3-2023

Safe and Reliable Software and the Formal Verification of Prim's Algorithm in SPARK

Brian S. Wheelhouse

SAFE AND RELIABLE SOFTWARE AND THE FORMAL VERIFICATION OF PRIM'S ALGORITHM IN SPARK

THESIS

Brian S Wheelhouse, Second Lieutenant, USAF

AFIT-ENG-MS-23-M-070

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-23-M-070

SAFE AND RELIABLE SOFTWARE AND THE FORMAL VERIFICATION OF PRIM'S ALGORITHM IN SPARK

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Brian S Wheelhouse, B.S.C.S.

Second Lieutenant, USAF

March 2023

AFIT-ENG-MS-23-M-070

SAFE AND RELIABLE SOFTWARE AND THE FORMAL VERIFICATION OF PRIM'S ALGORITHM IN SPARK

THESIS

Brian S Wheelhouse, B.S.C.S.
Second Lieutenant, USAF

Committee Membership:

Kenneth M Hopkinson, Ph.D
Chair

Laura R Humphrey, Ph.D
Member

Douglas D Hodson, Ph.D
Member

Scott R Graham, Ph.D
Member

AFIT-ENG-MS-23-M-070

Abstract

Despite evidence that formal verification helps produce highly reliable and secure code, formal methods, i.e., mathematically based tools and approaches for software and hardware verification, are not commonly used in software and hardware development. The limited emphasis on formal verification in software education and training suggests that many developers have never considered the benefits of formal verification. Despite the challenging nature of their mathematical roots, software verification tools have improved; making it easier than ever to verify software. SPARK, a programming language and a formal verification toolset, is of particular interest for the Air Force Research Laboratory (AFRL), and will be a primary focus of this thesis.

This thesis provides an overview of two safe and reliable languages with verification tools, namely SPARK and Rust. Then, to demonstrate the benefits of modern software verification tools, two examples of software verification in SPARK are presented. These examples include a verified implementation of the quaternion data structure and two implementations of Prim's algorithm, to further demonstrate the usability and methodology of the SPARK verification toolset.

Acknowledgements

Many thanks to my advisor and committee for their mentorship, revisions, and suggestions, and specifically to Dr. Laura Humphrey for contributing the Pythagorean theorem analogy, Compact Position Reporting algorithm (CPR), Automatic Dependent Surveillance-Broadcast (ADS-B), and NASA's Deep Space 1 probe examples found in paragraph two of Chapter II, and much of the background on formal methods, particularly those found in paragraph two of Chapter III.

Brian S Wheelhouse

Table of Contents

Page

List of Figures

List of Tables

SAFE AND RELIABLE SOFTWARE AND THE FORMAL VERIFICATION OF PRIM'S ALGORITHM IN SPARK

I. Introduction

1.1 Problem Background

Creating highly reliable and secure software is challenging. Most programming languages seem to consider reliability and security features as an afterthought, leaving the burden on the software developer to ensure the security and reliability of the software they produce. This is usually done by unit testing and sometimes fuzzing. However, these types of software testing approaches can only show that software errors exist and not that the software is error-free. As software has become increasingly complex, some programming languages have begun to consider reliability and security as key elements of the language design itself. Two languages, SPARK (a subset of Ada) and Rust, are prime examples of this trend in language design. Both languages also have software verification tools available that allow developers to prove that their software is functionally correct with respect to its specifications and that it is free from runtime errors and other possible bugs through static analysis.

The use of verification tools has begun to play a role in the certification of some safety critical systems such as those found in airplanes, trains, and automobiles, and also shows promising results for ensuring reliability in other software systems. As an example, consider that many distributed systems use a minimum spanning tree (MST) as the backbone for efficient communication within the system; in such a critical role, it is important that the MST source code is correctly implemented. One way to

ensure its correctness with a high degree of confidence is to use formal methods, i.e., mathematically based tools and approaches for the design and verification of software and hardware. Toward this end, an implementation of Prim's algorithm for constructing MSTs is provided in SPARK, which is both a programming language and an associated set of formal verification tools. The SPARK verification tools are then used to formally verify various properties of the implementation. At the most basic level, formal verification in SPARK requires proving that code satisfies data flow and initialization contracts and is free of runtime errors. Formal verification often reveals rare or subtle errors that are hard to detect through testing alone. Once errors have been corrected and all properties of the specification have been formally verified, the result is highly reliable software that is mathematically proven to meet the defined specification requirements.

1.2 Research Objectives

This thesis focuses on the benefits and features of safe and reliable software with an emphasis on SPARK. To provide background on formal verification and some of the attributes of highly reliable programming languages and tools, Chapter II gives an overview of SPARK and Rust and poses the following research question:

> *RQ1: What are common attributes of highly reliable programming languages and how do those attributes and formal verification enhance the quality of software?*

With an understanding of the benefits of a secure language and using software verification tools, Chapter III gives a tutorial of the basic formal verification of a naive implementation of Prim's algorithm. This proof of Prim's algorithm is performed without manually adding preconditions or postconditions, loop invariants, or other

annotations to assist in the proof, making it an automatic proof. It addresses the following research question:

RQ2: What qualities does an automatic proof of Prim's algorithm have?

Although this implementation and proof of Prim's algorithm in Chapter III is basic and beginner-oriented, the resulting code is a solid foundation for a reliable implementation of the algorithm. In fact, the results lead to a more interesting proof of Prim's algorithm and raise the following question, which is examined in Chapter IV:

RQ3: How do you "level up" basic SPARK code to a meaningful proof of functional correctness for Prim's algorithm?

Finally, Chapter V revisits these research questions and summarizes the results found in response to each question.

II. SPARK and Rust: An Overview of Safe and Reliable Software*

2.1 Introduction

Ensuring that a software system is secure and highly reliable is a challenging task. Security and reliability can be easy to overlook when programming languages and compilers fail to adopt programming practices that help prevent bugs, and thorough testing of software, with all its edge cases, is unrealistic. A possible solution to this problem is to use a language that enforces secure and reliable programming practices and to use formal verification tools to prove that the code is free from common errors. Rust and SPARK are two modern programming languages that have been built with security and reliability in mind. Reviewing the characteristics of these languages shows how they catch bugs early on and prevent security and reliability issues down the road. Both languages have formal verification tools that are able to prove that the software meets its desired specifications.

Rather than mathematically proving that software is correct, today's standard is to settle for writing tests for software that are often thorough, but hardly ever exhaustive or complete. The problem is that software testing proves that errors exist, but not that the software is error-free [4]. This practice of test-driven development (as opposed to proof-driven development) has led to a history of unsafe and insecure software. Testing software is a key process in the software development life-cycle; however, to improve software generally, formal verification as a component of proof-driven development can also be applied in the development life-cycle to enhance the safety and reliability of code. Formal verification is the process of creating mathemat-

*This chapter is based on a paper that will be published in an upcoming conference proceedings: B. Wheelhouse, K. Hopkinson, L. Humphrey, D. Hodson, "SPARK and Rust: An Overview of Safe and Reliable Software", in *Proceedings of the 2022 World Congress of Computer Science, Computer Engineering, and Applied Computing (CSCE)*, Las Vegas, 2022.

ically or logically verifiable software. It requires creating formal specifications defined in mathematical or logical terms, writing code that satisfies the specifications, and then using a variety of tools such as automated theorem provers and satisfiability modulo theories (SMT) solvers to mathematically prove that the code satisfies its specifications for all allowed inputs. Consider an analogy based on the Pythagorean theorem. Testing the Pythagorean theorem with various right triangles increases confidence that the theorem is correct, while formally verifying the theorem using geometric axioms proves that it is correct for all right triangles [5]. Similarly, formal verification tools often find errors that are not found by testing, both in software and other design artifacts such as requirements and algorithms. For example, consider the Compact Position Reporting algorithm (CPR), which is part of the Automatic Dependent Surveillance-Broadcast (ADS-B) protocol. Its goal is to encode aircraft position in terms of latitude and longitude in 17-bits each, then decode the encoded position back to approximate latitude and longitude values that are within 5 meters of the original position. However, significant errors were observed in positions computed by fielded implementations of CPR. Formal verification tools were later used to reformulate the underlying algorithms and verify software implementations of them [6]. As another example, consider NASA's Deep Space 1 probe, which included an artificial intelligence-based spacecraft control system architecture. Before fielding, formal verification tools were applied to services to manage interacting parallel goal-and-event driven processes in the plan execution module. As a result of this effort, five previously undiscovered concurrency errors were found, one of which was a major design flaw [7]. During operation, an analogous flaw in an unanalyzed part of the system resulted in problems with the probe's thrusters, and a remote patch that used an analogous fix was applied.

In some cases, formally specifying and verifying software may seem unreasonable.

Many critics argue that formal verification is too cumbersome and impractical due to the cost and the possibility of slowing down the development process. The challenges of verifying software are certainly evident when considering that software is constantly changing. Each new feature potentially requires adjustments to the formal specification and may require new and complex proofs to verify the code. As verification tools improve and become more accessible, the ease and flexibility of verifying software becomes less of a burden.

Advocates of formal methods claim that formal methods are an underused tool in software development, and mathematically or logically verifying the code will improve security and lead to highly reliable software [8]. Even when some education about the philosophy and underlying concepts of formal verification is applied informally, the quality of software can improve [4].

2.2 Safe and Reliable Languages

Rust and SPARK are two of the first languages that come to mind when it comes to safe and reliable programming languages. Safety and reliability have been key in the language constructs of both languages since they were created. The design requirements for both languages feature considerable similarities [9]: "Rust is a systems programming language focused on three goals: safety, speed, and concurrency... Making it a useful language for a number of use cases that other languages are not good at: embedding in other languages, programs with specific space and time requirements, and writing low-level code, like device drivers and operating systems. It improves on current languages targeting this space by having a number of compile-time safety checks that produce no runtime overhead, while eliminating all data races." [10]. Compare these requirements to the DoD *Ironman requirements* that led to Ada (the language from which SPARK was developed) in the 1970s: "The language

shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications require real-time control, self-diagnostics, input-output to non-standard peripheral devices, parallel processing, numeric computation, and file processing. ... The language should aid in the design and development of reliable programs. The language shall be designed to avoid error-prone features and maximize automatic detection of programming errors." [11].

Table 1 highlights some of the similarities between the two languages that Ochem describes in [9]. These features suggest some of the key features of a highly reliable language. Strict type safety prohibits implicit conversions between objects of different data types, which is a common source of errors. Strong static typing checks type safety at compile time to ensure that type issues are not encountered at runtime. Static memory checks ensure that the code handles memory safely and reliably before a program even runs. A key feature for safe memory handling is anti-aliasing, which removes the possibility of modifying shared data. For pointers, Rust uses an ownership policy that enforces anti-aliasing. SPARK was recently extended to include pointers (called access types) based on the Rust ownership model as described in [12] [13], although access types in SPARK have some limitations compared to Rust. Both languages manage memory directly. This means that instead of a garbage collector deciding what allocated memory is no longer needed, the allocated memory is freed

Table 1: SPARK and Rust Similarities Overview

Language Construct	**SPARK**	**Rust**
Strict Type Safety	✓	✓
Strong Static Typing	✓	✓
Static Memory Checks	✓	✓
Dynamic Memory (Pointers)	✓	✓
Memory Aliasing	✗	✗
Garbage Collection	✗	✗
Exception Handling	✗	✗

as soon as it goes out of scope. SPARK and Rust both have very limited forms of exception handling, which forces a developer to write code such that exceptions do not exist. This makes the code more reliable as exceptions that are not caught may crash a program.

Table 2 highlights some of the differences between the two languages that Ochem describes in [9]. These highlighted differences show stylistic choices or areas where the two languages are developing. For instance, Rust supports closures and lambda expressions, while SPARK does not. Rust also supports static type inference, whereas there is almost no type inference in SPARK. The SPARK language includes constructs for writing behavioral contracts on functions and procedures, including preconditions and postconditions. These are not included in the core Rust language, although there are crates such as the *contracts* crate [14] and the *prusti_contracts* crate that provide them. Given such contracts, formal verification tools can attempt to prove that programs satisfy their contracts, though this may require some assistance or guidance from the developer. By default, SPARK will attempt to prove contracts through analysis. However, since contracts in SPARK must be executable, there is also the option to compile them as runtime checks in the code, which is useful for testing or better exception handling if formal verification cannot be completed. To name a few other differences, Rust does not have separate specification and implementation files like SPARK does. File separation is a stylistic choice that is not critical to

Table 2: SPARK and Rust Differences Overview

Language Construct	**SPARK**	**Rust**
Support for Lambdas / Closures	✗	✓
Static Type Inference	✗	✓
Pre- and Post-Conditions	✓	✗
Separate Spec. and Impl. Files	✓	✗
Verify Class Consistency	✓	✗

software reliability or safety, but may be a helpful separation with respect to formal verification.

