicious Stuxnet worm (Falliere, Murchu et al.).

There is a growing demand to certify reliability in the development of safety critical systems (Parnas, van Schouwen et al. 1990). Traditionally, dynamic analysis or testing has been a common practice in identifying bugs within program code. However, this approach has often
proved too expensive and time consuming where attempts to fix errors incur significant expenditure at a later stage in development (Carrington and Stocks 1994). Moreover, industries concerned with providing high assurance in their software intend to perfect their system within the first cycle of development.

Hence, software standards have been set by industry regulators to ensure that the development of safety critical systems follow a more tailored approach. They advocate a constant degree of verification through all stages of software development to ensure correctness and reliability in a systems operation (Berg, Boebert et al. 1982). Standards set by the Ministry of Defence 00-55 (Standard 1991) and the Information Technology and Security Evaluation Criteria (U.K.I.C.) dictate the use of Formal Methods as a mandatory requirement for the development of safety-critical systems.

### 1.2 Formal Methods: Safety Critical Systems

Formal methods have helped induce precision and reasoning to software engineering by invoking the aid of mathematically based techniques (Clarke and Wing 1996). It provokes the developer to further reason about the behaviour of the system with the help of tools such as refinement and decomposition. The implementation of formal methods has been focused on the development of systems that require high assurance and has been successfully applied within domains such as aerospace (Rushby 1993), air traffic control (Hall 1996), financial services (Houston and King 1991), and the military (Bastide, Navarre et al. 2004). As discussed earlier, these industries require a certificate of assurance for their software and formal methods tend to aid this request (Rushby 1995). However, its implementation has often come under resistance within the industry, mostly due to misconception of the language (Hall 1990), but also real problems faced with productivity in software development. Firstly, formal methods tend to follow correctness by construction approach. This consumes more time in design level analysis, where a greater concern is placed in reasoning about the functionality of a system. However, the benefit to this approach is the less effort required when testing the system. Second, there exists a gap between design and code, which can be an error prone process as the model is hand-coded to a targeted programming language. Software standards require both design models and executable code to be verifiable to ensure correctness and high assurance in the system. These aspects in the development of formal methods have introduced a trade-off between high assurance in construction and the lack of efficiency in production. Industries are constrained between these conflicting requirements of meeting regulatory standards and a quicker pace of meeting market demand, reducing the need for high assurance is not possible but a solution to increasing efficiency has been possible through the use automatic code generation.
1.2 Automatic Code Generation: A Secure Bridge

“Other engineering disciplines have to handle a quantum leap into physical reality – the stuff of natural science. In software engineering there is a different quantum leap: that from description to execution.”

- (Fitzgerald and Larsen 1998)

Automated generation of code from design models have been commercially proven to be an effective tool in increasing efficiency (Selic 2003). Most notable In addition, bridging the gap between specification and implementation produces an advantage of maintaining correctness and reliability in translation, if the specification is correct and translation is done correctly. Some commercially available tools such as Simulink (Dabney and Harman 2004) provide automatic code generation from formally designed models to C code. However, the code generated can only be executed and not verified. As discussed earlier, it is a requirement for high integrity software development to produce code that can be mathematically verified. Programming languages such as Ada, C# and Java have access to formal program analysis tools, which provide additional reliability to code by the use of static analysis and automated proof checking. This additional analysis of code is possible by the use of formal assertions that are specified with the program’s source code. An industry related code generation and verification process conducted by QinetiQ on Simulink (Tudor 2004) proved to have positive outcome in the potential for reduction of 60-70% of the software development costs alone, which would translate into a 30-40% reduction in software life cycle cost. The results from the experiment showed a reduction in the overall development effort by 28%. An internal MoD report evaluated the code generation activity of QinetiQ to have positive effect in development life cycle. Thus, this project aims to investigate the automated code generation of formally verified design models to provably correct program code, in order to help meet standards of high assurance and productivity in software development.

1.4 Dissertation Overview

This dissertation is organized as follows:

Chapter 2 – A background on formal modelling techniques and program analysis tools is presented followed by discussion in related work on automatic code generation.

Chapter 3 – The requirements in the development of the automatic code generator is presented, discussing the functional and evaluation requirements of the code generator.

Chapter 4 – Explains the design of the automatic code generator and the reasoning behind the decision that shape its functionality.
Chapter 5 – The implementation of the automatic code generator is discussed based on the design presented in Chapter 4. An example is implemented and evaluated.

Chapter 6 – Concludes by summarizing the achievements of the project, the strengths and weaknesses of the automatic code generator, and identifies remaining issues and challenges to be addressed in short-term and long-term plans for future work.
Chapter 2

Background

2.1. Formal Modelling

“... Precise mathematical definition of what a program does must be present at the origin of its construction. If such a definition is lacking, or if it is too complicated, we might wonder whether our future program will mean anything at all.”

- Jean-Raymond Abrial (Abrial 1996)

Designing the functionality and behaviour of a complex system are problematic aspects in software development. Traditionally within the development of safety-critical systems, formal methodologies have helped provide reliability and correctness in the design of complex systems. It induces an approach of gradual refinement by logic, accompanied with definitions of precise mathematical syntax and semantics (Guttag, Horning et al. 1993). Through its use, developers have been able identify, interrelate and characterize the properties and functionalities of a complex system through reasoned modelling. This notion has long been within the domain of theoretical computing since its early inception in late 1960’s. It was expressed by in his early investigation into the logical foundations of computer programs, that the functionality of a system and all its consequences could be determined from deductive reasoning of the program itself (Hoare 1969).

Decomposition and refinement are two important tools evident in most formal specification languages, some more strongly present in one language than the other (Woodcock 1996).
Decomposition can be used to break down complex systems into smaller components or modules (Abrial and Hallerstede 2007). This for example, enables an abstract system to be built initially on only components that are considered necessary, and then later extended by adding other required modules for additional functionality. Refinement on the other hand, produces different levels of abstraction to the system. This entails that a module can be logically refined to a more precise definition of its desired functionality, proving that systems at even lower levels satisfy the system’s requirements. These traits are especially useful for the development of safety critical systems and also general software development, as they enable efficient structural recovery and functional enhancements.

This literature review further investigates standardized specification languages such as Z, VDM, B-Method and Event-B, which have all be commercially used within the development of safety-critical systems. The review aims at comparing their specification styles and techniques in defining complex systems to help determine which would prove a suitable design platform for model transition to verified program code. Apart from this a simple, but clear example of a vending machine is used to help illustrate the differences between each system model.

2.1.1. Z Notations

Z Notations was created in the early 1970s by Jean-Raymond Abrial and further developed by the Programming Research Group (PRG) at Oxford University. It was evident at the time, a growing divide between practical and theoretical computing, where development practices of software systems often evaded fundamental design principles leading to artificial stability and erroneous consequences. In order to prevent this during software development, Z aided with its use of set theory and mathematical logic to capture essential functionalities of a system in a clear and precise manner (Spivey 1988). At present, Z is one of the few modelling languages that have been ISO standardized.

In comparison to other modelling languages such as VDM and Event-B, Z takes a unique approach in the structural decomposition of a system with the help of schemas (Duce and Fielding 1987). Schemas help represent both static and dynamic properties of the module; enabling a clear division between its declarations and functional operations. To further attribute this, schema calculus can be used for the encapsulation of one schema into another in a process of allowing its properties to be shared by its heir. This enables complex operations to be broken down into many sub-components. As noted by (Woodcock 1996) it has proved a powerful tool in aid of developing precise descriptions of complex system: enabling easy implementation and extension of system modules. One of Z’s earliest and most significant industrial endowers was its implementation towards the specification of the CICS (Customer Information Control System).
Created by IBM United Kingdom, CICS proved to be a technological landmark in its successful management of real time transactions by advocating one of the most secure and recoverable methodologies for data exchange (Phillips 1989). However, as discussed by Woodcock (Woodcock 1996), the complexity of the system later proved erroneous during module extensions. Z was later invoked into the specification of some CICS’s components in hope of reducing the tangible mess created by its former informal specification. The result perceived was a significant improvement in the reliability and quality of the system. An analysis of the CICS 10,000+ lines of source code (See Fig. 2.1) revealed a clear reduction of errors in majority of the system modules.

Fig 2.1: Rate of Errors per KSLOC in CICS Modules (Woodcock 1996)

However, this semi-graphical schema representation creates a wider semantic gap between formal specification and the final program code. During development stages, the Z specification would require to be deciphered from its graphical representation to meaningful programming constructs. This can be seen as one of the drawbacks to the Z specification language, with a lack in explicitly defining the types of properties in the system. For the purpose of auto-code generation, it would aid the translator to know the difference in the properties defined. To further illustrate the Z notation, the example of the vending machine is designed.

<table>
<thead>
<tr>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>available: ( \mathcal{P} ) Items</td>
</tr>
<tr>
<td>unavailable: ( \mathcal{P} ) Items</td>
</tr>
</tbody>
</table>

available \( \notin \) unavailable
available \( \leq \) size
unavailable \( \leq \) size

Fig 2.2: Z Specification - Item Schema
An example of a vending machine is used to illustrate the modelling of properties in Z specification language. Firstly, for a vending machine there exist some number of items, this is represented by the given set \([\text{Items}]\). The system also requires the size of the number of items to be specified by the user. The schema items, as seen in Fig 2.2, illustrate the type definition of list ‘available’ and ‘unavailable’ as having values of the set \([\text{Items}]\). Compared to specifications such as VDM and Event-B, the post and pre conditions of the properties in Z are inexplicitly described in the lower half of the schema.

\[
\begin{align*}
\text{item\_check} \\
\Delta \text{items} \\
i?: \text{item} \\
r!: \text{response} \\
\end{align*}
\]

\[
\begin{align*}
i? \in \text{available} \\
\text{unavailable'} = \text{enrolled} \cup \{i?\} \\
\text{available'} \notin i? \\
r! = \text{sold} \\
\end{align*}
\]

*Fig 2.3: Z Specification – item\_check Schema*

Another schema item\_check (Fig 2.3) describes the operation of the system that is used to verify if an item requested is available in the list and if so, removes it from the available set and moves it to unavailable. From the schema item\_check, it can be seen that the variables of schema item is encapsulated as changeable entity within the item\_check schema. This highlights the advantage of modularisation available in Z. However, in the lower section, the pre and post condition of the operation are mixed together, with no explicit separation. The operation item\_check consists of both; conditions to be check and assignments to be executed. The algorithm design is poorly represented within Z’s specification, providing a loose semantic connection to the final program code.

2.1.2. The Vienna Development Method

VDM-SL is a standardised specification language of VDM, developed by IBM’s Vienna Laboratories in the 1970s. It was initiated from work on functional language description in an industrial environment (Jones 1990). In comparison to Z, its specification structure is made up of keywords without any graphical representations. These keywords are used to help describe the definition of properties of the system. VDM is strictly designed in the sense that rules are given to verify steps of an abstract machine. Its definition of properties can be broken down into key attributes as seen in Fig 2.4.
The above structure of VDM, explicitly defines the invariants, pre and post condition of a module. Z’s schema specification on the other hand, complies with the structure up to the point where the module is divided into ‘State’ and ‘Operation’; after which the module’s properties are not explicitly stated. However, this does not imply that pre and post conditions cannot be expressed in Z; it is only not made explicit for the developer to define within the schema. In Ian Hayes’s (Hayes, Jones et al. 1993) comparison, he complies that it is useful that VDM prompts the developer to think about the preconditions during development, assumption of the design are often forgotten.

The same example of the vending machine is used to illustrate VDM specification and differences of the models can be identified. The first observation noticed as mentioned before was the change from graphical notations to an almost pseudo-programming look in VDM. The specification explicitly specifies the state, components, and invariants, pre-condition and post-condition. Also, its construct of the input and output variables ‘item: i’ is very similar to a methods or functions found in programming language. From this observation, it is evident that the VDM has more opportunities of transitioning the properties of its specification to program code in comparison to Event-B.
2.1.3 B-Method

B-Method provides a model-oriented approach for the construction of software systems. An empirical view of its methodology entails a combination of concepts and techniques from Z and VDM. This is reasonable with Z as they both happen to share the same creator, Jean-Raymond Abrial (Abrial 1996). Also, Abrial explains that techniques of program development and refinement induced in B were learnt from C.B. Jones, the creator of VDM. This collective and combinatory development of B has produced a powerful tool in the construction of software systems.

MACHINE item_check
CONSTANTS available, unavailable
PROPERTIES available, unavailable ∈ items
OPERATIONS
b ← item_check (i) =
PRE i > available THEN
IF i ∈ available THEN
  unavailable := i
END
END

Fig 2.7: B-Method – Vending Machine Specification

The specification of a system in B-Method can seem very similar to writing a pseudo-code of the program itself. However, B-Method allows the developer to explicitly state the properties of the model and use tools such as refinement and decomposition to describe different levels of abstraction. In the example of the vending machine, the definition of the properties can be seen to be illustrated in a systematic way (Fig. 2.7). This code like representation of the system allows B-Method to directly refine the specification to implemented software.
2.1.4 Event-B

Event-B is a derivation of B-Method, which provides an elegant and unique approach in the sole purpose of designing models for the specification of software systems. In comparison to Z which provides only a static representation of the system using schemas, Event-B goes further as to provide a dynamic illustration with pre and post conditions. This is not similar to the specification style of VDM, where the pre and post conditions are explicitly stated. Event-B’s semantics can be observed as an extension of Dijkstra’s guarded commands, where the conditions are specified in an almost pseudo-programming style with, IF (pre-conditions), THEN (Post-Condition), END (ends the condition). This has been proved by Abrial (Abrial 2010) to have the same effect of the pre and post conditioning style as found in VDM, and at the same time provides a closer representation of the specification to low abstract levels of programming code. Aside from this, B follows a derivational modelling style that enables the system to be analysed through an evolutionary process of refinement. Models are not viewed as absolute, but are instead acknowledged from being refined of more basic models, enabling a developer to clearly understand how a system came to be. To further credit this style, (Abrial and Hallerstede 2007) believe that complex system models can only be defined correctly when a method of stepwise refinement is used.

The construction of a system in Event-B is initially broken down into two main components, machine and context. The machine represents the dynamic aspect of the system, which defines variables, invariants and events. While the static elements such as sets, constants and properties (axioms) of the system are defined in the context. A model of the system is constructed in such a way that only one machine component exists, which has access to either one or many contexts components (Fig 2.7). Hence, enabling the developer to easily extend or build additional functionality to a system.

---

**Fig 2.7: Event-B – Machine and Context (Abrial 2010)**

An open extensible tool environment for Event b, Rodin, presents a method for the development of Event-B models. The development platform described provides a wealth of
utilities, from model evaluation tools to proof obligation generation. Also, Rodin is capable of the generating special output files of the models created.

Using the vending machine example, a system is designed in Event-B as seen in Fig 2.8. The system as discussed before is initially broken down into two components, machine and context. The context represents the encoding of the post conditions that are present in the system.