Leaving aside the stylistic characteristics, both languages treat reliability and safety in very similar ways, which improves the ability to effectively verify programs in these languages.

2.3 Formal Verification Tools for Rust and SPARK

This section provides an overview of how tools for formal deductive program verification work, which summarizes some of the introductory points in [1]. Then we discuss aspects of the *Prusti* formal verification tool set for Rust [15] and the SPARK formal verification tool set [16].

Modern tools for formal deductive program verification attempt to automatically verify that software is free from run-time errors and some common vulnerabilities using mathematical and logical approaches such as weakest precondition calculus and tools such as SMT solvers. This is done by translating code with checks and contracts to be verified to a deductive verification platform such as Why3 (used by SPARK) or Boogie (used by Prusti) [17]. The deductive verification platform then generates verification conditions (VCs) using a weakest-precondition calculus that, if proven valid by SMT solvers or other provers, implies that the code satisfies its specifications. *Formal* verification tools are designed so that they should never state that a VC is true if it is not actually true. Figure 1 shows how several different verification toolsets use Why3 as a platform for deductive verification, with other formal verification tools used to prove VCs.

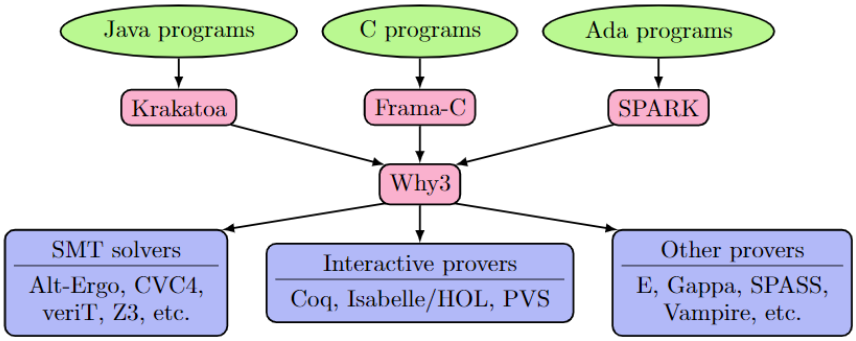

Figure 1: Deductive Program Verification Tools That Use Why3 (reproduced from [1])

2.3.1 Prusti

Prusti is a new static verifier for the Rust language based on the Viper verification infrastructure [18] [19]. Running this tool on Rust code will ensure that statements such as *panic!()* and *unreachable!()* are unreachable and potential overflow vulnerabilities are checked. Additionally, developers can add annotations such as preconditions and postconditions and loop invariants to prove the functional behavior of the code. If any of these checks fails, an error is reported stating that the code does not meet the specifications.

2.3.2 SPARK

SPARK, which is a subset of the Ada programming language, is both a programming language and a formal verification toolset. The SPARK toolset verifies code without compiling or executing it. SPARK performs different types of analysis. The simplest form of analysis *check*s that the code is valid SPARK code, that is, the code does not contain any disallowed features of the Ada language. The next level of analysis, *flow analysis*, checks the initialization of variables, unused assignments, unmodified variables, and data dependencies between the input and output of the subprograms. After flow analysis, *proof* analysis uses the toolset's provers to check

for the absence of runtime errors and the conformance of the program with its specifications. When proof analysis is performed, check and flow analysis are also performed since the proofs are not accurate unless the check and flow analysis pass.

Given the different types of analysis SPARK can perform and the fact that the level of detail in program specifications can vary from partial to complete, some colloquial terms have been adopted by SPARK users to indicate the level of verification that has been attained for the code. From lowest to highest, these levels are colloquially referred to as "stone," "bronze" "silver," "gold," and "platinum" [3]. A simple example of a gold level proof can be found in [20], which verifies that a merge sort algorithm satisfies a partial specification. Table 3 gives more details on the levels of SPARK verification and the use cases at each level.

Table 3: Levels of SPARK Verification [3].

Level	**Guarantees**	**Use Case**
Stone	Valid SPARK	Intermediate level during the adoption of SPARK
Bronze	Initialization, correct data flow	As large a part of the code as possible
Silver	Absence of run-time errors (AoRTE)	The default target for critical software (subject to costs and limitations)
Gold	Proof of key integrity properties	Only for a subset of the code subject to specific key integrity (safety/security) properties
Platinum	Full functional proof of requirements	Only for those parts of code with the highest integrity (safety/security) constraints

[21] and [5] provide an overview of how SPARK has been used in various projects, especially projects focused on cyber security and safety in the aerospace domain. These include the Ship Helicopter Operating Limits Information System (SHOLIS); the C130J "Hercules" core mission computer, which saw 80% savings in the modified condition/decision coverage (MC/DC) testing budget due to the low number of faults discovered during testing; and the NSA-funded Tokeneer demonstrator, for

which testing found zero defects for a period after delivery. In terms of cyber security, the silver-level verification in SPARK shows that the code is free of many of the cyber vulnerabilities classified in the MITRE Corporation Common Weakness Enumeration (CWE) database [22]. This is discussed in [23] which provides a mapping of how the language features of SPARK/Ada prevent certain classes of CWEs and how verification with SPARK prevents others.

2.4 Example: Quaternion Data Type

As an exercise to explore the benefits and limitations of formal deductive program verification, we verify an implementation of the quaternion data structure and associated operations in SPARK. Quaternions are useful in simulations of rigid body objects, such as airplanes. Bourg, the author of the book "Physics for Game Developers: Science, math, and code for realistic effects" defines and presents an implementation of quaternions as a means of solving the gimbal lock problem in a flight simulator in Chapter 11 of [24]. To summarize Bourg's definition, quaternions are a four-dimensional quantity similar to a vector that is made up of four components. A quaternion is typically represented in the form:

$$q = q_0 + q_x i + q_y j + q_z k \tag{1}$$

Furthermore, a *unit quaternion* satisfies the following, which makes it analogous to a unit vector:

$$q_0{}^2 + q_x{}^2 + q_y{}^2 + q_z{}^2 = 1 \tag{2}$$

We converted the quaternion sample code from [24] to Rust and then to SPARK. This implementation presents a quaternion as a scalar portion (q_0) and a three-element

```
1   --General Purpose Quaternion-----------
2   type Spark_Quaternion is record
3      N: Float := 0.0;  -- Scalar Part
4      V: Vector3;       -- Vector Part
5   end record;
6   --General Purpose 3-element vector------
7   type Vector3 is record
8      X: Float := 0.0;
9      Y: Float := 0.0;
10     Z: Float := 0.0;
11  end record;
```

Figure 2: Quaternion Representation in SPARK

vector to represent q_x, q_y and q_z, with each value represented as a floating point number, as shown in Figure 2.

Vector3 is our three-element vector implementation which implements the following methods:

- Magnitude
- Normalize
- Cross Product
- "+" operator
- "-" operator
- "*" operator
- "/" operator

These methods help simplify the implementation of the quaternion package, which includes the following functionality:

- Magnitude
- Get Vector

- Get Scalar
- Get Angle
- Get Axis
- Rotate Quaternion
- Rotate Quaternion by Vector
- Make Quaternion From Euler Angles
- Make Euler Angles From Quaternion
- "not" operator
- "+" operator
- "-" operator
- "*" operator
- "/" operator

All vector and quaternion methods and a couple of utility functions to convert degrees to radians and radians to degrees are included in the SPARK analysis summary presented in Figure 3. To help the provers, we added preconditions on the divide ("/" operator) subprograms to avoid division by zero.

Testing the SPARK implementation with a few test cases builds confidence that the implementation at least matches the Rust implementation, which we have not yet attempted to verify with Prusti. Running the SPARK verification tool in the *prove all* mode produces the summary of the SPARK analysis presented in Figure 3. [25] provides finer details of the SPARK analysis results table; however, for the purposes

Summary of SPARK analysis

SPARK Analysis results	Total	Flow	Provers	Unproved
Data Dependencies	.	.	.	.
Flow Dependencies	.	.	.	.
Initialization	5	5	.	.
Non–Aliasing	.	.	.	.
Run–time Checks	234	.	102 (CVC4 92%, Trivial 8%)	132
Assertions	.	.	.	.
Functional Contracts	10	.	8 (CVC4)	2
LSP Verification	.	.	.	.
Termination	.	.	.	.
Concurrency	.	.	.	.
Total	249	5(2%)	110 (44%)	134 (54%)

max steps used **for** successful proof: 4494

Figure 3: Initial SPARK Analysis of Quaternion Implementation

of this chapter, we will relate our results to the levels of SPARK use described in Table 3.

Based on the results in the analysis summary, this quaternion implementation passes flow analysis, which corresponds to a bronze level SPARK proof, because there are no data or flow dependencies and no accesses to uninitialized variables. Due to the size of the code base and the use of unbounded floats, 132 unproved run-time checks are produced with warnings similar to:

```
quaternion.adb:160:46: medium: float overflow check
might fail [reason for check: result of floating-
point multiplication must be bounded]
[possible fix: subprogram at quaternion.ads:50 should
mention LEFT and RIGHT in a precondition][#71]
```

Unproved checks involving floating-point computations make increasing the level of proof to the silver level complex and challenging. As stated in [26], "General pur-

pose tools for the deductive verification of programs have added support for floating-point computations in recent years, but often the proved properties are limited to verifying ranges or the absence of special values such as not a number (NaN) or Infinity. Proofs of more complex properties, such as rounding error bounds, are generally reserved to experts and often require the use of a proof assistant or a specialized solver as a backend." Resolving overflow checks, such as the ones shown above, may resolve the two unproved functional contracts; however, the unproved functional contracts should only be addressed once absence of runtime errors (AoRTE) is ensured. Once AoRTE is ensured, the code will meet the standards for a silver level proof.

2.5 Conclusions

Formally verifying software comes at the cost of educating software developers about formal methods for software verification. Choosing a safe and reliable language along with formal verification tools and techniques can increase the reliability and security of software by automatically checking commonly overlooked security and reliability concerns in software. SPARK and Rust, with their associated verification tools, exemplify how safe and reliable software goals can be achieved today.

In the future we would like to increase the level of proof of the quaternion package to the silver level by exploring available options for bounding floating point numbers in [26]. A gold-level proof could then be attained by creating an appropriate specification for quaternions and proving key properties of the quaternion implementation. We would also like to attempt to use Prusti to verify a Rust implementation of quaternions. The attempt would likely run into the same issues involving floating-point computations, but in any case it would be interesting to perform verification using both SPARK and Prusti and compare the results.

2.6 Additional Work

After the final draft was submitted for publication, specifying tighter bounds for input values and excluding large positive and negative floating point inputs for values *Deg_Angle*, *Rad_Angle*, and *Scalar* as suggested in [26] provided significantly improved results. Specifically, *Deg_Angle* was bounded to values between 0.0..360.0, *Rad_Angle* to $0.0..2.0 * Pi$, and *Scalar* to $-2^{24}..2^{24}$ which should be large enough to simulate any airplane that can physically be flown on Earth. The simple bounding of these three values decreased the number of run-time errors by more than 50% as shown in Figure 4. These results emphasize how simple specifications and bounding can increase the reliability of the code, although it is difficult to resolve all the issues when dealing with floating-point computations.

Summary of SPARK analysis

SPARK Analysis results	Total	Flow	Provers	Unproved
Data Dependencies	.	.	.	.
Flow Dependencies	.	.	.	.
Initialization	5	5	.	.
Non–Aliasing	.	.	.	.
Run–time Checks	316	.	252 (CVC4 31%, Trivial 69%)	64
Assertions	.	.	.	.
Functional Contracts	8	.	6 (CVC4)	2
LSP Verification	.	.	.	.
Termination	.	.	.	.
Concurrency	.	.	.	.
Total	329	5(2%)	258 (78%)	66 (20%)

max steps used **for** successful proof: 4529

Figure 4: SPARK Analysis After Bounding Specified Values

III. Formal Verification of Absence of Runtime Errors of Prim's Algorithm in SPARK*

3.1 Introduction

As software systems increase in complexity, it becomes more difficult to ensure their correctness. A major reason is that as the complexity of the system increases, the proportion of system behaviors that can be feasibly covered by standard test-based verification approaches decreases, leaving more room for latent errors. A possible solution to this problem lies in the use of formal methods, i.e., mathematically based tools and approaches for software and hardware verification [27, 28]. Whereas testing checks individual execution traces of a system, formal methods analyze a mathematical model of a system, opening the possibility of mathematically proving that all possible behaviors of the system are correct. To make an analogy, consider the Pythagorean Theorem. One could merely build confidence in its correctness by testing it against a set of randomly selected right triangles, or one could prove its correctness for all right triangles by applying geometric axioms.