Using the vending machine example, a system is designed in Event-B as seen in Fig 2.8. The system as discussed before is initially broken down into two components, machine and context. The context represents the encoding of the post conditions that are present in the system.

<table>
<thead>
<tr>
<th>Sets: Items</th>
<th>Constant: i</th>
</tr>
</thead>
</table>

**Fig 2.8: Event-B – Post-Condition Encoding**

The precondition is encoded into the machine components of the model, and has access to see the context created above. The precondition requires this access in order to describe the invariants of the system, which are in relation to the constant i described within the context. In comparison to other specification languages, the properties of the system are clearly identified and explicitly stated within the model.

| variable: available |
| variable: unavailable |

| inv0_1: i ∈ available |
| inv0_2: available > 1 |

**Fig 2.9: Event-B – Pre-Condition Encoding**

Finally, events are created in the machine component of the system. In Event-B, there always exists an initializing event, which helps state the property of the elements when the system is executed. The event “itemcheck” is created, and can be seen to have the invariants described earlier, as the guards to executing the assignment of the event.

```
Init
  i := item
```

itemcheck
  when
    i ∈ available
    available > 1
  then
    unavailable = i
  end

**Fig 2.10: Event-B – Events Initialise and Itemcheck**
2.1.5 Event-B Tool Comparison

As discussed in the previous section, the stepwise method for formal system modelling in Event-B poses several significant advantages in enabling developers to reason about the system in design. Apart from this its abstraction permits the development of different modelling domains without any semantic restrain to any particular one, such as reactive, distributed, concurrent systems, sequential programs, or even mixed designs. The RODIN toolset provides an excellent environment in harbouring these benefits, while enabling additional support to third-party developers by its extensibility and configurability in its platform. This section will provide further detail on the RODIN toolset and a final comparison with other modelling tools.

RODIN is built upon the Eclipse platform where programming techniques of automatic debugging without compilation have been effectively used to support system modelling. However, instead of compilation the main interests of this toolset lie in activities of proof obligation generation and automatically discharge of proof conditions. Thus, transparency is essential in allowing users to relate between the system being modelled and the proofs generated in order for any required changes in the system to be made. Also, important difficulties arise from the need of the toolset to support the use of Event-B’s refinement strategy and explicit definition of system properties such as machine and context elements. These needs have been more than effectively been ported to the RODIN toolset, allowing the user to take full advantage of modelling a system in Event, with responsive and incremental feedback mechanisms and its transparency between model and proof. Hence, the role of the developer does not simply lie in entering formal notations into the toolset, but is actively encouraged to reason about the system being modelled.

\[\text{Fig. x. Architectural overview of the Event-B Tool [x]}\]
Apart from effectively meeting Event-B’s formal modelling requirements RODIN provides openness in its own development with extensibility and configurability of its platform by third-party developers. It can be understood that no one tool can solve all development problems and the availability of a range of tools to aid formal development is essential for researchers and industrial uses. RODIN’s platform enables multiple users to actively work on extending and configuring its functionalities of formal modelling. This notion of actively extending the toolset with plug-in has led to support rigorous development, where tools such as model checkers, theorem provers, and translators have been successfully developed and used in RODIN. In addition, the RODIN architecture stores information of model properties (such as invariants, axioms, etc) in repository. A key advantage of this feature can ensure its uptake within the industry where development of plug-ins for specific task related requirements could be met. It is evident that the core RODIN tool has intentionally not been geared towards any particular application in order to make very different uses of its platform possible, not just software development.

As discussed, the RODIN toolset has unique features in its design where great care has been taken to relate proof obligations to the model being designed, enabling the user to quickly return back to the model if a proof has failed. This advantage however has not been overcome by tools such as Abstract State Machine, where large models could lead to the generation of many POs that cannot be maintained. Also, tools such as Isabelle, which has been used with Z, have requirements for the user to explicitly specify proof obligations. RODIN ensures that the proof obligations generated remain extensible and adaptable. Z/EVES is also known to support a prover, however the user is still responsible for specifying relevant POs. In addition, Z/EVES can be seen lack methods for reasoning about the model being designed and is generally good for the purpose of graphical representation of the system. Traditionally, it is understood that a user’s main intention for a tool such as this would primarily revolve around concerns with understanding and reasoning with the model, rather than dealing with proof obligations. Hence, the use of the Event-B tool, RODIN can be seen beneficial over most formal tools, in effectively balancing the modelling and proof stages during the design of a system.

2.2 Formal Program Analysis

Analysis or test of a program is compulsory for any software development project. It ensures that a constructed program is free from errors or bugs, and operates as its specification intended it to. This analysis can be categorised into two types, static and dynamic analysis. Most industrial projects tend to use dynamic analysis: test of a program during its execution in either a real or virtual process. Unfortunately, static analysis has not received similar acceptance within the industry, apart from those that are concerned with high assurance in software development.
(Blanchet, Cousot et al. 2003). This is unfortunate as static analysis provides a complementary and effective aid to dynamic analysis. Dynamic analysis only invokes a process of testing code at a later stage in the development cycle, which is not often cost effective for complex systems. Static analysis is similar to how a programmer would inspect a source code for vulnerabilities, but in an automated and fashion, where patterns that are concerned with programming errors are checked (Christodorescu and Jha 2003), such as:

- Buffer Overflow
- Type Range Violation
- Division by Zero
- References to uninitialized data

It operates in a fashion similar to a compiler, without actually executing the code. Its implementation has been best applied during the construction of code, where a programmer can analyse during development and gain feedback on their code design before testing is even possible. Most static analysis tool perform inspection on the flow of information through the code. This has been useful in identifying ineffective statements, loop stability, inconsistencies between actual and expected data, and more. Studies conducted by IBM have shown that simple automated static analysis has a chance of finding 5 to 15% of bugs in code (IBM 2008). In addition to this, some languages such as Ada, Java and C# have third party static analysis tools that encompass formal methods of verification with the use of code assertions. Similar to how properties of pre and post conditions are defined in formal specification languages. Formal assertions can be defined within program code to reflect on its expected behaviour. Automatic proof generation tools are present within some tools to generate verification conditions of the program code to be later tested. This further builds on the correctness and reliability during program construction.

2.2.1 SPARK/Ada

The development of Ada programming language was initiated by the US Department of Defence after concerns in the cost and maintenance of safety critical systems. The Ada language is a strictly defined language that was able to pass high assurance standards set by industry regulators (Ichbiah, Krieg-Brueckner et al. 1979). However, there still exist security vulnerabilities that arise during the development of software in Ada. The SPARK tool was developed to builds on the strengths of Ada (Barnes 2003), firstly by defining a very strict subset of the Ada language. However, this was not necessarily a syntactic subset, but more of a semantic one. Particularly, the aim of its design can be seen to ensure correctness of a program by producing unambiguous semantics of the program code. This established a singular meaning to the program, regardless of the type of system the software was acting on. This property of SPARK was essential as it
allowed the developer to strictly specify the behaviour of the system, and eased the process of producing provable correct code.

SPARK also removed features within the language that were wholly non-deterministic or of uncertainty. These properties of SPARK combined with the well supported software structure of Ada, provided a strong language for the development of safety critical systems. In addition to this SPARK provides its own verification mechanism known as the SPARK Examiner, which allows units of the code to be verified in isolation then later composed together.

![Diagram of SPARK and Ada subsets](image)

**Fig 2.11: Subset of SPARK and Ada (Barnes and Limited 1997)**

The SPARK system comprises of three main components, firstly the language design itself is specifically designed for the development of embedded and high assurance systems. Spark was built on top of this to enhance the expressive power of the language, particularly the properties that the developer would want to check. For example, a particular safety or security property of the system could be specified as it is intended to be, unambiguously, which can then be passed to the verification tools to be confirmed. The second part of the system is basically a set of verification tools, which are very much inter-linked to the language design itself. These tools can be considered special as they are intrinsically designed to offer analysis of sound nature. The toolset is capable of finding all the bugs of a given program, based on how well it has been correctly annotated. This greatly reduces the dependence of testing during later stages of software development in attempts to find bugs. The final part of the system can be best described as an emerging property, where the approach of developing software with SPARK almost becomes a discipline for developers in the field. The nature of the software formalism provokes the developer to think and reason about the intended behaviour of the system.

The industrial use of SPARK can be seen mainly within the railway, nuclear and aerospace sectors (King, Hammond et al. 1999). The concern of secure systems has become great, such that SPARK has been able to show and prove its dividends to this market. SPARK has been found over the years to be best suited for high assurance projects that tend to be characterised by the
possibility or risk of significant loss as a result of system failure (Barnes 2003). In systems that are safety critical, business critical or security critical, it is not possible to generate evidence of reliability by practical application of the system in real life. For these systems, there is a need to generate some body of evidence or an argument that the system would be fit for purpose. This is a categorically challenging engineering proposal to perform. Again, SPARK aligns with these needs to generate obligations to provide evidence that the system is correct and will meet its requirements before it can be tested. This approach helps identify errors at a much earlier stage in the life cycle of a system, rather than finding them at later stages.

2.2.2 ESC/Java

ESC/Java provides static verification of Java program code during compile time. This was developed by the Compaq Research Group as a tool aimed at finding common programming errors that are not usually detected till the time of program execution, such as null pointer and array out of bound exceptions. Formal verification techniques are used to detect certain types of errors, semantically more thorough than decidable static analysis techniques and considerably more automatic than full functional program verification technology (Leino, Nelson et al. 2000). In Fig 2.12, it can be observed that program verification is on the top-right, indicating a deeper coverage of the program along with a high cost in its effort. Type checking on the other hand is seen at the bottom-right, entailing a lower cost but also a lower coverage of the program. Extended static checking is said to be placed in between these two, providing both an effecting coverage along with reasonable effort.

![Fig 2.12: ESC/Java – Coverage vs. Effort in Program Analysis](image)

ESC/Java annotations are used to effectively provide a powerful tool by expressing expected and intended program behaviour. This can be observed to be similar to the SPARK approach in the use of annotations for specifying program behaviour. However, the newer version ESC/Java2 (KindSoftware 2011) uses a different style of annotations and can be quicker than traditional static analysis methods.
2.2.3 Spec#/C#

Spec# is a program analysis tool for the .Net language C#, extending the languages constructs for null-types, pre and post conditions, and object invariants (Barnett, Leino et al. 2005). It was developed by Microsoft Research team endeavoured to create a systems that removed a programmers need to make assumptions of a systems behaviour based on ambiguous specifications. The development of Spec# has been influenced by SPARK and ESC. It has an automatic code verifier called Boogie (Barnett, Chang et al. 2006).

2.3 Automated Code Generation

The aim of this project investigates the automated translation of formally designed models to verifiable program code. As discussed in Section 1.2, there exists a tight constraint in the development of high integrity software systems, where industrial standards require meeting both high levels of correctness and reduced development time. Formal specification help induce the aspects of reliability and correctness to the design of safety critical system, but as a drawback, consumes a lot of time during its development and implementation of design models to code. Traditionally, specifications are hand-coded by developers in a slow and tedious process, where an error in translation could lead to disastrous consequences such as unrecognized behaviour or worst system failure.

Traditional approaches of design to code implementation utilize a literate approach, were human intervention is required to read and translate the specification. (Grieskamp and Lepper 2000) have approached the development of Ada programs from Z specification through the use of their literate tool Zeta. They aimed at preserving properties of a model through refinement to code, but again the tool relied on human intervention to do so. Also, this literate refinement to code does not support SPARK annotations and hence unable to take advantage of the SPARK Examiner for code verification.

Automatic code generation (ACG) provides an effective technique in bridging this gap between specifications and code, by a process of automatically translating input models to executable system prototypes. This provides an early advantage of executing and testing the system design allowing the developer to form an intuition of further development (Tanik and Yeh 1989). Apart from boosting productivity and reliability, ACG has been documented to show an increase in system performance and efficiency in comparison to hand coded approaches. For example, the RASSP used in the prototyping of embedded signal processes indicated an improvement by 4x in the development time, and GADEA an automated code generation tool achieved a 10x reduction in implementation when compared to its former hand-coded approach.
(Floch, Gorman et al. 1994) describes the use of ProgGen, an industrially used ACG tool that supports flexibility in converting SDL models to different targeted languages. One of the aims of ProgGen was to reduce human intervention during coding phases of the system by generating code skeletons of design models. Apart from its flexibility in implementation, the tool limits itself to only producing code skeletons. This does not take full advantage of the properties described in the design models. Most ACGs tend to provide code skeleton from design models leaving important aspects of the system behaviour for developers to manually implement. This project takes a deeper investigation into the properties defined in formally designed models and its translation to full program constructs.

In context of developing high integrity software systems, there is an obvious requirement for the semantics of design models to be effectively reflected in the targeted language. SCADE (Caspi, Curic et al. 2003) a commercially available tool for the graphical design of complex systems, provides an automated method for the generation of C code from its models. The models designed can be formally defined and verified but the generated code lacks the same verification and validation. Any application of formal program analysis to the code is required to be manually added by the developer. This practice of adding formal assertions to completed code is not recommended as its implementation is best suited by the methodology correctness by construction (Hall and Chapman 2002). Industrial standards such as, MoD standard 00-56 specified by the Ministry of Defence dictate a constant degree of rigour carried through all stages in the development of safety critical systems (Standard 1991). An industry related code generation and verification process conducted by QinetiQ on Simulink (Tudor 2004) proved to have positive outcome in the potential for reduction of 60-70% of the software development costs alone, which would translate into a 30-40% reduction in software life cycle cost. The results from the experiment showed a reduction in the overall development effort by 28%. An internal MoD report evaluated the code generation activity of QinetiQ to have positive effect in development life cycle.

For the purpose of this project a targeted language would be preferable to have formal program analysis, making use of properties defining data and information flow of the desired system, along with program verification tools. As described in Section 2.2.1, SPARK/Ada provides an effective target language that would be able harbour both source code and formal assertions for added program verification.
Chapter 3

System Requirements

3.1 Aims

The overall aim of this project is to investigate the possibility of a secure bridge between stages of specification and implementation in software development. More precisely, the project caters to the development of safety-critical systems and to meet standards that dictate high assurance in all stages of software development. As discussed in Section 2.3.1, the use of commercial off-the-shelf automatic code generation in general has proved successful in improving efficiency in the speed of software development. However, code generated from these tools cannot be automatically verified. The development of safety critical systems requires a constant degree of verification as stated by standards such as MoD 00-55 and ITESC. Hence, this project aims at properties of models designed in a formal specification language and their translation into a programming language that supports formal code analysis. The research hypothesis is stated as follows:

"Properties of formal design models may provide leverage in automatically generating provable correct code."

From discussions in Section 2.1.4, Event-B proved to be a suitable formal specification language that explicitly defined properties of a model and used effective reasoning tools such as refinement and decomposition in its design. In addition to this, it is accompanied by the Rodin Tool-kit that provides a powerful tool in modelling Event-B systems that is also known for its
extensibility. Hence, this research has used Event-B specification language for modelling and investigating properties for code generation.