In safety-critical domains, where errors can lead to substantial damage or loss of life, there is a need to eliminate as many errors as possible. Therefore, certification standards for many safety-critical domains promote the use of formal methods, for example, ISO 26262 for the automotive domain [29], EN 50128 for the railway domain [30], and the DO-333 supplement to DO-178C for the aerospace domain [31]. There is a perception that formal methods require significant expertise to use and may not provide a good return on investment [32, 4, 33], so historically the use of formal methods has been concentrated in safety-critical domains. However, the challenge

*This chapter is based on a paper that is published in the conference proceedings:
B. Wheelhouse, L. Humphrey, K. Hopkinson. "Formal verification of prim's algorithm in spark", in *Proceedings of the 56th Hawaii International Conference on System Sciences (HICSS)*, pages 6695–6703, 2023.

of maintaining software correctness in the face of growing complexity has recently motivated the use of formal methods in other domains. For example, engineers at Amazon Web Services (AWS) have been using formal methods since 2011 to help solve difficult design problems in systems that use distributed algorithms for data management [34], and AWS also uses formal methods to address a variety of cyber security concerns [35, 36, 37]. Despite perceptions that formal methods are difficult and expensive to use, Newcombe et al. (2015) found that "formal methods find bugs in system designs that cannot be found through any other technique we know of," and "Formal methods are surprisingly feasible for mainstream software development and give good return on investment." In fact, they found that using formal methods to write and check proofs of certain types of algorithm was actually faster and easier than doing so by hand. In general, while the use of formal methods does require some investment, the return on investment is eventually realized in terms of better reliability, security, and fewer bugs to fix after development [33].

For formal program verification, one language and toolset that we have found relatively easy to use is SPARK [16]. This is both because the design philosophy of the language emphasizes safe and correct programming, it is freely available as part of the GNAT Community Edition [38], and there are a number of educational materials and examples available [2, 39]. In this chapter, we show how to use SPARK to implement and verify certain properties of Prim's algorithm for building minimum spanning trees (MSTs). We choose to focus on this algorithm because MSTs are used in problems involving network reliability, classification, and routing [40, 41] and would benefit from formal verification given the need for reliability in these problem domains.

The remainder of the chapter proceeds as follows. Section 3.2 discusses a few applications of SPARK in industry and efforts to prove Prim's algorithm using various

methods, Section 3.3 gives some background on SPARK, Section 3.4 demonstrates how SPARK is used to develop and verify Prim's algorithm, Section 3.5 addresses the results of the analysis report generated by SPARK, and Section 3.6 concludes the chapter.

3.2 Related Work

Formal methods have been used successfully in a variety of large projects and in proving Prim's algorithm. [21] and [5] provide an overview of how SPARK has been used in various projects, especially projects focused on cyber security and safety in the aerospace domain. These include the Ship Helicopter Operating Limits Information System (SHOLIS); the C130J "Hercules" core mission computer, which saw 80% savings in the modified condition/decision coverage (MC/DC) testing budget due to the low number of faults discovered during testing; and the NSA-funded Tokeneer demonstrator, for which testing found zero defects for a period after delivery. In terms of cyber security, the silver-level verification in SPARK shows that the code is free of many of the cyber vulnerabilities classified in the MITRE Corporation Common Weakness Enumeration (CWE) database [22]. This is discussed in [23], which provides a mapping of how the language features of SPARK/Ada prevent certain classes of CWEs and how verification with SPARK prevents others.

We briefly note that other tools and frameworks perform analogous types of formal verification for different languages. For example, Frama-C is a framework for analyzing C code in which contracts and assertions are written in ANSI/ISO C Specification Language (ACSL) and plugins for formal verification are available [42]. A case study comparing ACSL/Frama-C with SPARK can be found in [43]. There is also Prusti for Rust [44] and Krakatoa [45] for Java, just to name a few.

In regard to Prim's algorithm, some efforts have already been made to apply

formal methods. Abrial, Cansell, and Méry give an approach to proving Prim's algorithm using the formal modeling tool Atelier B [46]. Atelier B is an environment for generating and proving proof obligations for formal models, e.g. of algorithms. Such models can be automatically translated into C, C++, Ada, or HIA code [47], but since errors could be introduced during this translation, additional program verification tools such as SPARK should be used.

Another effort has succeeded in a proof of full functional correctness of an executable implementation of Prim's algorithm written in verifiable C using Coq: CompCert and the Verified Software Toolchain (VST) separation logic deductive verifier [48]. Mohan demonstrates that Prim's algorithm works on disconnected graphs (thus finding a minimum spanning forest (MSF) rather than a MST) and predicts that more-automated tools such as Why3 would not be able to prove full functional correctness as easily as their work with VST.

Based on the denotational semantics of the language, SPARK translates programs along with checks and contracts to be verified to the Why3 deductive verification platform [17]. Why3 then uses a weakest-precondition calculus to generate verification conditions (VCs), i.e. logical formulas whose validity would imply the soundness of the code with respect to its checks and contracts. Why3 then uses multiple theorem provers/satisfiability modulo theories (SMT) solvers to discharge VCs, including CVC4 [49], Alt-Ego [50], and Z3 [51]. While the tools attempt to automate this process, sometimes additional assertions in the code must be provided by the user to guide the underlying provers. We demonstrate what SPARK is able to prove automatically using Prim's algorithm as an example. This example also serves as a simple tutorial on how to begin using SPARK to formally verify an executable implementation of a common algorithm.

3.3 SPARK

This section provides an overview of SPARK, much of which is summarized in [2]. SPARK is a programming language *and* a formal verification toolset. SPARK, as a programming language, is based on the Ada programming language. Ada has a number of features that help to support the development of safe and correct programs, which SPARK builds upon. However, SPARK both adds some features that support formal verification, and removes some features that make formal verification difficult. In summary, SPARK leverages Ada features, such as

- Type safety
- Ada 2012 *aspects* for writing contracts
- A package system that enables clean separation of interfaces from implementations

and removes features such as

- Aliasing (assigning two names to the same object)
- Exception handlers
- Backward goto statements
- Controlled types
- Side-effects in expressions, including functions

For users who rely on Ada features that are restricted in the SPARK subset, note that, while SPARK can be used to prove an entire program, it can also be applied to only specific parts of a program, including designated lines, subprograms, or packages. Combined with the fact that SPARK is compiled using an Ada compiler, this makes

it possible to mix unproven Ada code with restricted features into the program if necessary. The SPARK User's Guide goes into more detail about these restrictions in [52]. The relationship between SPARK and Ada is depicted in Figure 5.

As a static verification toolkit, SPARK verifies code without compiling or executing it. SPARK performs several different types of static analysis. One is *flow analysis*, which checks the initialization of variables, unused assignments, unmodified variables, and data dependencies between the input and output of subprograms. The other is *proof*, which checks the absence of runtime errors and the conformity of the program with its specifications.

Given the different types of analysis SPARK can perform and the fact that the level of detail in program specifications can vary from partial to complete, some colloquial terms have been adopted by SPARK users to define the level of assurance that has been attained for the code. From lowest to highest, these levels are colloquially referred to as "stone," "bronze" "silver," "gold," and "platinum" [3]. A simple example of a gold level proof can be found in [20], which verifies that a merge sort algorithm satisfies a partial specification. Table 4 gives more details on the levels of SPARK verification and the use cases at each level.

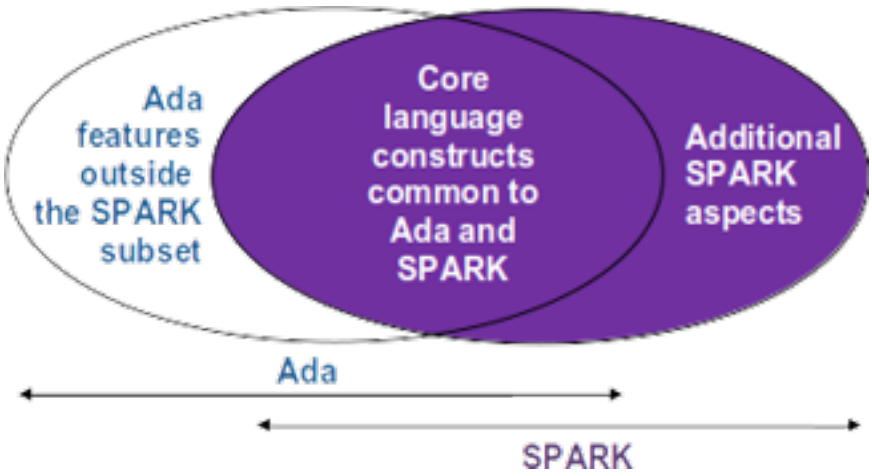

Figure 5: The Relationship Between SPARK and Ada [2].

Table 4: Levels of SPARK Verification [3].

Level	Guarantees	Use Case
Stone	Valid SPARK	Intermediate level during the adoption of SPARK
Bronze	Initialization, correct data flow	As large a part of the code as possible
Silver	Absence of run-time errors (AoRTE)	The default target for critical software (subject to costs and limitations)
Gold	Proof of key integrity properties	Only for a subset of the code subject to specific key integrity (safety/security) properties
Platinum	Full functional proof of requirements	Only for those parts of code with the highest integrity (safety/security) constraints

3.4 Example

Spanning trees can be used in communication protocols to provide paths from one node in the network to another non-neighboring node. A spanning tree is a subset of edges in a graph that connect all nodes or vertices in the graph without any cycles, where the number of edges is one less than the number of vertices. For a weighted graph, a *minimum* spanning tree is one whose edge weights have the smallest sum of all possible spanning trees in the graph [53].

Prim's algorithm is one algorithm that can be used to compute an MST for a graph. In this section, we start by describing this algorithm. Then, we present an implementation of this algorithm in SPARK and show how to refine it so that it is proved at the bronze and silver level.

3.4.1 Prim's Algorithm

Prim's algorithm is a greedy algorithm for finding an MST of a weighted undirected graph given a starting vertex. Let $G = (V, E)$ be a weighted undirected graph with vertices V, edges E, and a function $w : E \to \mathbb{R}$ assigning a weight $w(u, v)$ to every edge $(u, v) \in E$. Let us denote the set of vertices as $G.V$. Starting from an

arbitrary root vertex $r \in G.V$, Prim's algorithm incrementally builds a tree A. In each iteration, it adds to A the edge with minimum weight that connects a vertex in A to a vertex in $G.V$ that is not in A. In each iteration, A is an MST for the subgraph of G whose vertices are connected by the edges in A. As soon as all the vertices in $G.V$ are connected by edges in A, A is an MST of G. Note that the algorithm does not work when there are disconnected vertices in the graph and the computed MST may vary depending on the choice of r.

A min-priority queue is commonly used in Prim's algorithm to quickly extract the next minimum edge. For each vertex $v \in G.V$, let $G.Adj[v]$ be a list of adjacent vertices. For each vertex in the minimum priority queue Q, let $v.key$ store the minimum weight of any edge connecting v to a vertex in the tree A (with $v.key = \infty$ if there is no edge), so that function $Extract\text{-}Min(Q)$ returns the vertex associated with the smallest weight. Let $v.\pi$ store the corresponding parent of v in the tree. Then Algorithm 1 describes the steps of Prim's algorithm as given in [53], with r

Algorithm 1 Prim's Algorithm [53]

```
1:  procedure MST-PRIM(G, w, r)
2:      for each u ∈ G.V do
3:          u.key = ∞
4:          u.π = NIL
5:      end for
6:      r.key = 0
7:      Q = G.V
8:      while Q ≠ ∅ do
9:          u = Extract-Min(Q)
10:         for each v ∈ G.Adj[u] do
11:             if v ∈ Q and w(u,v) < v.key then
12:                 v.π = u
13:                 v.key = w(u, v)
14:             end if
15:         end for
16:     end while
17: end procedure
```

being the root of the generated tree A and the structure of A described by the values of $v.\pi$ extracted from the queue Q.

3.4.2 Prim's Algorithm in SPARK at the Silver Level

The Ada language includes two types of subprograms: functions and procedures. A function is a subprogram that returns a value, while a procedure is a subprogram that does not. The implementation of Prim's algorithm presented in this chapter uses two functions: one called $Extract\_Min$ and another called Mst_Prim. Rather than using a minimum priority queue as shown in Algorithm 1, this implementation tracks the visited vertices with a type called $Visited\_Set$, which is an array of Booleans, and uses $Extract\_Min$ to find the next minimum edge that has not yet been visited. The declaration of these functions is given in Figure 6, and the implementations are given in Figure 7 and revised in Figure 9. This example restricts the graph size to only five vertices for simplicity.

```
1   package MST_Prim with SPARK_Mode is
2     Subtype Weight is Integer range 0 .. Integer'Last;
3     Subtype Extended_Vertex is Integer range 0 .. 5;
4     Subtype Vertex is Extended_Vertex range 1 .. 5;
5     type Destinations is array(Vertex) of Vertex;
6     type Weights_List is array (Vertex) of Weight;
7     type MST is record
8       Weights: Weights_List;
9       Edges: Destinations;
10    end record;
11    type Visited_Set is array(Vertex) of Boolean;
12    type Adj_List is array (Vertex) of Weight;
13    type Graph is array (Vertex) of Adj_List;
14
15    function Mst_Prim (G: Graph; r: Vertex) return MST;
16    function Extract_Min (
17      Weights: Weights_List;
18      Visited: Visited_Set
19    ) return Vertex;
20  end MST_Prim;
```

Figure 6: Package Specification mst_prim.ads

Since SPARK is a subset of Ada, it follows the same packaging structure. The code is structured into two files: a specification file with an ".ads" file extension which contains function and parameter declarations of the package and a package body file with an ".adb," file extension which contains the function implementations of the package. The following code is an implementation of Prim's algorithm in SPARK at the stone level, i.e. the code compiles but the results show that it does not pass flow analysis as shown in Figure 8.

Figure 6 contains the package specification. In the specification file, we set types to describe a graph and an MST. We represent a graph as an adjacency matrix that specifies the weights for each edge, and we represent an MST as a record that contains a list of parents and a list of minimum edge weights, each list having one element per vertex. The list of minimum edge weights is stored using a type called $Weights\_List$, which is analogous to the set of $v.key$ values in Algorithm 1, and the list of parents is stored using a type called $Destinations$, which is analogous to the set of $v.\pi$ values. Each list is an array indexed by $Vertex$, so that each index of the array corresponds to a vertex in the graph. The $Vertex$ type in Figure 6 cannot have a value less than 1 or greater than 5. This SPARK feature, which is inherited from Ada, helps with type safety. Strong types in SPARK help clarify the intent of the code and ensure that values are not corrupted by incompatible types during run-time. Type names are also case-insensitive, which adds additional clarity to the code by enforcing that all type names must be unique. If the names are not unique, the code will not compile. For example, a variable cannot be named "vertex" when there is already a type called "Vertex." Bounding a type ensures that out-of-bounds values cannot be assigned to variables of that type without a runtime error during execution. SPARK verification tools can automatically prove that variables of bounded types are never assigned out-of-bounds values as part of the *flow analysis* check.

```
1   package body MST_Prim with SPARK_Mode is
2     function Extract_Min (
3       Weights: Weights_List; Visited: Visited_Set
4     ) return Vertex is
5       min: Weight := Weight'Last;
6       min_Index: Integer;
7     begin
8       min_Index := 1;
9       for I in Vertex loop
10        if Weights(I) < min
11          and Visited(I) = False
12        then
13          min := Weights(I);
14          min_Index := I;
15        end if;
16      end loop;
17      return min_Index;
18    end Extract_Min;
19
20    function Mst_Prim (G: Graph; r: Vertex) return MST
21    is
22      M : MST;
23      Visited: Visited_Set := (others => False);
24      u : Vertex;
25    begin
26      M.Weights(r) := 0;
27      M.Edges(r) := Extended_Vertex'First;
28      for I in Vertex loop
29        u := Extract_Min(M.Edges, Visited);
30        Visited(u) := True;
31        for V in Vertex loop
32          if G(u)(V) > 0 and Visited(V) = False and
33              G(u)(V) < M.Weights(V)
34          then
35            M.Weights(V) := G(u)(V);
36            M.Edges(V) := u;
37          end if;
38        end loop;
39      end loop;
40    return M;
41    end MST_Prim;
42  end MST_Prim;
```

Figure 7: Package Body mst_prim.adb

Figure 7 contains the implementations of the algorithms' functions. For a given initial vertex r, the function MST_Prim initializes the corresponding weight in MST M to zero and the edge to its parent as $Extended\_Vertex'First$ (which is zero) to

represent NIL. At each iteration, the vertex that is reachable with minimum weight from the current tree stored in MST M is selected as the current vertex and is marked as visited in $Visit\_Set$. From here, the function follows the process described in Algorithm 1 and updates the minimum weights to reach each vertex to which the current vertex is connected. Using $Extract\_Min$, the next reachable vertex with the minimum edge weight is selected, and the process of updating the edge weights in the solution MST continues. As a new edge is added, a new partial solution MST is created that contains all the vertices that have been visited so far.

Although the code in the Figure 7 is correct SPARK code (making this a stone-level verification), it does not pass the SPARK flow analysis. Flow analysis verifies that data will flow through variables and subprograms correctly at runtime [2]. In this case, M is not properly initialized, so SPARK analysis provides the output in Figure 8, identifying the fields of the record M that are not properly initialized. Uninitialized variables introduce nondeterminism, which is evident when the program specified in Figure 7 is executed. Without initializing the variable M, the execution of the code results in an incorrect MST. To solve this problem, we simply initialize M as shown in Figure 9, with the minimum edge weights set to the maximum possible value for the type $Integer$ and all the edges for the parents set to 0 (representing NIL).

Once M is initialized, the flow analysis passes and the code constructs a correct MST. This raises the verification level to bronze, and with no run-time errors, it is even considered a silver-level proof. The automatic achievement of silver-level verification

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
mst_prim.adb:35:27: medium: "M.Edges" might not be initialized
mst_prim.adb:39:19: medium: "M.Weights" might not be initialized
mst_prim.adb:46:14: medium: "M.Weights" might not be initialized
mst_prim.adb:46:14: medium: "M.Edges" might not be initialized
```

Figure 8: Console Output After SPARK Analysis of the Code in Figure 6 and Figure 7

```
1       type MST is record
2         Weights: Weights_List := (others => Integer'Last);
3         Edges: Destinations := (others => 0);
4       end record;
```

Figure 9: Initialization of MST Record

in this case is mainly due to the simplicity of the code and the straightforward type definitions. The analysis report in the next section explains what checks were proven to achieve these levels of verification.

3.5 Results

SPARK generates an analysis report that summarizes the checks performed on the code during analysis, including whether or not the checks were successful and which tools or provers were used to discharge them. These details are presented as a table included in the analysis report, as shown in Figure 10 and Figure 11. Each row in the table represents the categories of checks that SPARK performs, and the columns represent which tool was used to discharge each check. When a prover is used to discharge a check, the name of the prover is cited in the provers column. The numbers in the table represent the total number of checks verified by the associated tool. The number of steps needed to prove the checks and a breakdown of the checks by subprogram are also given in the analysis report. A detailed description of the analysis report that includes descriptions of the columns and rows of the table can be found in [25] but the results for this example are presented here.

Figure 10 shows the analysis report for our program when it had an uninitialized variable. When a variable is not initialized, the flow analysis fails. The code analyzed in this case is contained in Figure 6 and Figure 7, where the variable MST in the MST_Prim subprogram is not initialized. After running SPARK with the "prove all" option in this code, 4 checks (50% of all checks) are unproved, as shown in the analysis

report. These four checks are directly related to the warnings given in Figure 8. This means that there is no guarantee that valid values will pass through the variable M in the subprogram Mst_Prim. The column "Unproved" in Figure 10 demonstrates how flow analysis is able to catch initialization issues that are not caught by the compiler. Initializing M as shown in Figure 9 resolves the error, and the new analysis report, presented in Figure 11, shows that all initialization checks are successful. Additionally, the code now passes the flow analysis and is guaranteed to have valid information flow because flow analysis is sound, which means that if the errors it is supposed to catch are not caught, then there are no such errors [2].

Both results summaries show one check discharged by the CVC4 prover. In this case, the $Extract\_Min$ subprogram has been proven to contain no runtime errors when it uses an $Integer$ type for min_Index to return a $Vertex$ type. This means that min_Index will never hold a value outside the range of type $Vertex$, which is a proof that can be automatically performed by SPARK using provers such as CVC4 without the need for additional annotations from the developer.

Proofs performed by the provers are only guaranteed if flow analysis is passing. Therefore, the results in the "Provers" column should be considered only after the flow analysis is complete. For more complex code or code with gold or platinum level functional specifications, it is likely that more effort would be required from the developer, e.g., additional annotations in the code needed to guide the provers (see [20] for an example).

Since all subprograms in this example had 0 unproved checks in all rows above "Functional contracts," the subprograms are verified to the silver level. We consider this an automatic proof because the code did not need to be annotated with preconditions and postconditions, loop invariants, assertions, etc. for all checks to prove. Raising this code to the gold or platinum level would require first writing functional

SPARK Analysis results	Total	Flow	Provers	Unproved
Data Dependencies	.	.	.	.
Flow Dependencies	.	.	.	.
Initialization	6	2	.	4
Non–Aliasing	.	.	.	.
Run–time Checks	1	.	1 (CVC4)	.
Assertions	.	.	.	.
Functional Contracts	.	.	.	.
LSP Verification	.	.	.	.
Termination	.	.	.	.
Concurrency	.	.	.	.
Total	7	2 (29%)	1 (14%)	4 (57%)

```
max steps used for successful proof: 1

Analyzed 2 units
in unit main, 0 subprograms and packages out of 1 analyzed
  Main at main.adb:4 skipped
in unit mst_prim, 3 subprograms and packages out of 3 analyzed
  MST_Prim at mst_prim.ads:1 flow analyzed
  (0 errors, 0 checks and 0 warnings) and proved (0 checks)
  MST_Prim.Extract_Min at mst_prim.ads:23 flow analyzed
  (0 errors, 0 checks and 0 warnings) and proved (1 checks)
  MST_Prim.Mst_Prim at mst_prim.ads:27 flow analyzed
  (0 errors, 4 checks and 0 warnings) and proved (0 checks)
```

Figure 10: SPARK Analysis Report With Uninitialized MST

specifications for each subprogram in the form of preconditions and postconditions and then using SPARK to attempt to prove that they are satisfied. After that, if SPARK is not able to prove that the code satisfies the specifications automatically, then the developer would need to add additional annotations in the code to guide the provers. At the silver level, the code is currently guaranteed to have correct variable initialization and data flow and to be free of run-time errors, which is a key step in demonstrating that the code is highly reliable.

SPARK Analysis results	Total	Flow	Provers	Unproved
Data Dependencies	.	.	.	.
Flow Dependencies	.	.	.	.
Initialization	3	3	.	.
Non–Aliasing	.	.	.	.
Run–time Checks	1	.	1 (CVC4)	.
Assertions	.	.	.	.
Functional Contracts	.	.	.	.
LSP Verification	.	.	.	.
Termination	.	.	.	.
Concurrency	.	.	.	.
Total	4	3 (75%)	1 (25%)	.