Fig 3.1: General Overview of the Code Generation Process

The targeted program language for code generation is required to have formal program analysis, which provides both static and dynamic analysis of the generated code. From discussion in Section 2.1.2, SPARK provides a light weight, but effective formal verification tool for the Ada language. It is also widely used within the safety critical industry and more importantly provides verification of the automatic code generation at the implementation level. Hence, this project has aimed its requirements of automatic code generation from formally verified models of Event-B to Ada programming language with the addition of SPARK annotations, which provide formal program analysis of the code generated. A general overview of this automatic code generation process can be seen in Fig 3.1.

Fig 3.2: General Overview of the Verification Process at Code Level
After code generation, it is essential for the Ada program to be verified by both static and dynamic analysis. This can be accomplished by the SPARK Examiner, which not only provides flow analysis and language conformance checks of the code, but also generation of VCs (Verification Conditions). These VCs are potential theorems that can be discharged or proved by the Simplifier, or automatically with a more powerful theorem prover. This helps evaluate verification at the code level. A general overview of this evaluation process is seen in Fig 3.2. This concludes an overview of the aim and requirements of this project. A more detailed specification of the functional and optional requirements is the later sections of this chapter. The next section describes the type of system the code generation is aimed at and the difference in its representation as model and program code.

3.2 Sequential Systems: Model vs. Program Code

Before any step can be taken into specifying the requirements of the code generation tool, it is essential to understand the domain at which the code generator is aimed at and its relative modelling approach in Event-B. As discussed in Section x, it is evident that Event-B is capable of modelling discrete systems of a wide range of domains such as concurrent programs, distributed programs, sequential programs, electronic circuits, reactive systems, etc by intelligent use of system abstraction. However, it would be impossible to focus the project on all these systems for reasons affected by both technicality and the given time scale. Hence, the project aims its requirements at producing an automatic code generator for Event-B models concerned with sequential programs, which lay way to a range of algorithms that can be seen both challenging and interesting to model and subsequently perform code generation, respectively. This section moves on to describe the modelling paradigm of sequential programs in Event-B and what elements used in its design could relate to a formation of a more concrete representation in program code.

---

**Fig 3.3: Program Constructs in Sequential Systems**
Sequential programs can be seen as a collection of individual assignments that are held together by meaningful program constructs. It is essential to understand the relation between the two elements, as constructs play an important role in sequential composition by directing the execution of the individual assignments. Unlike other domains that may involve multiple execution agents or even dependence on interaction with its environment for execution. Sequential programs rely solely on the constructs used to describe the system. The model of sequential system in Event-B would consist of multiple sketches describing the system’s operations, but the final representation at code level would be a single monolithic description of these sketches held together by program. In order to better understand the difference between models of a sequential program to that of program code, a very simple example is used for illustration.

### Event 1

**Guards**

- \( i = n \)
- \( i > \text{min} \)
- \( i < \text{max} \)

**Actions**

- \( \text{sum} := \text{sum} + i; \)
- \( i := i + 1; \)

### Event 2

**Guards**

- \( i = n \)
- \( i \leq \text{min} \)
- \( i \geq \text{max} \)

**Actions**

- \( i := i + 1; \)

### Event 3

**Guards**

- \( i \neq n \)

**Actions**

- \( \text{result} := \text{sum}; \)

---

**Fig 3.4: Representation of Events in a Sequential System**

An Event-B model does not provide any description or detail to the execution of the system. As seen in Fig 3.4, a model would only consist of multiple naked events that show an operation protected by conditions. This allows specific events of the system to be refined in isolation while ensuring other part of the system remain unaffected. An Event-B model would lack knowledge of its scheduling operations, which is information required in the final program code. Hence, an essential requirement of the code generator would rely on the composition of events modelled in Event-B to a monolithic program structure in Ada. This concludes the difference in the description between model and program. The next section specifies the functional and optional requirements of the project.
3.3 Functional Requirements

FR1

The automatic code generation will focus on Event-B models which give rise to sequential systems, including those that are concerned with iterative and conditional constructs.

The code generation would target sequential programs. As discussed in the previous section of this chapter, models designed in Event-B do not provide information on how events are scheduled. Typically, the final refinement of an Event-B model will consist of multiple events describing the systems operation. It will be up to the code generator to automatically produce full monolithic description of the sequential system in Ada program code. Systems concerned with iterative loops and conditional constructs must also be conformant with the automatic code generation.

FR2

The automatic code generator will use both the static and dynamic aspects of the Event-B model to generate loop invariants, pre-conditions and post conditions.

The automatic code generator will require the use of properties of both the static and dynamic components of the Event-B model in order to generate SPARK proof annotation. These annotations, such as pre, post and assert (loop invariant) are required in order for the Examiner to generate verification conditions. The VCs generated aid in the dynamic analysis of the code, by the process of being proved by the SPARK Simplifier. Thus, allowing the code to be verifiable even at the implementation level.

FR3

The automatic code generator should be integrated as far as possible with the RODIN toolset

The automatic code generator produced by this project must incorporate the extensibility of the RODIN toolset to perform code generation activity. By configuring the RODIN tool kit to provide additional functionality of automatic code generation a user would be able to model a system in RODIN, and at the same time perform code generation without any external tool requirement.
### 3.4 Evaluation Requirement

<table>
<thead>
<tr>
<th>ER1</th>
<th>The SPARK simplifier should be used to discharge the VCs that arise from the auto generation code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER2</td>
<td><em>Evaluation set should include all constructs supported by the auto coder</em></td>
</tr>
<tr>
<td>ER3</td>
<td><em>Any undercharged VCs should be investigated.</em></td>
</tr>
</tbody>
</table>
Chapter 4

System Design

4.1 Overview

This chapter will discuss the design approach of the automatic code generation tool. The discussion initiates with a view of code generation for Event-Systems reasoning on the process in regards to the specification language and its components. It later moves onto discuss the translation approach, providing information on the translation phases.

4.2 Code Generation for Event-B Systems

This section delves into the design requirements and reasoning in the approach of providing automatic code generation to an Event-B system. The initial question that was raised during design was concerns regarding the stage at which the code generation would take place, and subsequently the access points between components of an Event-B system, from which model properties can be extracted. From section 2.1.4’s discussion of Event-B, it can be understood that successful code generation can only be made possible from the final refinement stage, where the level of system abstraction is at its lowest. This is simply due to the presence of almost complete information of the system available at that point. Hence, the interface for executing code generation can be understood to have more relevance to exist within RODIN’s refinable machine component, rather than any context component.
The translation process would also require prior information existing in previously refined machine components, for the variable declarations and its type definition. This is because information is gradually added onto the system by refinement, where new variables and invariants are introduced at each refinement level. However, gaining access to the final machine refinement will not supply information of variables and invariants declared in past refinements. Thus, the interface would have to be designed in traversing back through all previously refined machine components in order to collect the required information. Fortunately, an Event-B machine component has information relating to its prior machine refinement stored in definitions called “REFINES”.

Using this definition, the code generator can be designed to collect and store prior refinement information by iteratively traversing through the refined components. Also identification of context components can be acquired from the “SEES” and “EXTENDS” definition. Fig 4.1 illustrates an overall view of the components in an Event-B system and how they relate to each other. Also, it highlights the access points of the interface to machine components, from which related components can be acquired. Through this, the necessary properties of the system have been collected, after which the translation phase is initiated and subsequently ending with the build phase. The translation phase is broken down into individual translation stages between the context and machine components. The final aspect of the design ends with the output of the translated code as Ada specification and body file. In summary, this section has helped reflect on the design of the translation interface to an Event-B system, and lays way to a more detailed discussion of the translation phases. The next section moves on to describe how translation of Event-B notations to Ada and SPARK.
4.3 Translation Approach

4.3.1 Overview

The user interface has been designed to initiate an automatic process of acquiring model properties, which are then passed to subsequent translation and build phases to generate Ada files. This section provides an overview in the design of the translation phases, describing key aspects in relating Event-B models to Ada code and SPARK annotations. This prototype code generator is designed at producing code directly from the properties retrieved, and does not rely on an intermediate language for translation to a target programming language. As discussed in Section 2.3 of related work, code generation proposed by Wright (Wright 2009) utilizes an intermediate language from which C source code can be produced. The code generation produces a translatable subset of Event-B notations to its intermediate language. This process has clear advantages such as removing error in translation of unrecognized Event-B notations and mainly in the possibility of targeting other programming languages such as Java and Ada. However, with the specific requirements of this project targeting both Ada source code and SPARK annotations, any attempt in generating more than one source code would also rely on translation of an accompanying program analysis language. Under the given time scale this would be difficult to accomplish, but is seen as an interesting discussion for future work. The following chapters move on to discuss the Event-B syntax supported by the translation process and the use of the translations rules on properties in the machine and context component.

To begin, the translation of the context and machine properties have been designed to take place in individual phases, initiating with the context translation phase and on completion moving to the machine translation. The reason for this process of translation stems from the method in which models themselves are created in Event-B. Traditionally, modeling in Event-B would require a context component to be first created after which, its elements can be read by operations existing in the machine component. From this reason, the possible use of type definitions identified and defined during the context translation phase can be redefined for variables present in the machine component.

The translations that take place in each of the two phases are stored accordingly into relative Ada and SPARK translation blocks. These blocks are simple containers that are used to store code from the translated properties. The completion of the two phases initiate the final build phase, which is designed to compile the Ada and SPARK translation block into Ada body and specification program constructs.

Note the execution of events translated from the machine component is placed within a single subprogram, in case an Ada procedure. Since the code generation targets only Event-B
models of sequential programs, it is understood that the system would be translated to one monolithic execution in program code. Hence, the execution of the translated system is performed from one procedure call. More information on this can be found in later sections. This concludes a general overview of the translation process. The following sections will discuss the translation of Event-B’s modelling properties to translation blocks attained from the context and machine component.

4.3.2 Translation of Event-B Notations

Before any steps can be taken into describing the translation of an Event-B system, it is essential to first describe the translation of Event-B notations to Ada code and SPARK annotation. Formal mathematical notations are a key aspect of modelling in Event-B, where a system is described in a more detailed and precise manner. They are extensively used in forming axioms and invariants of a system and can vary in complexity of its description. For the purpose of designing a prototype code generation tool, only an essential subset of Event-B notations is investigated for translation to Ada program code and SPARK annotations. This is also because more complex mathematical notations are almost impossible to be supported by Ada or SPARK syntax. However, the notations that have been investigated prove to be able to translate to essential description of the system in a programming language. The remainder of this section will describe these investigated notations and its subsequent translations.

To begin, one of the requirements of translation is the identification and definition of types. From observations of modelling systems in Event-B, declaration of a constant or variable would require a form of type definition entailing the possible element (or elements) of the object declared. This is described by using the element-of notation (∈) in Event-B, by definitions of axioms and invariants of the system. It is important to note, that even when modelling an Event-B system in the RODIN toolset, the declaration of constants and variables are prompted with an explicit requirement for a type definition using an element of notation, as seen in Fig 4.2. This highlights some clear similarities in the construction of models using RODIN to that of writing program code in an IDE (Integrated Development Environment).
Fig 4.2: Screenshot of RODIN – Prompting a type definition using Axiom.

From this, an investigation was applied in identifying type definitions and declaration of Ada program code that can be translated from invariants and axioms. Table 4.1 describes the translation of these type definitions of Event-B notations to type definitions and declarations in Ada program code.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Event-B</th>
<th>Read</th>
<th>Ada</th>
<th>Program Type</th>
</tr>
</thead>
</table>
| 1.   | n ∈ ℕ   | n is an element of Natural Numbers | **Context Declaration:**
*Default constant value will be defined*  
Context Declaration:  
n: Constant;  
**Machine Declaration:**  
n: Naturals;  
| Global Constant |
| 2.   | x ∈ 1‥n | x is an element of range 1 to n. | **Type Definition:**  
Type x_Type is range 1..n;  
**Declaration:**  
x: x_Type;  
| Range Type |
| 3.   | x ∈ 1..n → ℕ | x is an element of array of range 1 to n of Natural Numbers | **Type Definition:**  
Type x_Array is array (x_Type) of Naturals;  
*x_Type will be defined if not already done so.*  
**Declaration:**  
x: x_Array;  
| Array Type |

Table 4.1: Event-B notations for Type Definitions
Apart from type definitions, axioms and invariants also provide information of the possible states of a constant or variable within the execution of the system. This is done through the use of predicates (Boolean-valued functions). It is understood from discussion of Event-B in Chapter 2, that axioms of the context component entail pre-conditions that are true before the execution of the system, while invariants specify conditions that are true during the execution of the system. These predicate definitions would require to be substituted within SPARK’s proof annotations during the translation process. More information on this process is provided in the following sections of this chapter. Some of these notation’s relational operators cannot be directly substituted to the syntax of SPARK annotations and require translation, as seen in Table 4.2. Also, the same translation of relational operators applies to guards found within events of machine component, and is subsequently similar in translation to Ada code.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Event-B</th>
<th>Read</th>
<th>Ada and SPARK</th>
<th>Ada/SPARK Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.</td>
<td>x = n</td>
<td>x is equal to n</td>
<td>x = y</td>
<td>Conditional/Predicate</td>
</tr>
<tr>
<td>6.</td>
<td>x ≠ n</td>
<td>x is not equal to n</td>
<td>x /= n</td>
<td>Conditional/Predicate</td>
</tr>
<tr>
<td>7.</td>
<td>x &lt; n</td>
<td>x is lesser than n</td>
<td>x &lt; n</td>
<td>Conditional/Predicate</td>
</tr>
<tr>
<td>8.</td>
<td>x ≤ n</td>
<td>x is lesser than or equal to n</td>
<td>x &lt;= n</td>
<td>Conditional/Predicate</td>
</tr>
<tr>
<td>9.</td>
<td>x &gt; n</td>
<td>x is greater than n</td>
<td>x &gt; n</td>
<td>Conditional/Predicate</td>
</tr>
<tr>
<td>10.</td>
<td>x ≥ n</td>
<td>x is greater than or equal to n</td>
<td>x &gt;= n</td>
<td>Conditional/Predicate</td>
</tr>
</tbody>
</table>

*Table 4.2: Event-B notation for Conditionals*

Apart from Event-B notations of axioms and invariants, there is also requirement in translation of assignments that are found in events of the machine component. These translations are fairly straightforward, where some changes to the arithmetic operators are required. Examples of these translations can be found in Table x. Functional assignments are not supported within this prototype code generator, as there are no direct requirements in the need to support functions in sequential programs at this stage. This concludes the translation of Event-B notations that are essential in producing Ada code and SPARK annotations. The following section will delve into an example model designed in Event-B, which will be subsequently used to aid the description in the translation process of the context and machine components.
### Table 4.3: Event-B notation for Assignment Actions

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Event-B</th>
<th>Ada</th>
<th>Program Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.</td>
<td>( n := x \ast y + z )</td>
<td>( n := x \ast y + z; )</td>
<td>Arithmetic Assignment</td>
</tr>
<tr>
<td>12.</td>
<td>( n := x &lt;{ x \mid &gt; y } )</td>
<td>( n(x) = y )</td>
<td>Array Action</td>
</tr>
</tbody>
</table>
| 13.  | \( n := x <\{ x \mid > y \} <\{ y \mid > x \} \) | \( n(x) = y; \)  
|      |         | \( n(y) = x; \) | Array Actions |

#### 4.3.3 Design Example for Code Generation

In order to best describe the design features of the translation process, an example model of an Event-B sequential program is used to help illustrate properties within the machine and context components. This Event-B system will allow a clear description of how Ada code and SPARK annotations are derived from it. For the purpose of explanation the example used is fairly simple to understand, but also expressive and interesting in design. A general description of the search can be described as an iterative search for the minimum value within an array of natural numbers. The design of the model starts off with the definition of the context component. As mentioned in section x, the context represents the static properties of the system and are represented by the elements such as sets, constants and axioms. These parameters are specified in Fig 4.3, which describes the context consisting of a constant \( n \) that is a natural number, an array \( f \) of size \( n \). As can be seen, there are axioms describing the constants declared. These are pre-conditions of the system that requires being true before the system executes. The next phase in modelling is the design of the machine component.