```
max steps used for successful proof: 1

Analyzed 2 units
in unit main, 0 subprograms and packages out of 1 analyzed
  Main at main.adb:4 skipped
in unit mst_prim, 3 subprograms and packages out of 3 analyzed
  MST_Prim at mst_prim.ads:1 flow analyzed
  (0 errors, 0 checks and 0 warnings) and proved (0 checks)
  MST_Prim.Extract_Min at mst_prim.ads:23 flow analyzed
  (0 errors, 0 checks and 0 warnings) and proved (1 checks)
  MST_Prim.Mst_Prim at mst_prim.ads:27 flow analyzed
  (0 errors, 0 checks and 0 warnings) and proved (0 checks)
```

Figure 11: SPARK Analysis Report With MST Initialized

3.6 Conclusion

In this chapter, we have given a brief overview of formal methods, with an emphasis on SPARK for formal program verification. We have shown how to use SPARK to develop and formally verify a basic implementation of Prim's algorithm for constructing MSTs, with an explanation of what types of analysis SPARK performs and how the different levels of verification in SPARK are categorized. In this case, the provers automatically formally verified that our implementation is free of data initialization, data flow, and runtime errors. This level of verification provides a solid foundation for reliable code.

In the future, we would like to reimplement the algorithm using a formally verified implementation of a minimum-priority queue such as the one presented in [54] to match Algorithm 1 more closely. Next, we would like to formally verify that our implementation satisfies the functional specifications of the algorithm. This would require writing appropriate specifications for our subprograms in the form of preconditions and postconditions that describe their desired behavior. While SPARK attempts to prove such properties automatically, fully automated proof is generally not feasible, so it is likely that annotations in the form of assertions and loop invariants will be needed to guide the provers. As a starting point, we can leverage work by Möller and Höfner who prove Prim's algorithm by hand (i.e. not with formal methods), but using a proof strategy explicitly designed to facilitate formal program verification of an implementation of the algorithm [55]. Most other proof approaches rely on (1) existence of a minimal spanning tree of the overall graph and (2) properties that rely on reasoning about graph cycles, both of which are hard to reason about in program verification tools. Möller and Höfner's proof strategy establishes invariants at each step of the algorithm, which is much easier to reason about in program verification. In a SPARK implementation, these invariants likely provide the assertions and loop invariants needed to prove that the code satisfies functional specifications. If we are able to complete a proof of full functional correctness using the approaches in [55] and also provide enough additional functionality to form a library, we plan to make our code available as a crate through the new Alire (Ada LIbrary REpository) distribution system. Completing this proof will show that a full functional correctness proof of Prim's algorithm is possible with more-automated tools such as Why3, contrary to Mohan et al.'s prediction that such tools would not be able to prove full functional correctness as easily as their work with VST [48].

Additional research can be done to compare a variety of formal verification tools

using other languages, such as Rust, with the Prusti formal verification tool. The Rust language was built with many of the same security concerns in mind that resulted in the creation of Ada/SPARK. Recently, SPARK has released support for a restricted form of pointers inspired by Rust [56]. This new feature allows for the verification of recursive data structures, demonstrating that as the SPARK language grows, more challenging structures and algorithms can be formally verified using SPARK. Since both the Rust and SPARK languages were developed with similar safety and security concerns in mind, they will probably continue to influence each other. An interesting exercise with these languages would be to use the same specification to generate and verify code in each language to see how the tools, reliability, safety, and security compare. This exercise may give some insight into what the future of formal software verification may look like.

IV. Formal Verification of Functional Properties of Prim's Algorithm in SPARK*

4.1 Introduction

As software systems have become increasingly complex, it has become more and more difficult to ensure their correctness. A major reason is that as the complexity of a system increases, the proportion of system behaviors that can be feasibly covered by standard test-based verification approaches decreases, leaving more room for latent errors. A possible solution to this problem lies in the use of formal methods, i.e., mathematically-based tools and approaches for software and hardware verification [27, 28]. Whereas testing checks individual execution traces of a system, formal methods analyze a mathematical model of a system, opening the possibility of mathematically proving that all possible behaviors of the system are correct. To make an analogy, consider the Pythagorean Theorem. One could simply build confidence in its correctness by testing it against a set of randomly selected right triangles, or one could prove its correctness for all right triangles by applying geometric axioms.

In safety-critical domains, where errors can lead to substantial damage or loss of life, there is a need to eliminate as many errors as possible. Therefore, certification standards for many safety-critical domains promote the use of formal methods, for example, ISO 26262 for the automotive domain [29], EN 50128 for the railway domain [30], and the DO-333 supplement to DO-178C for the aerospace domain [31]. There is a perception that formal methods require significant expertise to use and may not provide a good return on investment [32, 4, 33], so historically the use of formal methods has been concentrated in safety-critical domains. However, the challenge of maintaining software correctness in the face of growing complexity has recently

*This chapter is based on a paper that will be submitted for publication in a journal or conference that is yet to be determined: B. Wheelhouse, L. Humphrey, K. Hopkinson, "Formal Verification of Prim's Algorithm in SPARK," will be submitted for publication.

motivated the use of formal methods in other domains. For example, Amazon Web Services engineers have been using formal methods since 2011 to help solve difficult design problems in systems that use distributed algorithms for data management [34], and Amazon Web Services also uses formal methods to address a variety of cyber security concerns [35, 36, 37]. Despite perceptions that formal methods are difficult and expensive to use, Newcombe et al. (2015) found that "formal methods find bugs in system designs that cannot be found through any other technique we know of," and "Formal methods are surprisingly feasible for mainstream software development and give good return on investment." In fact, they found that using formal methods to write and check proofs of certain types of algorithm was actually faster and easier than doing so by hand. In general, while the use of formal methods does require some investment, the return on investment is eventually realized in terms of better reliability, security, and fewer bugs to fix after development [33].

For formal program verification, one language and toolset that we have found relatively easy to use is SPARK [16]. This is both because the design philosophy of the language emphasizes safe and correct programming, it used to be freely available as part of the GNAT Community Edition [38] and is now freely available under the GNU General Public License v3.0 [57], and there are several educational materials and examples available [2, 39]. In this chapter, we show how to use SPARK to implement and verify certain properties of Prim's algorithm for building minimum spanning trees (MSTs). We choose to focus on this algorithm because MSTs are used in problems involving network reliability, classification, and routing [40, 41] and would benefit from formal verification given the need for reliability in these problem domains. The rest of the chapter proceeds as follows. Section 4.2 discusses a few applications of SPARK in industry as well as efforts to prove Prim's algorithm using various methods, Section 4.3 gives some background on SPARK, Section 4.4 demonstrates how SPARK

is used to develop and verify Prim's algorithm, Section 4.5 addresses the results of the analysis report generated by SPARK, and Section 4.6 offers some conclusions.

4.2 Related Work

Formal methods have been used successfully in a variety of large projects and in proving Prim's algorithm. [21] and [5] provide an overview of how SPARK has been used in various projects, especially projects focused on cyber security and safety in the aerospace domain. These include Ship Helicopter Operating Limits Information System (SHOLIS), the C130J "Hercules" core mission computer, which saw 80% monetary savings in the modified condition/decision coverage (MC/DC) testing budget due to the low number of faults discovered during testing; and the NSA-funded Tokeneer demonstrator, for which testing found zero defects for a period after delivery. In terms of cyber security, the verification of code at the silver level in SPARK proves that the code is free of many of the cyber vulnerabilities classified in the MITRE corporation Common Weakness Enumeration (CWE) database [22]. This is discussed in [23], which provides a mapping of how the language features of SPARK/Ada prevent certain classes of CWEs and how verification with SPARK prevents others.

We briefly note that other tools and frameworks perform analogous types of formal verification for different languages. For example, Frama-C is a framework for analyzing C code in which contracts and assertions are written in ANSI/ISO C Specification Language (ACSL) and plug-ins for formal verification are available [42]. A case study comparing ACSL / Frama-C with SPARK can be found in [43]. There is also Prusti for Rust [44] and Krakatoa [45] for Java, just to name a few.

In regard to Prim's algorithm, some efforts have already been made to apply formal methods. Abrial, Cansell, and Méry give an approach to proving Prim's algorithm using the formal modeling tool Atelier B [46]. Atelier B is an environment

for generating and proving proof obligations for formal models, e.g. of algorithms. Such models can be automatically translated into C, C++, Ada, or HIA code [47], but since errors could be introduced during this translation, additional program verification tools such as SPARK should be used.

Another effort has succeeded in a proof of full functional correctness of an executable implementation of Prim's algorithm written in verifiable C using Coq: CompCert and the Verified Software Toolchain (VST) separation logic deductive verifier [48]. Mohan demonstrates that Prim's algorithm works on disconnected graphs (thus finding a minimum spanning forest (MSF) rather than a MST) and predicts that more automated tools such as Why3 would not be able to prove full functional correctness as easily as their work with VST.

4.3 SPARK

This section provides an overview of SPARK, much of which is a summary of [2]. SPARK is a programming language *and* a formal verification toolset. SPARK as a programming language is based on the Ada programming language. Ada has a number of features that help to support the development of safe and correct programs, which SPARK builds upon. However, SPARK both adds some features that support formal verification and removes some features that make formal verification difficult. In summary, SPARK leverages Ada features, such as

- Type safety
- Ada 2012 *aspects* for writing contracts
- A package system that enables clean separation of interfaces from implementations

and removes features such as

- Aliasing (assigning two names to the same object)
- Exception handlers
- Backward goto statements
- Controlled types
- Side-effects in expressions, including functions

For users who rely on Ada features that are restricted in the SPARK subset, note that, while SPARK can be used to prove an entire program, it can also be applied to only specific parts of a program, including designated lines, subprograms, or packages. Combined with the fact that SPARK is compiled using an Ada compiler, this makes it possible to mix unproven Ada code with restricted features into the program if necessary. The SPARK User's Guide goes into more detail about these restrictions in [52]. The relationship between SPARK and Ada is depicted in Figure 12.

As a static verification toolkit, SPARK verifies code without compiling or executing it. SPARK performs several different types of static analysis. One is *flow analysis*, which checks the initialization of variables, unused assignments, unmodified

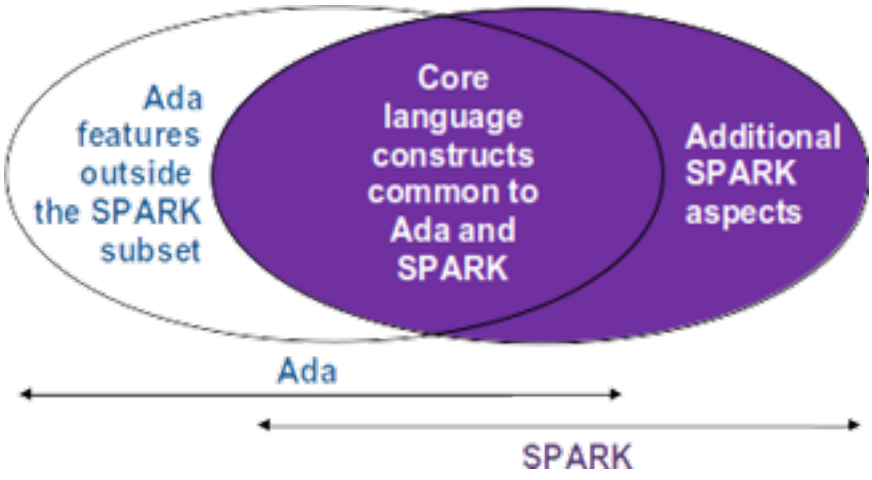

Figure 12: The Relationship Between SPARK and Ada [2].

variables, and data dependencies between the input and output of subprograms. The other is *proof*, which checks for the absence of runtime errors, as well as the conformity of the program with user-defined specifications. To perform these types of analyses, SPARK uses the denotational semantics of the language to translate programs along with checks and contracts to be verified to the Why3 deductive verification platform [17]. Why3 then uses a weakest precondition calculus to generate verification conditions (VCs), i.e., logical formulas whose validity would imply the soundness of the code with respect to its checks and contracts. Why3 then uses multiple theorem provers/satisfiability modulo theories (SMT) solvers to discharge VCs, including CVC4 [49], Alt-Ego [50], and Z3 [51]. Note that while the tools attempt to automate this process, sometimes additional assertions in the code must be provided by the user to guide the underlying provers.

Given the different types of analysis SPARK can perform and the fact that the level of detail in program specifications can vary from partial to complete, some colloquial terms have been adopted by SPARK users to define the level of assurance that has been attained for the code. From lowest to highest, these levels are colloquially referred to as "stone," "bronze" "silver," "gold," and "platinum" [3]. A simple example of a gold level proof can be found in [20], which verifies that a merge sort algorithm satisfies a partial specification. Table 5 gives more details on the levels of SPARK verification and the use cases at each level.

Table 5: Levels of SPARK Verification [3].

Level	**Guarantees**	**Use Case**
Stone	Valid SPARK	Intermediate level during the adoption of SPARK
Bronze	Initialization, correct data flow	As large a part of the code as possible
Silver	Absence of run-time errors (AoRTE)	The default target for critical software (subject to costs and limitations)
Gold	Proof of key integrity properties	Only for a subset of the code subject to specific key integrity (safety/security) properties
Platinum	Full functional proof of requirements	Only for those parts of code with the highest integrity (safety/security) constraints

4.4 Example

Spanning trees can be used in communication protocols to provide paths from one node in the network to other non-neighboring nodes. A spanning tree is a subset of edges in a graph that connect all nodes or vertices in the graph without any cycles, where the number of edges is one less than the number of vertices. For a weighted graph, a *minimum* spanning tree is one whose edge weights have the smallest sum of all possible spanning trees in the graph [53].

Prim's algorithm is one algorithm that can be used to compute an MST for a graph. In this section, we start by describing this algorithm. Then, we present an implementation of this algorithm in SPARK and show how to refine it so that it is proven to a gold level. The implementation in this chapter builds on previous work in which we used SPARK to automatically prove a version of Prim's algorithm to the bronze and silver level [58]. Whereas the previous version only allowed for graphs of a specified size, the version in this chapter allows for any reasonably sized graph. It also improves on the previous version by using a formally verified priority queue in the implementation.

4.4.1 Prim's Algorithm

Prim's algorithm is a greedy algorithm for finding an MST of a weighted undirected graph given a starting vertex. Let $G = (V, E)$ be a weighted undirected graph with vertices V, edges E, and a function $w : E \to \mathbb{R}$ assigning a weight $w(u, v)$ to every edge $(u, v) \in E$. Let us denote the set of vertices as $G.V$. Starting from an arbitrary root vertex $r \in G.V$, Prim's algorithm incrementally builds a tree A. In each iteration, it adds to A the edge with minimum weight that connects a vertex in A to a vertex in $G.V$ that is not in A. A is an MST for the subgraph of G whose vertices are connected by the edges in A [55]. As soon as all the vertices of $G.V$ are connected by the edges of A, A is an MST of G. Note that the algorithm does not work when there are disconnected vertices in the graph, and the computed MST may vary depending on the choice of r.

A min-priority queue is commonly used in Prim's algorithm to quickly extract the next minimum edge. For each vertex $v \in G.V$, let $G.Adj[v]$ be a list of adjacent vertices. For each vertex in the minimum priority queue Q, let $v.key$ store the minimum weight of any edge connecting v to a vertex in the tree A (with $v.key = \infty$ if there is no edge), so that the function $Extract\text{-}Min(Q)$ returns the vertex associated with the smallest weight. Let $v.\pi$ store the corresponding parent of v in the tree. Then Algorithm 2 outlines the steps of Prim's algorithm as given in a well-known algorithms textbook [53], with r being the root of the generated tree A and the structure of A described by the values of $v.\pi$ extracted from the queue Q.

In order to prove Prim's algorithm to the gold or platinum level, a specification must be given to form the requirements the implementation must satisfy to be proven correct. Using Algorithm 2 and the specifications given in [53], we present the following specifications to define several key properties that can be proven in our implementation of Prim's algorithm.

Algorithm 2 Prim's Algorithm [53]