#### Constants: \( n, f, \)  

\( \text{thm0}_1: \ n \geq 1 \)

\( \text{axm0}_1: \ n \in \mathbb{N} \)

\( \text{axm0}_2: \ f \in 1..n \Rightarrow \mathbb{N} \)

**Fig 4.3: Event-B Context Component of Minimum Search Problem**

The machine component consists of variables, invariants, and events that describe the functionality of the system. It is different from context component, in the ability to refine itself to a more concrete representation of the system. As can be seen, a variable \( x \) is defined with invariants describing its conditions, which is considered to be always true during the execution of
the system. Here there are four invariants that describe the variables defined. Some can be identified as type definitions, as mentioned in the previous section. Also, for every variable declared there is a compulsory initialization of a value using an initialise event. The events describe the operations that execute within the system. Here the initialise event is understood to take place before all other events. Apart from this, two other events are created, final and progress. The final event represents the end state of the systems operation, with only guards and no specific actions. The progress event is seen to modify the state’s x and has a status Anticipated, which indicates that it will change to convergent on the next refinement.

---

**Variables:** x

- **Initialise**
  
  \[ x : \mathbb{N} \]

- **progress**
  
  **status**
  
  Anticipated
  
  **then**
  
  \[ x \in 1..n \]
  
  **end**

- **final**
  
  **when**
  
  \[ x = n \]
  
  **then**
  
  skip
  
  **end**

**Fig 4.4: Event-B Machine Component of Minimum Search Problem**

The second refinement to the system modifies the status of the event progress to convergent and introduces a variant. This indicates that the event is repetitive and the termination is guaranteed. The guard for progress is updated to execute the event till the value of \( x \) reaches that of \( n \). Here, \( n \) is the size of the array and \( x \) is initialised as 1, meaning the event will execute iteratively by incrementing the value of \( x \) as its action till it reaches \( n \). This refinement has produced the search aspect required by the system to traverse through the array \( f \).
The third and final refinement splits the convergent event progress to two events namely, progress_1 and progress_2. Both these events the same guard from the previous refinement which checks if the value x has reached the total size $n$ of the loop. In addition, a variable $minimum$ is added onto the system, which is used to hold the minimum value during the search of the array. This introduces a new guard to each of the two new events of the refinement, one which checks if the value of minimum is greater than the current value of the array, and the other checks if it is lesser. If the current array value is lesser than that of $minimum$, then the value the array is copied into minimum. If not, the value x is incremented with knowledge that variable $minimum$ has the lowest value up to the current point in the array. An invariant is also added, which specifies that minimum is always lesser than that of the array $f$ of indices from 1 to $x-1$.

This model represents a sequential search process for the minimum value in an array, and is be used to describe the translation phases in the following sections, and also in Chapter 5 of System implementation.

<table>
<thead>
<tr>
<th>Initialise</th>
<th>Progress</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1$</td>
<td>status</td>
<td>status</td>
</tr>
<tr>
<td></td>
<td>Convergent</td>
<td>Ordinary</td>
</tr>
<tr>
<td></td>
<td>when</td>
<td>when</td>
</tr>
<tr>
<td></td>
<td>$x \neq n$</td>
<td>$x = n$</td>
</tr>
<tr>
<td></td>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td></td>
<td>$x := x + 1$</td>
<td>skip</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>Variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n - x$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig 4.5: Event-B Second Refinement of Machine Component**
In event-B the context component consists of sets, constants, axioms and can be understood to represent the static aspect of the system, which are read by events running in the machine component. It is important to note that the properties of the context cannot be changed within machine’s operation of the system, and are only intended to be read. Thus, the translation of the context properties is viewed as generating the global properties of the system in Ada, of which the machine translation depend on. Here, the constants of Event-B directly relate to global definitions, which in most cases also produce global type definition. Apart from global definitions, the context translation also requires to produce appropriate spark annotations. In this case, the use of global identifiers global declarations in Ada would also have requirements to produce SPARK annotations in describing the Ada package’s own global definitions. For this purpose, the translation of the context properties is further divided in to subsections of Ada and SPARK translations. Fig 4.7 highlights the Ada and SPARK translation blocks that are required to be derived from the context component. The design example of the Minimum Search problem is used to illustrate the translation process.

**Fig 4.6: Event-B Machine Component of Minimum Search Problem**

### 4.3.4 Context Translation Phase

In event-B the context component consists of sets, constants, axioms and can be understood to represent the static aspect of the system, which are read by events running in the machine component. It is important to note that the properties of the context cannot be changed within machine’s operation of the system, and are only intended to be read. Thus, the translation of the context properties is viewed as generating the global properties of the system in Ada, of which the machine translation depend on. Here, the constants of Event-B directly relate to global definitions, which in most cases also produce global type definition. Apart from global definitions, the context translation also requires to produce appropriate spark annotations. In this case, the use of global identifiers global declarations in Ada would also have requirements to produce SPARK annotations in describing the Ada package’s own global definitions. For this purpose, the translation of the context properties is further divided in to subsections of Ada and SPARK translations. Fig 4.7 highlights the Ada and SPARK translation blocks that are required to be derived from the context component. The design example of the Minimum Search problem is used to illustrate the translation process.

- **Variables:** \( x, \text{minimum} \)
- **inv1_1:** minimum \( \in \mathbb{N} \)
- **inv1_2:** \( \forall i \cdot i \in 1 \cdot x-1 \Rightarrow \text{minimum} \leq f(i) \)

**Initialise**

\[
\begin{align*}
x &:= 1 \\
\text{minimum} &:= f(n)
\end{align*}
\]

**Progress_1**

- **status** Progress
- **when**
  - \( x \neq n \)
  - minimum \( \leq f(x) \)
- **then**
  - \( x := x + 1 \)
- **end**

**Progress_2**

- **status** Progress
- **when**
  - \( x \neq n \)
  - minimum \( > f(x) \)
- **then**
  - minimum \( := f(x) \)
  - \( x := x + 1 \)
- **end**

**Final**

- **status** ordinary
- **when**
  - \( x = n \)
- **then**
  - skip
- **end**

---

Variables: \( x, \text{minimum} \)

**inv1_1:** minimum \( \in \mathbb{N} \)

**inv1_2:** \( \forall i \cdot i \in 1 \cdot x-1 \Rightarrow \text{minimum} \leq f(i) \)
Fig 4.7: Translations from Context Properties

As mentioned, the translation phases would be initiated once all model properties are acquired. The context translations for Ada code, initiates with the task of finding global declarations that are required to be defined. This process of identification is derived from the list of constants that are defined in the model’s context component, as each constant equates to global declaration that is required to be defined within the Ada package. Axioms define the property or condition of a constant, which are checked by the translator to produce type definitions along with subsequent declaration of the identifier. To better illustrate this process, the example sequential program for searching for minimum value in an array is used to illustrate the design of the translation. Each constant is shown to relate to an axiom, which is designed to initiates a translation to Ada code based on the notation describing it. Table 4.1 provides the ref numbers that indicates the type of translation. The translated code is stored in accordance to translation blocks that are assigned to it. Also, type definitions are mapped into storage with its defining Event-B notation, with regards to it being re-used again to define variables in the machine component.

Using the minimum search example, the translation of the context component’s properties is described. The design process initiates by assuming that all constants need to be declared. The process searches for axioms that provide a type definition for a given constant and produces appropriate Ada translation, which is store in appropriate translation blocks. Fig 4.8, describes the process of translating constant n, with its type definition axiom (axm0_2).
A more complex translation can be seen when defining constants that represent an array function. Fig 4.9 describes the breakdown of the axiom (axm0_3), into the definition of a range type, which is later used by the definition of the array. Finally, the constant \( f \) is declared with the produced array type definition. Note that any type definitions of natural or integer of a constant within the context component is directly translated to a constant type definition. This because they are often used to specify range type definitions, and Ada only allows this if the declaration type is constant.

This covers the general design in the translation process of the context properties to Ada code. Apart from translation to Ada code, the SPARK annotations are also a requirement to be produced by the translation process. First, the use of global declarations within an Ada package...
would automatically require the definition of SPARK’s own annotation, which specifies the control access to Ada’s package global variable. This is a fairly straightforward process, where during the process of global declaration, they are also added onto the translation block of SPARK’s own annotation. Apart from the own annotations, the global annotation of SPARK is mandatory in using the SPARK Examiner. From discussion in section x, a major aim of SPARK is to strengthen the interface of the Ada subprograms such as procedures and functions. In this case, the execution of events within the machine is performed within a single Ada procedure, where the definitions of the context properties are read by the events of the machine component. Since the context properties translate to global definitions within the Ada package, the SPARK Examiner would require the definitions of the global annotation on the interface of the Ada procedure. Note, it is understood that the context properties remain unchanged and are only read by the machine. Thus all global definitions found from the context properties are placed within the translation block of SPARK’s global annotation. This is also accompanied by modes that define the flow of data. In this case, the mode for any definition in the global annotation is set to in, as they are known to be only read by operations within the procedure. Apart from this, derives annotation would be defined within the translation of the machine component, as information flow would require knowledge of the machine’s variable definitions.

The SPARK Examiner is also capable of performing check on the dynamic behaviour of the Ada code using proof annotation. The proof annotations consist of pre and post annotations, which provide information of the pre-conditions and post-conditions of the Ada procedure. The proof annotations also consist of assertion annotations that describe loop invariants within a procedure containing a loop construct. These annotations enable the Examiner to produce verification conditions (potential theorems), which can be later proved by external tools such as SPADE Simplifier. In relation of proof annotations to Event-B models, it can be understood that the context component represents the pre-conditions of the system. These entail a set of facts that must be true before the execution of the machine’s operations. The context component uses its axioms to specify conditions that apply to constants that are declared. In translation the constants are defined as global declarations of the Ada package, and in relation the axioms that describe them can be used as pre-conditions in SPARK’s pre proof annotation. However, not all axioms are substituted to the pre annotation, only those that represent a predicate relation as seen in Table 4.2. This is because, unlike Event-B which can use a range of mathematical notations to describe axioms and invariants, SPARK is limited to common relation programming syntax, which is considerable limited. The proof annotations, post and assert will be discussed in the machine translation.
4.3.5 Machine Translation Phase

The translation of properties such as variables and events are placed within a single Ada procedure. This decision was a follow up of the discussion in Section 3.2 of requirements, where the events of the final refinement would converge into one monolithic system operation, which can be executed within a single Ada procedure; there are no requirements in the use of Ada functions to represent individual events. Also, from observations in attempts at hand-coded translation of Event-B sequential models to Ada program construct, it was evident that global type definitions produced from context properties were at times required to be used to define variables within machine context i.e. the procedure. Hence, it was seen beneficial to have the context translation to take place first and then move on to the translation of machine properties, where pre-defined types in the translator could be re-used. The design of the machine translation discusses the translation of variables and invariant properties, and also the process of merging events of sequential programs to monolithic executions of Ada program code, which are placed within translation blocks.

Fig 4.10: Translations from Machine Properties

The translation of the machine’s variables and invariants properties, aim at producing local variables within the Ada procedure. This is similar to the translation of context properties to global definitions within the Ada package. As mentioned in the previous section, the type definitions of the global variables are designed to be reused by the local variables. The local variables are not defined within the procedure, but are passed as formal parameters of the procedure call. This choice in design was required to satisfy the use of local variables used in proof annotations, where declaration of local variables within the procedure is unreadable by the SPARK examiner.

In regards to the execution of the system, the operational design of a model in Event-B is represented as events within the machine component. This representation very specifies the
operations of the event by the use of assignments and the conditions in which they are executed by means of guards. This method of representing a system’s functionality is very appealing during the design of the system. However, its representation cannot be directly substituted to an execution within a programming language. Here, the translation aims to merge the events to a single monolithic execution in Ada program code. For sequential programs in Event-B, Abrial defines the use of merging rules that combine events corresponding to a particular logical construct to produce conditional (IF) or loop (WHILE) statements, as can be seen in Fig. 4.11.

![Fig 4.11: Merging Rules of Events to Sequential Constructs](image)

However, it can be observed that both merging rules, M_IF and M_WHILE, have the same guards in the preceding events that merge. This can be problematic in differentiating between the two constructs. In order to diffuse this Abrial (Abrial 2010) states the following:

“The M_WHILE rule requires that the antecedent event appears as new or non-anticipated, thus convergent at one refinement level below that of the second one. In this way, we are certain that there exists a variant ensuring that the loop terminates.”

This indicates a special requirement for the M_WHILE merge rule, stating that during the design of a model, an event that intends to repeat its execution within the system would require its status to be first declared as an antecedent event, and then refined to a convergent status in the next refinement level. This ensures the presence of a variant within the machine that entails the termination of the event. From this, the identification of a while conditions would require a mapping of the convergence in relation to the level of refinement. This design in translation of events to Ada program construct is described using the search example from Section x.
As can be seen, Fig 4.6 represents the final refinement of the machine component, which consists of four events namely Initialise, Progress_1, Progress_2, and Final. The translation of events starts with the initialise event, which on observation is the only event not to have any guards. This is because, the event is intended to be initialised at the very start of the machine’s execution and provides the initial values for the variables declared. It is translated using the rules defined in Table 4.3 to Ada code, and stored within the Variable Initialisation translation block.

The merging rules are applied to the remaining events, where the level of convergence would have been obtained by traversing through past refinements. The status and refinements of these events can be viewed in Fig 4.12, where the origin of each event and its preceding status can be checked. The translation first aims at identifying any event that has a convergent status in its previous refinements. The event is assigned a convergence level, which indicates to what refinement level the event was found to be convergent. This is important in enabling the translation to identify nested loops, where event having lower convergence tend to form the outer loop, while events with higher convergence levels are placed as loops within them. Finally, using Abrials merging rules the execution of the events are placed in appropriate sequential constructs.

The translation attained from this are placed within the Events Translation block, which is built within the procedure call soon after the translated variable initialisation block.

---

**Fig 4.12: Events at each refinement level from Abstract to Concrete.**

The machine component consists of operations that contribute to the flow of information by the use of global definitions. SPARK enables the use of derives annotation to describe data
flow. This states the dependence of the final value of uninitialized formal parameters (translated from the variables of the machine) from global definitions (from constants of the context). Since the global definitions are all assumed to be constant, were their modes are defined as in within SPARK’s global annotations. The translation has been designed to set all formal parameters to derive from global definitions of the same type. This ensures that all uninitialized formal parameters are provided with a derives annotation, that can be analysed by the SPARK examiner.

In regards to the proof annotations, the assert annotation is directly derived from the invariants present within the machine component. This is similar to the translation of axioms as annotation for defining SPARK’s pre condition. The invariants describe conditions that are true within all events of the machine component, including events that are meant to loop. Hence, invariants of predicate type are directly substituted to the assert annotation. In addition, the assertion to a given loop would also represent the guards of the loop. The post-condition is however not as direct as the assert annotation. It is required to represent the condition of the machine at the end of its execution. The invariant present within the machine component can be manipulated to form the post condition. The final events guard of the machine component indicates the termination condition of the loop, which checks the variable $x$ meets constant $n$. The logical quantifier defined in invariant (inv1_2) of the machine component, has its variable $x$ substituted with constant $n$. This manipulation of the invariant, by replacing the variable with the constant allows the condition to be set as SPARK’s post proof annotation. However, this manipulation is limited to invariants of logical quantifiers. The final build phase makes use of the translation blocks generated from the translation of the context and machine component. The build phase uses the templates of the Ada body and specification file (as seen in Table 4.4 and 4.5), to substitute the translation blocks in them. This concludes the final phase of the translation process.
package <package_name />
--#own <SPARK-Translation: own annotations/>;
is
<Ada-Translation: Global Identifier Definitions>
   <Constant Definitions />
   <Range Type Definitions />
   < Array Type Definitions />
</Ada-Translation: Global Identifier Definitions>

procedure <procedure_name /> (<Ada-Translation: Formal Parameters />);
--#global <SPARK-Translation: global annotations />;
--#derives <SPARK-Translation: derives annotations />;
--#pre <SPARK-Translation: pre annotations />;
--#post <SPARK-Translation: post annotations />;

end <package_name >;

Table 4.4: Translation Blocks in the Ada Specification

package body <package_name /> is

procedure <procedure_name /> (<Ada-Translation: Formal Parameters />) is
begin
<Ada-Translation: Parameter Initialization />
<Ada-Translation: Event Translation />
end <procedure_name >;

end <package_name >;

Table 4.5: Translation Blocks of the Ada Body
Chapter 5

System Implementation

5.1 Overview

This chapter discusses the architecture of the RODIN plug-in, ESpark. The plug-in incorporates the functionalities specified in Chapter 4, Design. It later discusses the implementation of the example minimum search problem and its evaluation using SPARK.

5.2 ESpark: Plug-in Architecture

As stated in the requirements (Chapter 3), sequential models designed in Event-B are aim at translation to SPARK Ada code. The RODIN toolset provides a strong platform for modelling Event-B specifications. It is built on top of an Eclipse framework enabling key advantages of extensibility and configurability in its functionality via plug-ins. To meet the functional requirements of this project, a fully automatic code generator for Event-B models is required in generating AdaSPARK code. This has been achieved by developing a third-party RODIN plug-in called ESpark.

The ESpark plug-in has been developed in Java using the standard Eclipse Plug-in Development Environment. From specifications in the design of the system, it was understood that the code generation would be required to be activated by the user, at some stage in the final refinement. Thus, the architecture of ESpark extends the *rodin.core.ui* to add an additional user
interface onto the tool bar of the machine component. It was important to extend only the machine component as it leads to the final refinement, where the system is at low levels of abstraction. The activation of the interface begins the code generation process. This involves the extraction of model properties and its subsequent translation to SPARK Ada code.

From the execution of the user interface the code generation uses appropriate API’s to access RODIN’s statically checked internal databases. The activation of the user interface initiates a search for model properties by iteratively seeking out the roots of past refinements and also context components. This as discussed in Section 4.3.1 of the Design chapter can be found by using information attained from the model’s “REFINES” and “SEES” definitions respectively.

The roots of both the context and machine components are acquired by this process. This allows the code generator ESpark, to gain of further access to the model properties from implementing these roots. The extraction of the model properties is divided into two extraction phases, namely context and machine. The properties extracted are stored appropriately for use in the translation phases. The machine and context translation phases implement the design approaches discussed in Section 4.4. The translations of the context properties are used in defining the global properties of the program, such as global type declarations and definitions.
The translation process allows global types to be reused by definitions found in the machine translation. The context properties also generates the global, own and pre SPARK annotations. The machine translation defines the variables extracted as formal parameters of an Ada procedure. Note, types defined by the context translations can be reused by the variable translations. Merging rules are applied to the events extracted from the machine component to form monolithic representation of the system’s operation (see Section 4.4.5). This sequential execution of the system is defined within the Ada procedure. The properties translated from both of these phases and stored into appropriate translation blocks, which are then substituted by the final build phase which compiles the translations to appropriate Ada files. The SPARK annotations are

The final ESpark plug-in consists of three eclipse plug-in projects, with one main eclipse plug-in executing the code generation process, an eclipse feature project that provides branding information and licensing details, and an eclipse site project that generates appropriate .jar files for the feature project and the main plug-in project. The jar files built are required to install the plug-in into the RODIN toolset, which is a fairly simple and straightforward process with no additional components dependencies apart from the base RODIN platform. After installation, a GUI button will be present on any machine component created in the RODIN toolset. Translation is performed automatically with no required user input. At this stage of the prototype, error conditions are not displayed to the user. This is discussed further in future work.

5.3 ESpark: Implementation of Minimum Search Problem

5.3.1 Implementation

As the name of code generators ESpark suggests code generation of Event-B models to SPARK Ada code, an implementation of a specification Event-B model Minimum Search is taken in order test the code generation process. This example was used in the previous chapter in describing the design of the code generation system. The specification model is designed using the RODIN toolset, using its modelling tools of abstraction and refinement. The model reaches its final level of abstraction after three refinements of its machine component. At this stage the model describes the full operation of the system as a collection of individual events.

Once the final refinement of the model is reached, the ESpark code generation is activated from the machine component in the RODIN toolset. The code generation produces an Ada body and specification file. The specification as seen in Table x consists of the SPARK annotations describing the interface of the procedure that executes the Minimum Search program. The body file consists of the procedure that executes the operations of the minimum search program as seen in Table 5.1 and 5.2. ESpark was successful in its implementation on to the
RODIN toolset and its code generation process was fully automatic in producing the AdaSPARK code.

---

**Minimum Search – Generated Ada Specification File**

```ada
package Search_Minimum
--# own f;
is
    --Note To Programmer: Please reassign desired value for constant n.
    n: Constant:= 1000;
    Type f_Type is range 1 .. n;
    Type f_Array_Type is array (f_Type) of Natural;
    f: f_Array_Type;

    procedure Search_Minimum_Procedure(x: out f_Type; minimum: out Natural);
    --# global in f;
    --# derives x from f & minimum from f;
    --# pre n >= 1;
    --# post x = n and (For all i in f_Type range 1..n-1 => (f(i)>=minimum));
end Search_Minimum;
```

**Table 5.1: Minimum Search – Ada Specification File**

---

**Minimum Search – Generated Ada Body File**

```ada
package body Search_Minimum is
    procedure Search_Minimum_Procedure(x: out f_Type; minimum: out Natural)
    is
        begin
            x := 1;
            minimum := f(n);

            while x/= n loop
                --# assert (For all i in f_Type range 1..x-1 => (f(i)>=minimum)) and x/= n;

                if minimum <= f(x) then
                    x := x + 1;
                elsif minimum > f(x) then
                    minimum := f(x);
                    x := x + 1;
                end if;

            end loop;

        end Search_Minimum_Procedure;
end Search_Minimum;
```

**Table 5.2: Minimum Search – Ada Body File**
5.3.2 Evaluation

The use of ESpark on the minimum search example was successful in automatically generating SPARK Ada code. In order to evaluate the generated code, the SPARK Examiner was used to perform both static and dynamic analysis. The flow analysis of the procedure `Search_Minimum_Procedure` found no errors. The generated code was in conformance with the program language, and the sequential program constructs were generated automatically by the application of the merging rules and the identification of convergence level. In the declaration of global properties in the Ada specification, the code generation applies a default value to a constant declared (Ada requires all constants to be initialised on declaration). This is accompanied by an Ada comment that informs the user to change the value of the constant to a desired value, as seen in Table 5.1. Also, global types declared are reused to define formal parameters of the same type in Ada procedure. This is beneficial as Ada language is strict on the use of types and the use of different types would result in Ada type errors. The code generation also makes useful indentations to allow the programmer to read the code easily.

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>assert @ 10</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>assert @ 10</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>assert @ 10</td>
<td>Contradiction</td>
<td>No DPC</td>
<td>X-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>rtc check @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>rtc check @ 12</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>rtc check @ 13</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>rtc check @ 14</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>rtc check @ 15</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>12</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

Table 5.3: Verification Conditions for Minimum Search

For the dynamic analysis, the verification conditions were generated by the SPARK Examiner. Table 5.3 states the number of VCs generated from the procedure `search_minimum_procedure`. These verification conditions were put through the SPARK Simplifier (Proof Checker) to discharge the VCs. The POGS report as seen in Table 5.4 provides a summary of the number of VCs discharged and by what type. 87% of the VCs (13) generated was discharged by the simplifier, the remaining 13% was undercharged. The un-discharged VC (2), were informally hand proven to be true. This indicated that a stronger Theorem Prover could be used to discharge the remaining VCs.
Total VCs by type: | Proved By Or Using |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Examnr</td>
<td>Simp (User)</td>
</tr>
<tr>
<td>Assert/Post</td>
<td>8</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>7</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
</tr>
</tbody>
</table>

Totals: 15 0 13 0 0 0 0 2

%Totals: 0% 87% 0% 0% 0% 0% 13%

Table 5.4: Verification Conditions for Minimum Search

From this evaluation, the development of the Minimum Search problem has verification at both the design level and the implementation (code) level. The model in Event-B has its Proof Obligations discharged by the RODIN prover. The code generation makes use of the verified properties of the model and produces SPARK proof annotations to the code in order to generate verification conditions, which are discharged by a proof checker. This concludes the evaluation of this particular example. The project has looked into code generation of eight other examples which is discussed in the next Chapter 6, evaluation and conclusion.
Chapter 6

Evaluation & Conclusion

6.1 Summary of Strengths and Achievements

**FR3** - *The automatic code generator should be integrated as far as possible with the RODIN toolset*

The aim of ESpark was to perform automatic code generation with respect to an Event-B model, where the generated code (Ada and SPARK) was probably correct. The approach taken was to extend the functionality of Event-B’s toolkit RODIN, by implementing a third party plug-in for automatic code generation. ESpark has been successful in its development and implementation on to RODIN. Information on how to install and use ESpark can be found in Appendix A. Once its user interface found on the RODIN tool-kit is activated, the process of extracting and translating model properties for code generation is fully automatic. The generated program files can be found within the project workspace, allowing the generated code to be easily accessed and evaluated by the user. The RODIN plug-in ESpark can be easily implemented on RODIN, and requires no additional component dependencies. This satisfied functional requirement FR3.

**FR1** - *The automatic code generation will focus on Event-B models which give rise to sequential systems, including those that are concerned with iterative and conditional constructs.*

For the purpose of evaluating the ESpark automatic code generator, ten example sequential systems were modelled in Event-B’s RODIN tool kit and passed through the ESpark
code generation plug-in. The Event-B specification of these models can be found in Appendix B. The examples modelled in Event-B targeted sequential systems of different categories such as arithmetic operations, searching problems and sorting algorithms. As discussed in Section 3.2, models in Event-B lack knowledge of its execution. It was an important design element of the auto code generator to support and transform events modelled in Event-B to a monolithic program structure in Ada code. This was accomplished by identifying the convergence level of each event from the model properties and applying appropriate merging rules. The result was the automatic code generation of programs composed of multiple and even nested program constructs. Examples of the code generated can be found in Appendix C. This was an important functional requirement that was essential in order for the auto code generator to produce meaningful program constructs. This strongly satisfies the functional requirement FR2.

These examples as seen in Appendix A are composed of arithmetic, search and sorting algorithms, and are intended to test the code generator in terms of producing different sequential program constructs, and also to test the translation of Event-B notations in producing type definitions and more importantly SPARK annotations. The system has been successful in generating Ada code with SPARK annotations for all ten examples. The verifiability of these program files have been evaluated by the use of the SPARK Examiner.

FR2 - The automatic code generator will use both the static and dynamic aspects of the Event-B model to generate loop invariants, pre-conditions and post conditions.

The generated Ada files with SPARK annotations can be fed through the SPARK examiner to test its validity in terms of information flow analysis, data flow analysis and mathematical verification of proofs. An important element of the project was to use static and dynamic properties found within the design of Event-B models to produce SPARK proof annotations. These annotations namely pre-conditions, post-conditions and loop assertions were required by the SPARK Examiner in order to produce verification conditions VCs. The code generation was partially successful in producing these annotations. Problems were faced with implementing more complex mathematical notations describing model properties onto the SPARK syntax. However, SPARK proof annotation did support the translation of logical quantifiers and simple predicates onto its proof annotations, which sufficed the SPARK Examiner in generating verification condition of the code generated. Also, as discussed in the implementation of the Minimum Search model in Chapter 5, the static property of the model were substituted within the loop invariant to produce the post condition for SPARK’s proof annotation. This was successfully passed through the SPARK Examiner to produce verification conditions. The examiner’s result can be found in Appendix D of Listing D.7. In contrast to this use, some properties of the system such as variants have not been investigated, due to the given time scale of
this project. This is discussed further in the section related to future work. This investigation in to model properties has to an extent satisfied the requirement FR2.

**ER1 - The SPARK Simplifier should be used to discharge the VCs that arise from the auto generation code.**

The SPARK Simplifier has been used to discharge verification conditions generated by the Examiner, as seen in Appendix D for the nine example problems. Some verification conditions were not discharged. It was understood that a more powerful theorem prover was required in order to do so. However, the models designed in Event-B’s RODIN toolset, have had the Proof Obligations discharged by the RODIN’s prover. This ensured that the properties of the system have been proved before automatic code generation of the Event-B models.