```
1: procedure MST-PRIM(G, w, r)
2:     for each u ∈ G.V do
3:         u.key = ∞
4:         u.π = NIL
5:     end for
6:     r.key = 0
7:     Q = G.V
8:     while Q ≠ ∅ do
9:         u = Extract-Min(Q)
10:        for each v ∈ G.Adj[u] do
11:            if v ∈ Q and w(u,v) < v.key then
12:                v.π = u
13:                v.key = w(u, v)
14:            end if
15:        end for
16:    end while
17: end procedure
```

1. The queue entries must be initialized to the specified values representing an initial priority of ∞ and an initial parent vertex of NIL, as shown in lines 2-4 of Algorithm 2.

2. The queue must initially contain an entry for each vertex in the input graph, as shown on line 6 of Algorithm 2.

3. For each iteration of the loop on lines 8-16 of Algorithm 2 the following properties must hold:

   - The minimum priority value is extracted from the Queue
   - The edges of the growing MST A are within the set

   $$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

   - The vertices already placed in the MST are those in $V - Q$

- For all vertices $v \in Q$, if $v.\pi \neq NIL$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed in the MST

4.4.2 Prim's Algorithm in SPARK at the Gold Level

The purpose of this implementation is to follow Algorithm 2 as closely as possible. To do so, we encode a graph as shown in Figure 14. A graph can be initialized as a set of vertices and a set of edges. We use $Ada.Containers.Formal\_Vectors$, which is part of SPARK's Formal Containers library [59], to hold edges and vertices, so that graphs can be constructed by appending to the appropriate vector. The Formal_Vectors are defined as shown in Figure 13. Our representation of a *Vertex* is intentionally simple, such that a vertex is an integer, specifically the integers from one to the size of the input graph. The subtype *Extended_Vertex* provides an uninitialized value, or NIL, of 0 to initialize *Edge.From* at the beginning of our implementation of Prim's algorithm. We define an *Edge* as a two-element subset of a finite set of vertices. An edge may be denoted as $\{u, v\} \in V \times V$ with $u \neq v$ by uv or vu. *Weights* are mapped directly to *Edges*, so we simply include the weight (W) as a property of an *Edge*.

SPARK does not allow unbounded data structures within records, so the parameters *Num_Vertices* and *Num_Edges* allow the user to define the size of the graph being created in terms of the number of vertices and the number of edges. The parameters *Num_Vertices* and *Num_Edges* are checked by the provers using the *Dynamic_Predicate*, however, this check is redundant because the bounds in the predicate are identical to the bounds of type *Count_Type*. By including the predicate, we can ensure that our specified bounds remain the same even if the bounds of *Count_Type* change.

We use a formally verified implementation of a minimum-priority queue that is

```
1   Subtype Extended_Vertex is Natural range 0 .. 100;
2   Subtype Vertex is Extended_Vertex range 1 ..
        Extended_Vertex'Last;
3   Subtype Weight is Natural
4     range 0 .. 1000;
5   type Edge is record
6     From: Extended_Vertex;
7     To: Vertex;
8     W: Weight;
9   end record;
10    with Dynamic_Predicate =>
11      From /= To;
12
13  package Vertices is new Formal_Vectors(
14      Element_Type => Vertex, Index_Type => Positive);
15  use Vertices;
16  package Edges is new Formal_Vectors(
17      Element_Type => Edge, Index_Type => Positive);
18  use Edges;
```

Figure 13: Vertices, Edges, and Weights

```
1   type Graph(Num_Vertices, Num_Edges: Count_Type) is record
2     V: Vertices.Vector(Num_Vertices);
3     E: Edges.Vector(Num_Edges);
4   end record
5    with Dynamic_Predicate =>
6       Num_Vertices >= 0 and
7       Num_Edges >= 0;
```

Figure 14: Representation of Graph in SPARK

based on the priority queue presented by Baity in [54] with the difference being the data structure used to represent the queue. Our implementation uses a Formal_Vector with an element type of *Queue_Item* shown in Figure 15.

This approach makes the queue more usable for a user, as its functionality is the same as any other formal vector, but with some additional functionality. The added functionality includes *Is_In_Queue*, *Get_Item_Priority*, *Update_Item*, and Baity's verified method for extracting the minimum element *Extract_Min*. Verification of *Extract_Min* was simplified using Formal_Vectors because the library already verifies operations such as deleting an element, which requires shifting the elements in the

```
1       type Queue_Item is record
2         ID: Vertex;
3         Value: Edge;
4         Priority: Weight;
5       end record
6         with Dynamic_Predicate =>
7           ID in Vertex and
8           Value.From in Extended_Vertex and
9           Value.To in Vertex and
10          Value.W = Priority and
11          Priority in Weight;
12
13      Package PQ is new Formal_Vectors(
14        Element_Type => Queue_Item, Index_Type => Positive);
15      use PQ;
```

Figure 15: Min-Priority-Queue Queue_Item

vector to fill the index of the removed element. Figure 16 shows that we still preserve the preconditions and postconditions and the loop invariants provided in the original min-priority queue. This functionality ensures that the minimum weighted edge currently connected to the graph is always given at the beginning of each iteration.

When comparing Figure 17 with Algorithm 2, the similarities are evident. First, a queue is initialized with items for each vertex and a given default priority of infinity. Next, the graph is searched, and the queue is updated by extracting the minimum priority (weight) item from the queue and updating all the weights of the edges connected to the item's corresponding vertex.

Despite the similarities, there are also some apparent differences. Beginning with the input parameters, our implementation accepts an adjacency matrix rather than a graph, as described in the specification. This is due to the simplicity of implementing an adjacency matrix as opposed to the graph implementation described above. Converting a graph to an adjacency matrix or adjacency list is a straightforward exercise but adds some verification complexity that is beyond what we need to prove Prim's algorithm in this example. We also include weights as part of a graph edge rather than as an additional parameter, as shown in Algorithm 2 and pass an empty

```
1       -- declaration in .ads file
2       procedure extract_min(Q: in out PQ.Vector;
3                             pair: out Queue_Item)
4        with
5          pre =>
6            not PQ.Is_Empty(Q) and then
7            Integer(Length(Q)) in Vertex,
8          Post =>
9              Did_Queue_Decrease(Q'Old, Q) and
10             Is_Min(Q, pair) and
11             Is_First_Extracted(Q'Old, Q, pair);
12
13      -- definition in .adb file
14      procedure extract_min(Q: in out PQ.Vector;
15                            pair: out Queue_Item)
16      is
17        min_index: Vertex := 1;
18      begin
19        for I in 1 .. Integer(Length(Q)) loop
20           if Q.Element(I).Priority < Q.Element(min_index).
                Priority
21           then
22              min_index := I;
23           end if;
24           pragma Loop_Invariant(min_index in 1 .. I);
25           pragma Loop_Invariant(for all E in 1 .. I =>
26                   Q.Element(E).Priority >=
27                   Q.Element(min_index).Priority);
28           pragma Loop_Invariant(
29              for all E in 1 .. min_index-1 =>
30                   Q.Element(E).Priority >
31                   Q.Element(min_index).Priority);
32        end loop;
33        pair := Q.Element(min_index);
34        PQ.Delete(Q, min_index);
35      end extract_min;
```

Figure 16: Min-Priority-Queue Extract_Min

initialized graph to be able to construct it and return it as an MST.

When an item is extracted from the queue, we prove that the edge of the resulting item is the minimum edge in the loop-invariant directly following the call to *Extract_Min*. Having obtained the minimum edge, we add the edge and a new vertex to the resulting MST, as demonstrated in Figure 18.

```
1      procedure Prim_MST(G: Adj_Matrix; r: Integer;
2                         MST: in out Graph)
3      is
4        u : Queue_Item;
5        w : Weight;
6        Q : PQ.Vector(G'Length);
7      begin
8        for I in G'Range(1) loop
9           Q.Append((I,(0,I,Weight'Last),Weight'Last));
10          pragma Loop_Invariant(
11             Integer(Length(Q)) = I and
12             I <= Integer(Capacity(Q)));
13          pragma Loop_Invariant(for all J in 1 .. I =>
14                       Q.Element(J).ID = J and
15                       Q.Element(J).Value.To = J);
16       end loop;
17       Update_Item(Q, r, 0, r);
18       while Is_Not_Empty(Q) loop
19          extract_min(Q, u);
20          pragma Loop_Invariant(for all I of Q =>
21             u.Priority <= I.Priority);
22          -- Construct MST
23          for I in 1 .. G'Length loop
24             pragma Assume(u.Value.To <= G'Length);
25             w := G(u.Value.To, I);
26             pragma Assume(Is_In_Queue(Q, I));
27             if w > 0 and I /= u.Value.To and
28                w < Get_Item_Priority(Q, I)
29             then
30                Update_Item(Q,(I,(u.Value.To,I,w),w));
31             end if;
32          end loop;
33       end loop;
34     end Prim_MST;
```

Figure 17: SPARK Implementation of Prim's Algorithm

```
1   if Length(MST.V) < Capacity(MST.V) then
2      MST.V.Append(u.Value.To);
3   end if;
4   if u.X.From /= r and Length(MST.E) < Capacity(MST.E) and
5     Length(MST.W) < Capacity(MST.W)
6   then
7      MST.E.Append((u.Value.From, u.Value.To, u.Priority));
8   end if;
```

Figure 18: MST Construction

4.5 Results

We have presented a more refined and dynamic min-priority queue based on Baity's work in [54], which allowed us to prove the key property of having the current minimum edge returned from the queue at each iteration of Prim's algorithm.

SPARK generates an analysis report that summarizes the checks performed on the code during analysis, including whether or not the checks were successful and which tools or provers were used to discharge them. These details of the checks performed on *Prim_MST* and the types associated with it, i.e., *Graph, Adj_matrix* and *Formal_Vectors*, are presented as a table included in the analysis report shown in Figure 19. Each row in the table represents the categories of checks that SPARK performs, and the columns represent the tool used to discharge each check. When a prover is used to discharge a check, the name of the prover is cited in the provers column. Proofs performed by the provers are only guaranteed if flow analysis is passing. Therefore, the results in the *Provers* column should be considered only after the flow analysis is complete. The numbers in the table represent the total number of checks verified by the associated tool. A detailed description of the analysis report, including descriptions of the columns and rows of the table, can be found in [25] but the results for this example are presented here.

Reviewing the table in Figure 19, the *Data Dependencies* checks verify that the Formal_Vectors (Vertices, Edges, and Weights) are read and written properly in the subprogram *Prim_MST* according to the Formal_Vectors specification. These checks also allow GNATprove to assist in proving flow dependencies, initialization, and other proof checks related to the Formal_Vectors. The three *Initialization* checks belong to the variables *u, w,* and *Q* in Prim_MST. Since these checks pass, these variables will be initialized properly at runtime and will not have garbage values that could make the resulting MST unreliable. *Run-time Checks* include overflow, range, and index

checks. In this case, 147 run-time checks are proven, including range, index, dynamic accessibility, pointer dereference, and overflow checks. Finally, the report shows 93 proven *Functional Contracts*. Functional contracts include predicates and pre- and post-conditions that help prove that the code meets the specification requirements. An example of a functional contract proven in this project is shown in Figure 16 where the declaration provides the preconditions and postconditions which must be true to prove that the subprogram runs as intended. Specifically, the preconditions specify that the queue must not be empty and the length of the queue is in the range of the Vertex type, and the postconditions specify that an item was removed from the queue, shortening the queue by one, and that the removed item is the minimum item, and the item was to be the first extracted. Other functional contracts are declared in other parts of the source code, such as in the Formal_Vectors library.

Since this example had 0 unproved checks in all rows above and including *Functional contracts*, this implementation is verified to the gold level. Raising this code to a higher gold or platinum level would require implementing more functional contracts

SPARK Analysis results	Total	Flow	Provers	Unproved
Data Dependencies	4	4	.	.
Flow Dependencies	.	.	.	.
Initialization	4	4	.	.
Non–Aliasing	.	.	.	.
Run–time Checks	147	.	147 (CVC4 95%, Trivial 5%)	.
Assertions	12	.	12 (CVC4)	.
Functional Contracts	93	.	93 (CVC4 80%, Trivial 20%)	.
LSP Verification	.	.	.	.
Termination	.	.	.	.
Concurrency	.	.	.	.
Total	260	8 (3%)	252 (97%)	.

Figure 19: SPARK Analysis Report on Prim_MST

based on the functional specifications listed in Section 4.4.1 in the form of loop invariants, predicates, preconditions and postconditions, then using SPARK to attempt to prove that they are satisfied. After that, if SPARK is not able to prove that the code satisfies the specifications automatically, then the developer would need to add additional annotations in the code to guide the provers (see [20] for an example). At the gold level, the code is currently guaranteed to have correct variable initialization, data flow, no run-time errors, and has proven and functionally correct key properties, which is a key step in demonstrating that the code is highly reliable.

4.6 Conclusion

In this chapter, we have given a brief overview of formal methods with an emphasis on SPARK for formal program verification. We have shown how to use SPARK to develop and formally verify an implementation of Prim's algorithm for constructing MSTs, with an explanation of what types of analysis SPARK performs and how the different levels of verification in SPARK are categorized. In this case, we formally verified that our implementation is free of variable initialization, data flow, and run-time errors and has proven key functional properties within the graph and priority queue on which the algorithm relies. This level of verification provides a solid foundation for reliable code.

In the future, we would like to formally verify that our implementation satisfies other key properties defined in the functional specifications of the algorithm in Section 4.4.1, such as each iteration of Prim's algorithm adding a vertex that is not in the current MST but is connected to a vertex that is in the MST. This will require writing additional loop invariants and additional preconditions and postconditions that describe their desired behavior. If we are able to complete a proof of full functional correctness with respect to the properties in the specification and also provide

enough additional functionality to form a graph library, we plan to make our code available as a crate through the new Alire (Ada LIbrary REpository) distribution system. Completing this proof will show that a full functional correctness proof of Prim's algorithm is possible with more-automated tools such as Why3, although it will probably still require a significant amount of manual proof, supporting Mohan's prediction that Why3 would not be able to prove full functional correctness automatically [48]. We suspect that a proof of a stronger property may be attained by leveraging elements of a purely algebraic proof of Prim's algorithm presented by Höfner and Möller in [55] but such a proof would probably be very complex in SPARK. Performance factors such as an optimized min-priority queue that uses a formally verified heap sort algorithm or using an adjacency list instead of an adjacency matrix can also be considered.

V. Conclusions

This thesis provides three examples demonstrating the benefits and features of safe and reliable software with an emphasis on SPARK and formal verification. The contributions of each example are summarized in the results of the following research questions.

> *RQ1: What are common attributes of highly reliable programming languages and how do those attributes and formal verification enhance the quality of software?*

This question is addressed in Chapter II by comparing the similarities and differences between SPARK and Rust. Although the differences that are identified are primarily stylistic, since SPARK is a language designed specifically to be verifiable and Rust primarily focuses on data safety, the similarities suggest that strict type safety, static typing, anti-aliasing, and memory checks help users develop more secure code. Formal verification can ensure that code is reliable by mathematically proving that codes satisfies properties described in its specifications. Combining a security-focused language and formal verification tools makes creating safe and reliable software available to more developers with varying experience without too much training.

With an understanding of the benefits of what a highly reliable language provides, Chapter III gives a tutorial of the basic formal verification of a naive implementation of Prim's algorithm. This proof of Prim's demonstrates how simple it can be to verify code to a silver level and addresses the question:

> *RQ2: What qualities does an automatic proof of Prim's algorithm have?*

This question is addressed by showing that even without preconditions and post-conditions, loop invariants, and other annotations, which may be considered difficult, the code can be proven to be free of data initialization, data flow, and run-time errors. These are the qualities and characteristics of any silver-level proof in SPARK. These findings from Chapter III are encouraging because they suggest that beginners to software verification can still verify meaningful properties in their code in a short period of time.

Building on the naive implementation and proof of Prim's algorithm in Chapter III, Chapter IV provides an answer to the following question.

> *RQ3: How do you level up basic SPARK code to a meaningful proof of functional correctness for Prim's algorithm?*

By using a clearly defined specification for Prim's algorithm from a textbook [53], a formally verified min-priority queue, and incorporating contracts, loop invariants, and other annotations to prove stronger properties of Prim's algorithm, a *gold* level proof was implemented. This implementation ensures that the minimum edge is always removed from the queue at each iteration. Additionally, the code provides a solid foundation for proving stronger properties of the specification that were not attained in this work.

In summary, this thesis has made three main contributions:

- Provides an overview of safe and reliable programming languages, namely SPARK and Rust, and the benefits of their characteristics from a formal verification perspective.
- Demonstrates the simplicity and usability of formal verification in SPARK on a naive implementation of Prim's algorithm.

- Emphasizes the importance of having a well-defined specification by demonstrating how to increase basic silver level SPARK code to a gold level SPARK code.

5.1 Future Work

A natural direction for future research as a continuation of this thesis is to continue to formally verify graph algorithms and properties in SPARK. This vein of research is a great starting place for those that are beginning to investigate formal verification because graph algorithms generally have a well-defined specification. Eventually, this work could grow into a formally verified graph library, which could be a valuable contribution to enabling the development of more complex and formally verified software.

Appendix A. Quaternion: Full Source Code

Listing A.1: main.adb

```
1  with Quaternion; use Quaternion;
2  with Vector; use Vector;
3  with Ada.Text_IO; use Ada.Text_IO;
4  with Math_Utils; use Math_Utils;
5  with Ada.Numerics; use Ada.Numerics;
6  with Ada.Numerics.Elementary_Functions;
7  use  Ada.Numerics.Elementary_Functions;
8
9  procedure main with SPARK_Mode is
10    Q: Spark_Quaternion := Spark_Quaternion'(
11          1.0, Vector3'(1.0, 2.0, 3.0));
12    V: Vector3;
13    S: Float;
14    N1: Float := -Float'First;
15 begin
16    Put_Line("Float'First: " & N1'Image);
17    Put_Line("Magnitude: " & Magnitude(Q)'Image &
18          " === " & Sqrt(15.0)'Image);
19    V := Get_Q_Vector(Q);
20    Put_Line("Get Vector: " & V.X'Image & ", " &
21          V.Y'Image & ", " & V.Z'Image);
22    S := Get_Q_Scalar(Q);
23    Put_Line("Get Scalar: " & S'Image);
24    Put_Line("Running not operator on Quaternion...");
```

```
25    Q := not Q;
26    V := Get_Q_Vector(Q);
27    Put_Line("Result Vector: " & V.X'Image & ", " &
28         V.Y'Image & ", " & V.Z'Image);
29    Q := not Q;
30    Put_Line ("Get angle: " & Get_Q_Angle(Q)'Image &
31         " === 2.0");
32    Q := Spark_Quaternion'(2.0, Vector3'(2.0, 2.0, 2.0));
33    Put_Line("New Q is (2.0, 2.0, 2.0, 2.0)");
34    V := Get_Q_Axis(Q);
35    Put_Line ("Get Axis: " & V.X'Image & ", " &
36         V.Y'Image & ", " & V.Z'Image);
37    Q := Spark_Quaternion'(
38         1.0, Vector3'(0.00000000000000000001,
39                       0.00000000000000000001,
40                       0.00000000000000000001));
41    Put_Line("ew Q is (1.0,
42                        0.00000000000000000001,
43                        0.00000000000000000001,
44                        0.00000000000000000001)");
45    V := Get_Q_Axis(Q);
46    Put_Line ("Get Axis: " & V.X'Image & ", " &
47         V.Y'Image & ", " & V.Z'Image);
48
49
50    Put_Line("New Q is (1.0, 1.0, 2.0, 3.0)");
```

```
51     Q := Spark_Quaternion'(1.0, Vector3'(1.0, 2.0, 3.0));
52     Q := Rotate_Q(Q, Q);
53     Put_Line ("Rotate Q: N=" & Q.N'Image &
54          " -- X, Y, Z = " & Q.V.X'Image & ", " &
55          Q.V.Y'Image & ", " & Q.V.Z'Image);
56
57
58     Put_Line("New Q is (1.0, 1.0, 2.0, 3.0)");
59     Q := Spark_Quaternion'(1.0, Vector3'(1.0, 2.0, 3.0));
60     V := Vector3'(1.0, 1.0, 1.0);
61     V := Q_V_Rotate(Q, V);
62     Put_Line ("Rotate Q by V: " & V.X'Image & ", " &
63          V.Y'Image & ", " & V.Z'Image);
64
65
66     Q := Make_Q_From_Euler_Angles(1.0, 2.0, 3.0);
67     Put_Line ("Q from Euler (1.0, 2.0, 3.0): N=" &
68          Q.N'Image & " -- X, Y, Z = " & Q.V.X'Image &
69          ", " & Q.V.Y'Image & ", " & Q.V.Z'Image);
70
71     V := Make_Euler_Angles_From_Q(Q);
72     Put_Line ("Euler from Q: " & V.X'Image & ", " &
73          V.Y'Image & ", " & V.Z'Image);
74
75 end main;
```

Listing A.2: constants.ads

```
1 package Constants with SPARK_Mode is
2    -- numerical constants
3    TOL: constant := 0.000000000000001;
4 end Constants
```

Listing A.3: math_utils.ads

```
1 with Constants; use Constants;
2 with Vector; use Vector;
3 with Ada.Numerics; use Ada.Numerics;
4 with Ada.Numerics.Elementary_Functions;
5 use  Ada.Numerics.Elementary_Functions;
6
7 package Math_Utils with SPARK_Mode is
8    subtype Deg_Angle is Float range 0.0 .. 360.0;
9    subtype Rad_Angle is Float range 0.0 .. 2.0 * Pi;
10   --"The bounds for the input values can come
11   -- from the application context..."
12   function Degrees_To_Radians(deg: Deg_Angle)
13        return Rad_Angle;
14   function Radians_To_Degrees(rad: Rad_Angle)
15        return Deg_Angle;
16 end Math_Utils;
```

Listing A.4: math_utils.adb

```
1 package body Math_Utils with SPARK_Mode is
```

```
2     function Degrees_To_Radians(deg: Deg_Angle)
3           return Rad_Angle is
4     begin
5        return deg *  Pi / 180.0;
6     end Degrees_To_Radians;
7
8     function Radians_To_Degrees(rad: Rad_Angle)
9           return Deg_Angle is
10    begin
11       return rad * 180.0 / Pi;
12    end Radians_To_Degrees;
13 end Math_Utils;
```

Listing A.5: quaternion.ads

```
1  with Constants; use Constants;
2  with Vector; use Vector;
3  with Math_Utils; use Math_Utils;
4  with Ada.Numerics; use Ada.Numerics;
5  with Ada.Numerics.Elementary_Functions;
6  use  Ada.Numerics.Elementary_Functions;
7
8  package Quaternion with SPARK_Mode is
9     --General Purpose Quaternion----------------------------
10    type Spark_Quaternion is record
11       N: Scalar := 0.0;  -- Scalar Part
12       V: Vector3;       -- Vector Part
```