**ER2 - Evaluation set should include all constructs supported by the auto coder**

Automatic code generation targeted different categories of sequential programs, such as arithmetic operations, search problems and sorting algorithms. The generated code SPARK Ada code for the nine examples can be found in Appendix C. As can be seen, different sequential programs constructs have been code generated successfully. This has been possible by applying information of the convergence level found in the model’s machine component and its application to the merging rules for sequential programs. This has successfully met the evaluation requirement ER2.

**ER3 - Any undercharged VCs should be investigated.**

The investigation into all un-discharged VCs was not possible within the given time frame. The SPARK Simplifier was not powerful enough to prove all VCs generated. The application of a more powerful theorem prover could have aided this evaluation.
6.2 Limitations

At present, ESpark performs automatic code generation from a specification Event-B model to code generation of sequential programs. This limits itself from code generation of other systems such as concurrent systems or distributed programs. It was seen not viable to target more than one system for the given time scale of this project. This limitation is further discussed in the future work. Also Event-B notations can vary in complexity of its use during design of a specification model. There are limitations to the ESpark auto code generator in covering only a subset of Event-B’s notations. One of the reasons for this was the syntax of SPARK in some cases could not hold the transformations of Event-B’s notations to program code. However, an area of SPARK not covered by this project was the use of SPARK functions, which could help represent more complex Event-B notations.

Almost all Event-B model properties were used by ESpark in the automatic code generation process. The one property that was not used was the variant property, which is a quantity that represents the possible number of an event or operation of the system would repeat itself. It can be used to prove that the process will terminate, i.e. that process will not run indefinitely.

6.3 Future Work

6.3.1 Short-term

Traceability within code generation can be beneficial to the user in identifying the automatically generated code from the part of the model it is derived from. This form of static traceability would allow the user to inspect the code generated and easily understand its origin and functionality in regards to the source model. This is highly advantageous with a system like Event-B, were reasoning is a key element when modelling a system to have some form of translation in the development of the system to be part of the code generated. This can be viewed possible as a short term goal, for future work on the current code generator.

The interface of the code generator can be further improved by adding additional information of the code generation process within the RODIN tool in itself. More information could be provided on errors that could arise during the process of code generation. A documentation of the types of errors such could help in providing feedback to the user in any necessary requirements to the model for code generation.

The process of automatic code generation in ESPark targets the final refinement of the model in producing the source code. This provides the final representation of the system’s operation and functionality in generated code. However, work can be taken into iteratively
activating code generation to also previous refinements of the system to produce operational code at each stage of the model design. This could be beneficial in allowing the user to view the system’s functionality and understand its operation at each of stage of its previous refinements.

6.3.2 Long-term

The automatic code generation currently targets sequential systems. It is known that Event-B is also capable of modelling discrete systems of a wide range of domains such as concurrent programs, distributed programs, sequential programs, electronic circuits, reactive systems. In this concurrent programs also provide an interesting domain for automatic code generation to be targeted.

ESpark specifically aims in the generation of Ada and SPARK code. However, other languages such as ESC/Java, Spec#, etc that program analysis has not been looked into for targeting the code generation. A significant contribution to future work on the auto code generator would be to introduce an intermediate language for code generation, from which other language platforms can be targeted.

More work can be taken into examining the transformation of Event-B notations to program code, in order to expand the Event-B notations supported by ESpark. By increasing the current subset of Event-B properties, the automatic code generation can represent more complexities in system design. Also, the termination proofs at code level could be investigated by the use of variant property found in the Event-B model. This is left for future work of the ESpark code generation.

6.4 Conclusion

The investigation of the project has led into the research and development of ESpark automatic code generation for RODIN, which given an Event-B specification model would produce a low-level representation i.e. program code. The focus here was placed on sequential subset of Event-B. ESpark was tested successfully on examples drawn from literature in automatically generating SPARK Ada program code. Also, the main emphasis of the project laid in the extraction of SPARK proof annotation from the model properties, i.e. pre, post and loop assertion from the invariants and guards associated with the Event-B models. The system was successful in this requirement allowing the SPARK Examiner to produce Verification Conditions for the code, which could be discharged by the SPARK Simplifier. This ensured that the code generated could be verified at the implementation level.

There were some verification conditions that were not proved by the SPARK Simplifier. This problem requires further investigation; a possible solution could be the requirement for
stronger theorem prover to discharge the remaining VCs. However, it is important to note that Event-B models from which the code generation is derived from are designed in RODIN, were Proof Obligations are discharged by the tool’s prover.

Aside from meeting the preliminary requirement of the investigation, ESpark was successfully integrated, as a plug-in on to the RODIN tool kit. The tool is known for its use in both industry and academia, which could allow the plug-in to be further developed and upgraded onto newer versions. Future development of the auto coder could delve into new domains such as concurrent systems and could also look into targeting other program languages such as C and Java.
Bibliography


Diller, A. Z: An introduction to formal methods, oucl.


Appendix A

ESpark RODIN Plug-in

A.1 ESpark - Installation Guide

This section provides information on how the ESpark RODIN plug-in can be installed onto the RODIN toolset. Prior to installation, the RODIN toolset is required to be installed. Once this is complete, the ESpark plug-in can be installed by selecting Menu ➔ Help ➔ Install New Software, which can be found on the menu bar.

![Fig A.1: Screenshot of RODIN interface](image.png)

This opens the Install menu of RODIN, which allows third-party plug-ins to be installed on to the RODIN toolset. In order to install ESpark, a repository is required for RODIN to search for available plug-ins. This is done by selecting the Add button and then selection Local on the Add Repository window.
The repository looks for jar files to be installed. The ESpark plug-in installation jar files can be found by selecting “hw.rm339.eventb.ESpark.site”. The java site project is where the code generator builds into. Once this folder is selected, the repository searches for the installation components for the plug-in.
Fig A.3: Screenshot of Install Window

It is important to ensure that the following options within the Additional Software window to be unselected:

- Show only the latest version of available software.
- Group items by category.
- Contact all update sites during install to find required software.

At this stage, the ESpark plug-in and version should appear in the available software. Once this is selected, the Next button should be selected, which brings up the installation details. To finish installation, the click button is selected. The user must restart RODIN for the software to function correctly. This concludes the guide to installing the ESpark plug-in on the RODIN. The next chapter provides information on how to use the plug-in.
Fig A.2: Screenshot of Plug-in Installation Details
A.2 ESpark – How to Activate Auto Code Generator

Once the ESpark plug-in has been installed, the use of the automatic code generator is easy to be activated by the user. First the Event-B project that requires the code generation should be selected. In this, the final refinement that is the last machine component is required to be selected. This opens the machine component window and makes the machine editor’s tool bar to appear. On this the ESpark interface button is present. When the final refinement is completed, the ESpark button can be activated, which performs the code generation automatically.

![Fig A.2: Screenshot of Java Site Project to Select](image)

The files produced can be found in the workspace of the project, within the folder Code Generated. The files use can be later verified by the SPARK Examiner.
**Fig A.2: Screenshot of Java Site Project to Select**
Appendix B

Event-B Specification

B.1 Division Algorithm

Listing B.1: Division Algorithm – Context

```
CONTEXT
   Division_Context0
CONSTANTS
   x
   M
   N
AXIOMS
   axm1 : x ∈ ℕ
   axm2 : M ∈ 0‥x
   axm3 : N ∈ 0‥x
   axm4 : M ≥ 0
   axm5 : N > 0
END

Listing B.1: Division Algorithm – Machine Refinement 1

MACHINE
   Division_Machine0
SEES
   Division_Context0
VARIABLES
   Q
   R
INVARINTANS
   inv1 : Q ∈ 0‥x
   inv2 : R ∈ 0‥x
EVENTS
   INITIALISATION
      STATUS
      ordinary
      BEGIN
         act1 : Q := 0
         act2 : R := M
      END
      final
      STATUS
      ordinary
      WHEN
         grd1 : R ≥ N
      THEN
         skip
      END
      progress
      STATUS
```
ANTICIPATED
BEGIN
  act1 : R ∈ N
END
END

Listing B.1: Division Algorithm – Machine Refinement 2

MACHINE
  Division_Machine1
REFINES
  Division_Machine0
SEES
  Division_Context0
VARIABLES
  Q
  R
INVARaints
  inv1 : M = Q ⋅ N + R
  inv2 : R < N
  inv3 : R ≥ 0
VARIANT
  R − N
EVENTS
  INITIALISATION
    extended
    STATUS
    ordinary
BEGIN
  act1 : Q = 0
  act2 : R = M
END
final
  extended
  STATUS
  ordinary
REFINES
final
WHEN
  grd1 : R ≥ N
THEN
  skip
END
progress
  STATUS
  convergent
REFINES
progress
WHEN
  grd1 : R < N
THEN
  act1 : Q := Q + 1
  act2 : R := R − N
END
END
B.2 Sorted Array Algorithm

Listing B.2: Sorted Array Algorithm – Context

CONTEXT
   Search_Context1

CONSTANTS
   n
   f
   v

AXIOMS
   axm1 : n ∈ ℕ
   axm2 : f ∈ 1‥n → ℕ
   axm3 : v ∈ ran(f)
   axm4 : n ≥ 1

END

Listing B.2: Sorted Array Algorithm – Machine Refinement 1

MACHINE
   Search_Machine1
SEES
   Search_Context1

VARIABLES
   r

INVARIANTS
   inv1 : r ∈ 1‥n

EVENTS
   INITIALISATION ▲
   STATUS ordinary
   BEGIN
      act1 : r ∈ ℕ
   END
   final ▲
   STATUS ordinary
   WHEN
      grd1 : r ∈ 1‥n
      grd2 : f(r) = v
   THEN
      skip
   END
   progress ▲
   STATUS anticipated
   BEGIN
      act1 : r ∈ ℕ
   END

END

Listing B.2: Sorted Array Algorithm – Machine Refinement 2

MACHINE
   Search_Machine2
REFINES
   Search_Machine1
SEES
Search_Context1

VARIABLES
  r
  p
  q

INVARIANTS
  inv1 : p ∈ 1‥n
  inv2 : q ∈ 1‥n
  inv3 : r ∈ p‥q
  inv4 : v ∈ f[p‥q]

VARIANT
  q − p

EVENTS
  INITIALISATION ▲

  STATUS
  ordinary

  BEGIN
    act1 : r ∈ 1‥n
    act2 : p := 1
    act3 : q := 1
  END

  final ▲
    extended

  STATUS
  ordinary

  REFINES
  final

  WHEN
    grd1 : r ∈ 1‥n
    grd2 : f(r) = v
  THEN
    skip
  END

  inc ▲
    STATUS
    convergent

    REFINES
    progress

    WHEN
      grd1 : f(r) < v
    THEN
      act1 : p := r + 1
      act2 : r ∈ r+1‥q
    END

  dec ▲
    STATUS
    convergent

    REFINES
    progress

    WHEN
      grd1 : f(r) > v
    THEN
      act1 : q := r − 1
      act2 : r ∈ p‥r−1
    END

END
Listing B.2: Sorted Array Algorithm – Machine Refinement 3

MACHINE
  Search_Machine3
REFINES
  Search_Machine2
SEES
  Search_Context1
VARIABLES
  r
  p
  q
EVENTS
  \headINITIALISATION\triangle
  \head STATUS
  ordinary
BEGIN
  \head act1\triangleright
  \triangleq r \equiv \frac{(1+n)-2}{2}
  \head act2\triangleright
  \triangleq p \equiv 1
  \head act3\triangleright
  \triangleq q \equiv 1
END
\head final\triangleright
\head STATUS
  ordinary
REFINES
  final
WHEN
  \head grd2\triangleright
  \triangleq f(r) = v
THEN
  \head skip
END
\head inc\triangleright
\head STATUS
  ordinary
REFINES
  inc
WHEN
  \head grd2\triangleright
  \triangleq f(r) \neq v
  \head grd1\triangleright
  \triangleq f(r) < v
THEN
  \head act1\triangleright
  \triangleq p \equiv r + 1
  \head act2\triangleright
  \triangleq r \equiv \frac{(r+1+q)-2}{2}
END
\head dec\triangleright
\head STATUS
  ordinary
REFINES
  dec
WHEN
  \head grd2\triangleright
  \triangleq f(r) \neq v
  \head grd1\triangleright
  \triangleq f(r) > v
THEN
  \head act1\triangleright
  \triangleq q \equiv r - 1
  \head act2\triangleright
  \triangleq r \equiv \frac{(p+r-1)-2}{2}
END
END
B.3 Filter Program

Listing B.3: Filter Program – Context

CONTEXT
    Filter_Context1
CONSTANTS
    n
    DataSet
AXIOMS
    axm1 : n ∈ ℕ
    axm2 : DataSet ∈ 1‥n → ℕ
    axm3 : n ≥ 1
END

Listing B.3: Filter Program – Machine Refinement 1

MACHINE
    Filter_Machine1
SEES
    Filter_Context1
VARIABLES
    x
INVARIANTS
    inv1 : x ∈ 1‥n
    inv2 : x ≤ n
EVENTS
    INITIALISATION
        STATUS ordinary
        BEGIN
            act1 : x := 1
        END
final
    STATUS ordinary
    WHEN
        grd1 : x = n
    THEN
        skip
END
    progress
        STATUS anticipated
        BEGIN
            skip
        END
END
END
Listing B.3: Filter Program – Machine Refinement 2

MACHINE 
  Filter_Machine2 
REFINES 
  Filter_Machine1 
SEES 
  Filter_Context1 
VARIABLES 
  x 
  result 
INVARIANTS 
  inv1 : result ∈ ℕ 
  inv2 : result ≥ 0 
  inv3 : result ≤ x * 100 
VARIANT 
  n = x 
EVENTS 
  INITIALISATION \[\triangleright\] 
    extended 
    STATUS 
    ordinary 
BEGIN 
  act1 : x := 1 
  act2 : result := 0 
END 
final \[\triangleright\] 
  extended 
  STATUS 
  ordinary 
REFINES 
final 
WHEN 
  grd1 : x = n 
THEN 
  skip 
END 
Filter_Process \[\triangleright\] 
  extended 
  STATUS 
  convergent 
REFINES 
progress 
WHEN 
  grd1 : x < n 
  grd2 : DataSet(x) > 1 
  grd3 : DataSet(x) ≤ 100 
THEN 
  act1 : result := result + DataSet(x) 
END 
END 

B.4 Simple Sorting Algorithm

Listing B.4: Simple Sorting Algorithm – Context 1

CONTEXT 
  Sort_Context1
CONSTANTS
n
f

AXIOMS
axm1 : n ∈ N
axm2 : f ∈ 1‥n → N

END

Listing B.4: Simple Sorting Algorithm – Machine Refinement 1

MACHINE
Sort_Machine1
SEES
Sort_Context1
VARIABLES
g

INVARIANTS
inv1 : g ∈ N ↔ N

EVENTS
INITIALISATION △
STATUS
ordinary
BEGIN
act1 : g :∈ N ↔ N
END
final △
STATUS
ordinary
WHEN
grd1 : g ∈ 1‥n → N
grd2 : ran(g) = ran(f)
    ∀i,j∈1‥n−1
grd3 : ∧ j∈i+1‥n
    ⇒
g(i) < g(j)
THEN
skip
END
progress △
STATUS
anticipated
BEGIN
act1 : g :∈ N ↔ N
END
END

Listing B.4: Simple Sorting Algorithm – Machine Refinement 2

MACHINE
Sort_Machine2
REFINES
Sort_Machine1
SEES
Sort_Context1
VARIABLES
g
k
l

INVARIANTS
inv1 : \( g \in 1 \cdot n \rightarrow \mathbb{N} \)
inv2 : \( \text{ran}(g) = \text{ran}(f) \)
inv3 : \( k \in 1 \cdot n \)
inv4 : \( l \in \mathbb{N} \)

**VARIANT**
\( n = k \)

**EVENTS**

**INITIALISATION**

**STATUS**
ordinary

**BEGIN**
act1 : \( g := f \)
act2 : \( k := 1 \)
act3 : \( l \in \mathbb{N} \)

**END**

**final**

**STATUS**
ordinary

**REFINES**
final

**WHEN**

**grd1** : \( g \in 1 \cdot n \rightarrow \mathbb{N} \)
**grd2** : \( \text{ran}(g) = \text{ran}(f) \)
\( \forall i, j \in 1 \cdot n-1 \)
**grd3** : \( \land j \in i+1 \cdot n \)
\( \Rightarrow g(i) < g(j) \)

**THEN**

**skip**

**END**

**progress**

**STATUS**
convergent

**REFINES**
progress

**WHEN**

**grd1** : \( k \neq n \)
**grd2** : \( l \in k \cdot n \)
**grd3** : \( g(l) = \min(g[k \cdot n]) \)

**THEN**

act1 : \( g := g \ { k \mapsto g(l) } \)
act2 : \( k := k+1 \)
act3 : \( l \in \mathbb{N} \)

**END**

**prog**

**STATUS**
anticipated

**BEGIN**
act1 : \( l \in \mathbb{N} \)

**END**

**END**

Listing B.4: Simple Sorting Algorithm – Machine Refinement 3

**MACHINE**
Sort_Machine3

**REFINES**
Sort_Machine2
SEES
Sort_Context1

VARIABLES
\( g \)
\( k \)
\( l \)
\( j \)

INVARINTS
\( \text{inv1} : j \in k \cdots n \)
\( \text{inv2} : l \in k \cdots j \)
\( \text{inv3} : g(l) = \min(g[k \cdots j]) \)

VARIANT
\( n - j \)

EVENTS
INITIALISATION

STATUS
ordinary
BEGIN
\( \text{act1} : g := f \)
\( \text{act2} : k := 1 \)
\( \text{act3} : l := 1 \)
\( \text{act4} : j := 1 \)
END

final

STATUS
ordinary

REFINES
final

WHEN
\( \text{grd1} : k = n \)
THEN
skip
END

progress

STATUS
ordinary

REFINES
progress

WHEN
\( \text{grd1} : k \neq n \)
\( \text{grd2} : j = n \)
THEN
\( \text{act1} : g := g \{ k \mapsto g(l) \} \)
\( \text{act2} : k := k + 1 \)
\( \text{act3} : l := k + 1 \)
\( \text{act4} : j := k + 1 \)
END

prog1

STATUS
convergent

REFINES
prog

WHEN
\( \text{grd1} : k \neq n \)
\( \text{grd2} : j \neq n \)
\( \text{grd3} : g(l) \leq g(j + 1) \)
THEN
\( \text{act1} : j := j + 1 \)
END

prog2
convergent

REFINES prog

WHEN
  grd1 : k ≠ n
  grd2 : j ≠ n
  grd3 : g(l) > g(j + 1)

THEN
  act1 : j := j + 1
  act2 : l := j + 1

END

END

B.5 Summation Program

Listing B.5: Summation Program – Context 1

CONTEXT
  Sum_Con0

CONSTANTS
  n
  f

AXIOMS
  axm1 : n ∈ \mathbb{N}
  axm2 : f ∈ 1 \cdots n \to \mathbb{N}

END

Listing B.5: Summation Program – Machine Refinement 1

MACHINE
  Sum_Mac0

SEES
  Sum_Con0

VARIABLES
  k
  total

INVARIANTS
  inv1 : k ∈ 1 \cdots n
  inv2 : total ∈ \mathbb{N}

EVENTS
  INITIALISATION △
    STATUS ordinary
    BEGIN
      act1 : k := 1
      act2 : total := 0
    END

    final △
    STATUS ordinary
    WHEN
      grd1 : k = n
      THEN
        skip
      END

    Summing △
    STATUS anticipated
    BEGIN
      act1 : total \in \mathbb{N}
    END
Listing B.5: Summation Program – Machine Refinement 2

MACHINE
Sum_Mac1
REFINES
Sum_Mac0
SEES
Sum_Con0
VARIABLES
k
total
VARIANT
n-k
EVENTS
INITIALISATION
\[\triangleq\]
STATUS
ordinary
BEGIN
act1 : k := 1
act2 : total := 0
END

END

B.6 Least Common Multiple

Listing B.6: Least Common Multiple – Context 1

CONTEXT
LCM_context1
CONSTANTS
n A B
AXIOMS
axm1 : n ∈ \mathbb{N}
Listing B.6: Least Common Multiple – Machine Refinement 1

MACHINE LCM_machine1
SEES LCM_context1
VARIABLES i j
INVARIANTS
  inv1 : i ∈ 1 .. n
  inv2 : j ∈ 1 .. n
  inv3 : i ≤ B
  inv4 : j ≤ A
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      act1 : i := 1
      j := 1
    END
  final
    STATUS ordinary
    WHEN
      grd1 : (A*i) = (B*j)
    THEN
      skip
    END
  progress
    STATUS anticipated
    BEGIN
      skip
    END
END

Listing B.6: Least Common Multiple – Machine Refinement 2

MACHINE LCM_machine2
REFINES LCM_machine1
SEES LCM_context1
VARIABLES i j lcm_result
INVARIANTS
  inv2 : lcm_result ∈ 1 .. n
VARIANT A − j
EVENTS
  INITIALISATION extended
    STATUS
ordinary
BEGIN
act1 : j := 1
act2 : j := 1
act3 : lcm_result := 1
END

final $\triangle$ extended
STATUS
ordinary
REFINES
final
WHEN
  grd1 : (A*i) = (B*j)
THEN
  skip
END

prog_1 $\triangle$
STATUS
convergent
REFINES
progress
WHEN
  grd1 : (A*i) ≠ (B*j)
  grd2 : (A*i) < (B*j)
THEN
  act1 : i := i+1
  act2 : lcm_result := i*j
END

prog_2 $\triangle$
STATUS
convergent
REFINES
progress
WHEN
  grd1 : (A*i) ≠ (B*j)
  grd2 : (A*i) > (B*j)
THEN
  act1 : j := j + 1
  act2 : lcm_result := i*j
END
END

B.7 Minimum Search

Listing B.7: Minimum Search – Context 1

CONTEXT
Minimum_Context1

CONSTANTS
n
f

AXIOMS
  axm1 : n ∈ N
  axm2 : f ∈ 1 .. n → N
  axm3 : n ≥ 1
END
Listing B.7: Minimum Search – Machine Refinement 1

MACHINE
Minimum_Machine1
SEES
Minimum_Context1
VARIABLES
x
INvariants
inv1 : x ∈ 1‥n
EVENTS
INITIALISATION
STATUS
ordinary
BEGIN
act1 : x := 1
END
final
STATUS
ordinary
WHEN
grd1 : x = n
THEN
skip
END
progress
STATUS
anticipated
BEGIN
act1 : x ∈ N
END
END

Listing B.7: Minimum Search – Machine Refinement 2

MACHINE
Minimum_Machine2
REFINES
Minimum_Machine1
SEES
Minimum_Context1
VARIABLES
x
VARIANT
n - x
EVENTS
INITIALISATION
extended
STATUS
ordinary
BEGIN
act1 : x := 1
END
final
extended
STATUS
ordinary
REFINES
final
WHEN
\texttt{grd1} \rightarrow x = n \\
\texttt{THEN} \\
\texttt{skip} \\
\texttt{END} \\
\texttt{progress} \triangleright \\
\texttt{STATUS} \convergent \\
\texttt{REFINES} \\
\texttt{progress} \\
\texttt{WHEN} \\
\texttt{grd1} \rightarrow x \neq n \\
\texttt{THEN} \\
\texttt{act2} \rightarrow x := x + 1 \\
\texttt{END} \\
\texttt{END} \\

\textbf{Listing B.7: Minimum Search – Machine Refinement 3}

\textbf{MACHINE} \\
Minimum\_Machine3 \\
\textbf{REFINES} \\
Minimum\_Machine2 \\
\textbf{SEES} \\
Minimum\_Context1 \\
\textbf{VARIABLES} \\
x \\
minimum \\
\textbf{INVARINTANTS} \\
inv1 \rightarrow minimum \in \mathbb{N} \\
inv2 \rightarrow \forall i \in 1 \cdots x-1 \Rightarrow f(i) \geq minimum \\
\textbf{EVENTS} \\
\texttt{INITIALISATION} \triangleright \\
\texttt{STATUS} \\
\texttt{ordinary} \\
\texttt{BEGIN} \\
\texttt{act1} \rightarrow x := 1 \\
\texttt{act2} \rightarrow minimum := f(n) \\
\texttt{END} \\
\texttt{final} \triangleright \\
\texttt{extended} \\
\texttt{STATUS} \\
\texttt{ordinary} \\
\textbf{REFINES} \\
\texttt{final} \\
\texttt{WHEN} \\
\texttt{grd1} \rightarrow x = n \\
\texttt{THEN} \\
\texttt{skip} \\
\texttt{END} \\
\texttt{progress\_1} \triangleright \\
\texttt{STATUS} \\
\texttt{ordinary} \\
\textbf{REFINES} \\
\texttt{progress} \\
\texttt{WHEN} \\
\texttt{grd1} \rightarrow x \neq n \\
\texttt{grd2} \rightarrow minimum \leq f(x) \\
\texttt{THEN} \\
\texttt{act2} \rightarrow x := x + 1 \\
\texttt{END} \\
\texttt{progress\_2} \triangleright \\
\texttt{STATUS} \\
\texttt{ordinary}
REFINES

WHEN

  grd1 : x ≠ n
  grd2 : minimum > f(x)

THEN

  act1 : minimum := f(x)
  act2 : x := x + 1

END

END

B.8 Power Evaluation Program

Listing B.8: Power Evaluation Program – Context 1

CONTEXT
  PowerEvaluation_Context

CONSTANTS
  n
  m
  x

AXIOMS
  axm1 : n ∈ ℕ
  axm2 : m ∈ 1‥n
  axm3 : x ∈ ℕ

END

Listing B.8: Power Evaluation Program – Machine Refinement 1

MACHINE
  PowerEvaluation_M1

SEES
  PowerEvaluation_Context

VARIABLES
  k

INVARIANTS
  inv1 : k ∈ 1‥n

EVENTS
  INITIALISATION
    STATUS
    ordinary
    BEGIN
      act1 : k := m
    END

  final
    STATUS
    ordinary
    WHEN
      grd1 : k ≠ 0
    THEN
      skip
    END

  progress
    STATUS
    anticipated
    BEGIN
      act1 : k ∈ 1‥n
    END

END

END
Listing B.8: Power Evaluation Program – Machine Refinement 2

MACHINE
PowerEvaluation_M2
REFINES
PowerEvaluation_M1
SEES
PowerEvaluation_Context
VARIABLES
k
y
INVARIANTS
inv1 : y ∈ N
inv2 : y ≥ 1
VARIANT
k
EVENTS
INITIALISATION
extended
STATUS
ordinary
BEGIN
act1 : k := m
act2 : y := 1
END
final
extended
STATUS
ordinary
REFINES
final
WHEN
grd1 : k ≠ 0
THEN
skip
END
progress
extended
STATUS
convergent
REFINES
progress
WHEN
grd1 : k ≠ 0
THEN
act1 : k := k − 1
act2 : y := y + x
END
END

B.9 Polynomial Evaluation Model

Listing B.9: Polynomial Evaluation Model – Context 1

CONTEXT
Poly_Cont0
CONSTANTS
n
f
x
AXIOMS

axm1 : n ∈ ℕ
axm2 : f ∈ 1 · · · n → ℕ
axm3 : x ∈ ℕ

END

Listing B.9: Polynomial Evaluation Model – Machine Refinement 1

MACHINE Poly_Mach0
SEES Poly_Cont0
VARIABLES k

INVARIANTS
inv1 : k ∈ 1 · · · n
inv2 : total ∈ ℕ

EVENTS
INITIALISATION
STATUS ordinary
BEGIN
act1 : k := n
act2 : total := 0
END

final
STATUS ordinary
WHEN
grd1 : k = 1
THEN
skip
END
Poly
STATUS anticipated
BEGIN
skip
END

Listing B.9: Polynomial Evaluation Model – Machine Refinement 2

MACHINE Poly_Mach1
REFINES Poly_Mach0
SEES Poly_Cont0
VARIABLES k

VARIANT k

EVENTS
INITIALISATION extended
STATUS ordinary
BEGIN
act1 : k := n
act2 : total := 0

83
final \triangleq extended
STATUS ordinary
REFINES final
WHEN 
  grd1 : k = 1
THEN
  skip
END
Poly \triangleq STATUS convergent
REFINES Poly
WHEN 
  grd1 : k \neq 1
THEN
  act1 : k := k - 1
  act2 : total := total + x + f(k)
END
END
Appendix C

Generated Code

C.1 Division Algorithm

Listing C.1: Ada Specification – Division Algorithm

package DivQuo
--# own M, N;
is
--Note To Programmer: Please reassign desired value for constant x.
x: Constant:= 1000;

Type M_Type is range 0 .. x;
M: M_Type;
N: M_Type;

procedure DivQuo_Procedure(Q: out M_Type; R: out M_Type);
--# global in M, N;
--# derives Q from M, N & R from M, N;
--# pre M >= 0;
--# post M = Q * N + R and R >= 0 and R>=N;
end DivQuo;

Listing C.1: Ada Body – Division Algorithm

package body DivQuo is
procedure DivQuo_Procedure(Q: out M_Type; R: out M_Type)
is
begin
Q := 0;
R := M;

while R < N loop
--assert true;
--# assert M = Q * N + R and R >= 0 and R < N;
--0--
Q := Q + 1;
R := R - N;
end loop;

end DivQuo_Procedure;
end DivQuo; end DivQuo;
C.2 Sorted Array Search Algorithm

Listing C.2: Ada Specification – Sorted Array Search Algorithm

```ada
package Search_SortedArray
  is
    -- Note To Programmer: Please reassign desired value for constant n.
    n: Constant := 1000;
    Type f_Type is range 1 .. n;
    Type f_Array_Type is array (f_Type) of Natural;
    f: f_Array_Type;
    v: Natural;

    procedure Search_SortedArray_Procedure(r: out f_Type; p: out f_Type; q: out f_Type);
      -- global in f, v;
      -- derives r from f, v & p from f, v & q from f, v;
      -- pre n >= 1;
      -- post f(r) = v;
  end Search_SortedArray;
```

Listing C.2: Ada Body – Sorted Array Search Algorithm

```ada
package body Search_SortedArray is
  procedure Search_SortedArray_Procedure(r: out f_Type; p: out f_Type; q: out f_Type)
  is
    begin
      r := (1+n)/2;
      p := 1;
      q := 1;

      while f(r)/= v loop
        -- assert f(r)/= v;
        if f(r) < v then
          p := r + 1;
          r := (r+1+q)/2;
        elsif f(r) > v then
          q := r - 1;
          r := (p+r-1)/2;
        end if;
      --0--
      end loop;

    end Search_SortedArray_Procedure;
  end Search_SortedArray;
```
C.3 Filter Program

Listing C.3: Ada Specification – Filter Program

package Filter
--# own DataSet;
is
--Note To Programmer: Please reassign desired value for constant n.
n: Constant:= 1000;
Type DataSet_Type is range 0 .. n;
Type DataSet_Array_Type is array (DataSet_Type) of Natural;

DataSet: DataSet_Array_Type;

procedure Filter_Procedure(x: out DataSet_Type; result: out Integer);
--# global in DataSet;
--# derives x from DataSet & result from DataSet;
--# pre n >= 1;
--# post result >= 0 and x = n;
end Filter;

Listing C.3: Ada Body – Filter Program

package body Filter is

procedure Filter_Procedure(x: out DataSet_Type; result: out Integer)
is
begin
x := 1;
result := 0;

while x < n and DataSet(x) > 1 and DataSet(x)<=100 loop
--# assert result >= 0 and x <= n and x < n and DataSet(x) > 1
and DataSet(x)<=100;
--0--
result := result + DataSet(x);
x := x + 1;
end loop;
end Filter_Procedure;

end Filter;
C.4 Simple Sorting Algorithm

Listing C.4: Ada Specification – Simple Sorting Algorithm

package SimpleSort

is

-- Note To Programmer: Please reassign desired value for constant n.
 n: Constant:= 1000;
 Type f_Type is range 1 .. n;
 Type f_Array_Type is array (f_Type) of Natural;

f: f_Array_Type;

procedure SimpleSort_Procedure(g: out f_Array_Type; k: out f_Type; l: out f_Type; j: out f_Type);

end SimpleSort;

Listing C.4: Ada Body – Simple Sorting Algorithm

package body SimpleSort is

procedure SimpleSort_Procedure(g: out f_Array_Type; k: out f_Type; l: out f_Type; j: out f_Type)

is

begin
 g := f;
 k := 1;
 l := 1;
 j := 1;

 while k/= n and j = n loop
  -- assert k/= n and j = n;
  while k/= n and j/= n loop
   -- assert k/= n and j = n and k/= n and j/= n;
   if g(l)<=g(j + 1) then
    j := j + 1;
   elsif g(l) > g(j + 1) then
    j := j + 1;
    l := j + 1;
   end if;

  --1--
   end loop;
  --0--
   g(k) := g(l);
   k := k + 1;
   l := k + 1;
   j := k + 1;

  end loop;

end SimpleSort_Procedure;

end SimpleSort;
C.5 Summation

Listing C.5: Ada Specification – Summation

```ada
package Summation
--# own f;

is
--Note To Programmer: Please reassign desired value for constant n.
n: Constant:= 1000;
Type f_Type is range 1 .. n;
Type f_Array_Type is array (f_Type) of Natural;
f: f_Array_Type;

procedure Summation_Procedure(k: out f_Type; total: out Integer);
--# global in f;
--# derives k from & total from f;
--# post k = n;
end Summation;
```

Listing C.5: Ada Body – Summation

```ada
package body Summation is

procedure Summation_Procedure(k: out f_Type; total: out Integer)
is
begin
k := 1;
total := 0;

while k /= n loop
--# assert k /= n;
--0--
total := total + f(k);
k := k + 1;
end loop;

end Summation_Procedure;
end Summation;
```
C.6 Least Common Multiple

Listing C.6: Ada Specification – Least Common Multiple

package LCM
--# own A, B;
is
   --Note To Programmer: Please reassign desired value for constant n.
   n: Constant:= 1000;
   Type A_Type is range 1 .. n;
   A: A_Type;
   B: A_Type;

   procedure LCM_Procedure(i: out A_Type; j: out A_Type; lcm_result: out A_Type);
   --# global in A, B;
   --# derives i from A, B & j from A, B & lcm_result from A, B;
   --# pre n >= 1;
   --# post (A*i) = (B*j);
end LCM;

Listing C.6: Ada Body – Least Common Multiple

package body LCM is
   procedure LCM_Procedure(i: out A_Type; j: out A_Type; lcm_result: out A_Type)
   is
      begin
      i := 1;
      j := 1;
      lcm_result := 1;

      while (A*i) /= (B*j) loop
         --# assert (A*i) /= (B*j);
         if (A*i) < (B*j) then
            i := i+1;
            lcm_result := i*j;
         elsif (A*i) > (B*j) then
            j := j + 1;
            lcm_result := i*j;
         end if;
      end loop;
   end LCM_Procedure;
end LCM;
C.7 Minimum Search

Listing C.7: Ada Specification – Minimum Search

```ada
package Search_Minimum
--# own f;
is
--Note To Programmer: Please reassign desired value for constant n.
n: Constant := 1000;
Type f_Type is range 1 .. n;
Type f_Array_Type is array (f_Type) of Natural;
f: f_Array_Type;

procedure Search_Minimum_Procedure(x: out f_Type; minimum: out Natural);
--# global in f;
--# derives x from f & minimum from f;
--# pre n >= 1;
--# post x = n and (For all i in f_Type range 1..n-1 =>
(f(i)>=minimum));
end Search_Minimum;
```

Listing C.7: Ada Body – Minimum Search

```ada
package body Search_Minimum is
procedure Search_Minimum_Procedure(x: out f_Type; minimum: out Natural)
is
begin
x := 1;
minimum := f(n);

while x /= n loop
--# assert (For all i in f_Type range 1..x-1 =>
(f(i)>=minimum)) and x/= n;
if minimum <= f(x) then
x := x + 1;
elsif minimum > f(x) then
minimum := f(x);
x := x + 1;
end if;
--0--
end loop;

end Search_Minimum_Procedure;
end Search_Minimum;
```
C.8 Power Evaluation

Listing C.8: Ada Specification – Power Evaluation Program

package PowerEvaluation
--# own m;
is

-- Note To Programmer: Please reassign desired value for constant n.
n: Constant:= 1000;
-- Note To Programmer: Please reassign desired value for constant x.
x: Constant:= 1000;

Type m_Type is range 1 .. n;

m: m_Type;

procedure PowerEvaluation_Procedure(k: out m_Type; y: out Integer);
--# global in m;
--# derives k from m & y from m;
--# pre x >= 1 and n >= 1 and m <= n;
--# post y >= 1 and k/= 0;
end PowerEvaluation;

Listing C.8: Ada Body – Power Evaluation Program

package body PowerEvaluation is

procedure PowerEvaluation_Procedure(k: out m_Type; y: out Integer)
is
begin
  k := m;
y := 1;

  while k/= 0 loop
    --# assert y >= 1 and k/= 0;
    --0--
    k := k - 1;
y := y * x;
  end loop;

end PowerEvaluation_Procedure;
end PowerEvaluation;
C.9 Polynomial Evaluation Program


package Polynomial
--# own f;
is

--Note To Programmer: Please reassign desired value for constant n.
n: Constant:= 1000;
--Note To Programmer: Please reassign desired value for constant x.
x: Constant:= 1000;

Type f_Type is range 1 .. n;
Type f_Array_Type is array (f_Type) of Natural;
f: f_Array_Type;

procedure Polynomial_Procedure(k: out f_Type; total: out Integer);
--# global in f;
--# derives k from & total from f;
--# post k = 1;
end Polynomial;

Listing C.9: Ada Body – Polynomial Evaluation Program

package body Polynomial is
    procedure Polynomial_Procedure(k: out f_Type; total: out Integer)
is
    begin
        k := n;
        total := 0;
        while k /= 1 loop
            --# assert k /= 1;
            --0--
            k := k - 1;
            total := total * x + f(k);
        end loop;
    end Polynomial_Procedure;
end Polynomial;
Appendix D

Semantic Analysis Summary

D.1 Division Algorithm

Listing D.1: Division Algorithm – Verification Conditions

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>rtc check @ 13</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>rtc check @ 14</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>7</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

Listing D.1: Division Algorithm – Total VCs by type:

<table>
<thead>
<tr>
<th></th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCToR Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert/Post</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals:</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>%Totals:</td>
<td>0%</td>
<td>75%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>25%</td>
</tr>
</tbody>
</table>

<<<

<<>
D.2 Sorted Array Search Algorithm

Listing D.2: Sorted Array Search Algorithm – Verification Conditions

VCs for procedure_search_sortedarray_procedure :

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>rtc check @ 8</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>start</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>start</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>rtc check @ 12</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>rtc check @ 13</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>rtc check @ 14</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>rtc check @ 15</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>rtc check @ 16</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>rtc check @ 17</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>18</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>20</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

---

Listing D.2: Sorted Array Search Algorithm – Total VCs by type:

Total VCs by type:  

<table>
<thead>
<tr>
<th></th>
<th>Proved By Or Using</th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCToR Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert/Post</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td></td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Totals: 21 0 16 0 0 0 0 5

%Totals: 0% 76% 0% 0% 0% 24%

---
## D.3 Filter Program

### Listing D.3: Filter Program – Verification Conditions

VCs for procedure_filter_procedure:

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>rtc check @ 9</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>rtc check @ 9</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>start</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>rtc check @ 12</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>rtc check @ 13</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>9</td>
<td>start</td>
<td>assert @ finish</td>
<td>False</td>
<td>No DPC</td>
<td>F-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>assert @ finish</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
</tbody>
</table>

### Listing D.3: Filter Program – Total VCs by type:

Total VCs by type:

<table>
<thead>
<tr>
<th></th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCToR Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert/Post</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Totals: 10 0 6 0 0 1 3

%Totals: 0% 60% 0% 0% 0% 10% 30%

---

96
### Listing D.4: Simple Sorting Algorithm – Verification Conditions

VCs for procedure_search_sortedarray_procedure:

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>rtc check @ 8</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>start</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>start</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>rtc check @ 12</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>rtc check @ 13</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>rtc check @ 14</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>rtc check @ 15</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>rtc check @ 16</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>rtc check @ 17</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>18</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>20</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Examnr</td>
<td>Simp (User) ViCToR Checkr Review False Undisc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assert/Post</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals:</td>
<td>21</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%Totals:</td>
<td>0%</td>
<td>76%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

---

### Listing D.4: Simple Sorting Algorithm – Total VCs by type:

Total VCs by type:

<table>
<thead>
<tr>
<th>Assert/Post</th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCToR Checkr Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals:</td>
<td>21</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<<<

%Totals: 0% 76% 0% 0% 0% 0% 24%
D.5 Summation

**Listing D.5: Summation – Verification Conditions**

VCs for procedure_summation_procedure:

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check  @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check  @ 7</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>rtc check @ 12</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>rtc check @ 13</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>7</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

**Listing D.5: Summation – Total VCs by type:**

Total VCs by type:

<table>
<thead>
<tr>
<th>Assert/Post</th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCTOR Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Totals: 8 0 6 0 0 0 0 2

<<<%Totals: 0% 75% 0% 0% 0% 0% 25%
## D.6 Least Common Multiple

### Listing D.6: Least Common Multiple – Verification Conditions

VCs for procedure lcm_procedure:

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>rtc check @ 8</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>start</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>rtc check @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>start</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>assert @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>rtc check @ 12</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>rtc check @ 13</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>rtc check @ 14</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>rtc check @ 15</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>rtc check @ 16</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>rtc check @ 17</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>18</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>20</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

---

### Listing D.6: Least Common Multiple – Total VCs by type:

Total VCs by type:

<table>
<thead>
<tr>
<th></th>
<th>Proved By Or Using</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Examinr Simp (User) ViCToR Checkr Review False Undisc</td>
</tr>
<tr>
<td>Assert/Post</td>
<td>8 0 8 0 0 0 0 0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Check stmtn.</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>13 0 6 0 0 0 0 7</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Totals: 21 0 14 0 0 0 0 7

%Totals: 0% 67% 0% 0% 0% 0% 33%
D.7 Search Minimum

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>assert @ 10</td>
<td>Un discharged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>assert @ 10</td>
<td>Un discharged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>assert @ 10</td>
<td>Contradiction</td>
<td>No DPC</td>
<td>X-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>rtc check @ 11</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>rtc check @ 12</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>rtc check @ 13</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>rtc check @ 14</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>rtc check @ 15</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>12</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>VICToR Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert/Post</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Refin. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Totals: 15 0 13 0 0 0 2

%Totals: 0% 87% 0% 0% 0% 0% 13%
D.8 Power Evaluation Program

Listing D.8: Power Evaluation Program – Verification Conditions

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>rtc check @ 12</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>rtc check @ 13</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>7</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

Listing D.8: Power Evaluation Program – Total VCs by type:

Total VCs by type:

<table>
<thead>
<tr>
<th></th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCToR</th>
<th>Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert/Post</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precondition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check stmt.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Runtime check</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Refinem. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inherit. VCs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Totals: 8 0 5 0 0 0 0 3

%Totals: 0% 63% 0% 0% 0% 0% 38%
## D.9 Polynomial Evaluation Program

### Listing D.9: Polynomial Evaluation Program – Verification Conditions

VCs for procedure_polynomial_procedure:

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>To</th>
<th>Proved By</th>
<th>Dead Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>start</td>
<td>rtc check @ 6</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>rtc check @ 7</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>assert @ 10</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>rtc check @ 12</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>rtc check @ 13</td>
<td>Undischarged</td>
<td>No DPC</td>
<td>U-</td>
</tr>
<tr>
<td>7</td>
<td>start</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>assert @ finish</td>
<td>Inference</td>
<td>No DPC</td>
<td>I-</td>
</tr>
</tbody>
</table>

### Listing D.9: Polynomial Evaluation Program – Total VCs by type:

Total VCs by type:

<table>
<thead>
<tr>
<th>Assert/Post</th>
<th>Total Examnr</th>
<th>Simp (User)</th>
<th>ViCToR Checkr</th>
<th>Review</th>
<th>False</th>
<th>Undisc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

% Totals: 0% 75% 0% 0% 0% 25%

---

102