```
13    end record;
14
15    --Methods-------------------------------------------
16    function Magnitude(Q: Spark_Quaternion) return Scalar;
17    function Get_Q_Vector(Q: Spark_Quaternion)
18         return Vector3;
19    function Get_Q_Scalar(Q: Spark_Quaternion)
20         return Scalar;
21    function Get_Q_Angle(Q: Spark_Quaternion)
22         return Rad_Angle;
23    function Get_Q_Axis(Q: Spark_Quaternion)
24         return Vector3;
25    function Rotate_Q(Q1, Q2: Spark_Quaternion)
26         return Spark_Quaternion;
27    function Q_V_Rotate(Q: Spark_Quaternion; V: Vector3)
28         return Vector3;
29    function Make_Q_From_Euler_Angles(X, Y, Z: Scalar)
30         return Spark_Quaternion;
31    function Make_Euler_Angles_From_Q(Q: Spark_Quaternion)
32         return Vector3;
33
34    --Operators-----------------------------------------
35    function "not" (Q: Spark_Quaternion)
36         return Spark_Quaternion;
37    function "+" (LEFT, RIGHT: Spark_Quaternion)
38         return Spark_Quaternion;
```

```
39     function "-" (LEFT, RIGHT: Spark_Quaternion)
40          return Spark_Quaternion;
41     function "*" (LEFT, RIGHT: Spark_Quaternion)
42          return Spark_Quaternion;
43     function "*" (LEFT: Spark_Quaternion; RIGHT: Scalar)
44          return Spark_Quaternion;
45     function "*" (LEFT: Scalar; RIGHT: Spark_Quaternion)
46          return Spark_Quaternion;
47     function "*" (LEFT: Spark_Quaternion; RIGHT: Vector3)
48          return Spark_Quaternion;
49     function "*" (LEFT: Vector3; RIGHT: Spark_Quaternion)
50          return Spark_Quaternion;
51     function "/" (LEFT: Spark_Quaternion; RIGHT: Scalar)
52          return Spark_Quaternion;
53 end Quaternion;
```

Listing A.6: quaternion.adb

```
1  package body Quaternion with SPARK_Mode is
2     --Methods-----------------------------------------------
3     function Magnitude(Q: Spark_Quaternion)
4          return Scalar is
5     begin
6        return Sqrt (Q.N * Q.N + Q.V.X * Q.V.X +
7                     Q.V.Y * Q.V.Y + Q.V.Z * Q.V.Z);
8     end Magnitude;
9
```

```
10    function Get_Q_Vector(Q: Spark_Quaternion)
11          return Vector3 is
12    begin
13       return Q.V;
14    end Get_Q_Vector;
15
16    function Get_Q_Scalar(Q: Spark_Quaternion)
17          return Scalar is
18    begin
19       return Q.N;
20    end Get_Q_Scalar;
21
22    function Get_Q_Angle(Q: Spark_Quaternion)
23          return Rad_Angle is
24    begin
25       return 2.0 * Arccos (Cos(Q.N));
26    end Get_Q_Angle;
27
28    function Get_Q_Axis(Q: Spark_Quaternion)
29          return Vector3
30    is
31       V: Vector3 := Q.V;
32       M: Scalar := Magnitude(V);
33    begin
34       if M <= TOL then
35          return Vector3'(0.0, 0.0, 0.0);
```

```
36        end if;
37        return V/M;
38     end Get_Q_Axis;
39
40     function Rotate_Q(Q1, Q2: Spark_Quaternion)
41           return Spark_Quaternion is
42     begin
43        return Q1 * Q2 * (not Q1);
44     end Rotate_Q;
45
46     function Q_V_Rotate(Q: Spark_Quaternion; V: Vector3)
47           return Vector3
48     is
49        T: Spark_Quaternion := Q * V * (not Q);
50     begin
51        return T.V;
52     end Q_V_Rotate;
53
54     function Make_Q_From_Euler_Angles(X, Y, Z: Scalar)
55           return Spark_Quaternion
56     is
57        Roll: Scalar := Degrees_To_Radians(X);
58        Pitch: Scalar := Degrees_To_Radians(Y);
59        Yaw: Scalar := Degrees_To_Radians(Z);
60
61        Cyaw: Scalar := Cos (0.5 * Yaw);
```

```
62       Cpitch: Scalar := Cos (0.5 * Pitch);
63       Croll: Scalar := Cos (0.5 * Roll);
64
65       Syaw: Scalar := Sin (0.5 * Yaw);
66       Spitch: Scalar := Sin (0.5 * Pitch);
67       Sroll: Scalar := Sin (0.5 * Roll);
68
69       CyawCpitch: Scalar := Cyaw * Cpitch;
70       SyawSpitch: Scalar := Syaw * Spitch;
71       CyawSpitch: Scalar := Cyaw * Spitch;
72       SyawCpitch: Scalar := Syaw * Cpitch;
73    begin
74       return Spark_Quaternion'(
75         CyawCpitch * Croll + SyawSpitch * Sroll,
76         Vector3'(CyawCpitch * Sroll - SyawSpitch * Croll,
77         CyawSpitch * Croll + SyawCpitch * Sroll,
78         SyawCpitch * Croll - CyawSpitch * Sroll));
79    end Make_Q_From_Euler_Angles;
80
81    function Make_Euler_Angles_From_Q(Q: Spark_Quaternion)
82         return Vector3
83    is
84       R12: Scalar;
85       R13: Scalar;
86
87       Q00: Scalar := Q.N * Q.N;
```

```
88        Q11: Scalar := Q.V.X * Q.V.X;
89        Q22: Scalar := Q.V.Y * Q.V.Y;
90        Q33: Scalar := Q.V.Z * Q.V.Z;
91
92        R11: Scalar := Q00 + Q11 - Q22 - Q33;
93        R21: Scalar := 2.0 * (Q.V.X * Q.V.Y + Q.N * Q.V.Z);
94        R31: Scalar := 2.0 * (Q.V.X * Q.V.Z - Q.N * Q.V.Y);
95        R32: Scalar := 2.0 * (Q.V.Y * Q.V.Z + Q.N * Q.V.X);
96        R33: Scalar := Q00 - Q11 - Q22 + Q33;
97
98        Tmp: Scalar := Abs (R31);
99     begin
100       if Tmp > 0.999999 then
101          R12 := 2.0 * (Q.V.X * Q.V.Y - Q.N * Q.V.Z);
102          R13 := 2.0 * (Q.V.X * Q.V.Z - Q.N * Q.V.Y);
103          return Vector3'(
104             Radians_To_Degrees(0.0),         --roll
105             Radians_To_Degrees(
106                       -(Pi/2.0) * R31/Tmp), --pitch
107             Radians_To_Degrees(Arctan(
108                       -R12, (-R31 * R13)))  --yaw
109          );
110
111       end if;
112       return Vector3'(
113             Radians_To_Degrees(Arctan(R32, R33)),  --roll
```

```
114             Radians_To_Degrees(Arcsin(Sin(-R31))), --pitch
115             Radians_To_Degrees(Arctan(R21, R11)));  --yaw
116     end Make_Euler_Angles_From_Q;
117
118
119     --Operators-------------------------------------------
120
121     function "not" (Q: Spark_Quaternion)
122          return Spark_Quaternion is
123     begin
124        return Spark_Quaternion'(Q.N, -Q.V);
125     end "not";
126
127     function "+" (LEFT, RIGHT: Spark_Quaternion)
128          return Spark_Quaternion is
129     begin
130        return Spark_Quaternion'(LEFT.N + RIGHT.N,
131                                 (LEFT.V + RIGHT.V));
132     end "+";
133
134     function "-" (LEFT, RIGHT: Spark_Quaternion)
135          return Spark_Quaternion is
136     begin
137        return Spark_Quaternion'(LEFT.N - RIGHT.N,
138                                 (LEFT.V - RIGHT.V));
139     end "-";
```

```
140
141     function "*" (LEFT, RIGHT: Spark_Quaternion)
142          return Spark_Quaternion is
143     begin
144        return Spark_Quaternion'(
145              LEFT.N * RIGHT.N - LEFT.V.X * RIGHT.V.X -
146                  LEFT.V.Y * RIGHT.V.Y -
147                  LEFT.V.Z * RIGHT.V.Z,
148              Vector3'(
149                   LEFT.N * RIGHT.V.X + LEFT.V.X *
150                      RIGHT.N + LEFT.V.Y * RIGHT.V.Z -
151                      LEFT.V.Z * RIGHT.V.Y,
152                    LEFT.N * RIGHT.V.Y + LEFT.V.Y *
153                      RIGHT.N + LEFT.V.Z * RIGHT.V.X -
154                      LEFT.V.X * RIGHT.V.Z,
155                    LEFT.N * RIGHT.V.Z + LEFT.V.Z *
156                      RIGHT.N + LEFT.V.X * RIGHT.V.Y -
157                      LEFT.V.Y * RIGHT.V.X));
158     end "*";
159
160     function "*" (LEFT: Spark_Quaternion; RIGHT: Scalar)
161          return Spark_Quaternion is
162     begin
163        return Spark_Quaternion'(
164              LEFT.N * RIGHT, LEFT.V * RIGHT);
165     end "*";
```

```
166
167     function "*" (LEFT: Scalar; RIGHT: Spark_Quaternion)
168          return Spark_Quaternion is
169     begin
170        return Spark_Quaternion'(
171              LEFT * RIGHT.N, RIGHT.V * LEFT);
172     end "*";
173
174     function "*" (LEFT: Spark_Quaternion; RIGHT: Vector3)
175          return Spark_Quaternion is
176     begin
177        return Spark_Quaternion'(
178              -(LEFT.V.X * RIGHT.X + LEFT.V.Y *
179                RIGHT.Y + LEFT.V.Z * RIGHT.Z),
180              Vector3'(
181                  LEFT.N * RIGHT.X + LEFT.V.Y *
182                      RIGHT.Z - LEFT.V.Z * RIGHT.Y,
183                  LEFT.N * RIGHT.Y + LEFT.V.Z *
184                      RIGHT.X - LEFT.V.X * RIGHT.Z,
185                  LEFT.N * RIGHT.Z + LEFT.V.X *
186                      RIGHT.Y - LEFT.V.Y * RIGHT.X));
187     end "*";
188
189     function "*" (LEFT: Vector3; RIGHT: Spark_Quaternion)
190          return Spark_Quaternion is
191     begin
```

```
192        return Spark_Quaternion'(
193               -(RIGHT.V.X * LEFT.X + RIGHT.V.Y *
194                 LEFT.Y + RIGHT.V.Z * LEFT.Z),
195               Vector3'(
196                   RIGHT.N * LEFT.X + RIGHT.V.Y *
197                       LEFT.Z - RIGHT.V.Z * LEFT.Y,
198                   RIGHT.N * LEFT.Y + RIGHT.V.Z *
199                       LEFT.X - RIGHT.V.X * LEFT.Z,
200                   RIGHT.N * LEFT.Z + RIGHT.V.X *
201                       LEFT.Y - RIGHT.V.Y * LEFT.X));
202     end "*";
203
204     function "/" (LEFT: Spark_Quaternion; RIGHT: Scalar)
205          return Spark_Quaternion is
206     begin
207        return Spark_Quaternion'(
208          LEFT.N/RIGHT, LEFT.V/RIGHT);
209     end "/";
210 end Quaternion;
```

Listing A.7: vector.ads

```
1 with Constants; use Constants;
2 with Ada.Numerics; use Ada.Numerics;
3 with Ada.Numerics.Elementary_Functions;
4 use  Ada.Numerics.Elementary_Functions;
5
```

```
6  package Vector with SPARK_Mode is
7     subtype Scalar is Float range -2.0**24 .. 2.0**24;
8     --"In general large positive and negative
9     -- floating-point inputs need to be excluded"
10
11    --General Purpose 3-element vector---------------------
12    type Vector3 is record
13       X: Scalar := 0.0;
14       Y: Scalar := 0.0;
15       Z: Scalar := 0.0;
16    end record;
17
18    --Methods-----------------------------------------------
19    function Magnitude(V: Vector3) return Scalar;
20    function Normalize(V: Vector3) return Vector3;
21    function Cross_Product(U, V: Vector3) return Vector3;
22
23    --Operators---------------------------------------------
24    function "+" (LEFT, RIGHT: Vector3) return Vector3;
25    function "-" (LEFT, RIGHT: Vector3) return Vector3;
26    function "*" (LEFT, RIGHT: Vector3) return Scalar;
27    function "/" (LEFT: Vector3; RIGHT: Scalar)
28         return Vector3;
29    function "*" (LEFT: Vector3; RIGHT: Scalar)
30         return Vector3;
31    function "-" (V: Vector3) return Vector3;
```

```
32 end Vector;
```

Listing A.8: vector.adb

```
1  package body Vector with SPARK_Mode is
2     --Methods-----------------------------------------------
3     function Magnitude(V: Vector3) return Scalar is
4     begin
5        return Sqrt (V.X * V.X + V.Y * V.Y + V.Z * V.Z);
6     end Magnitude;
7
8     function Normalize(V: Vector3) return Vector3 is
9        M : Scalar := Magnitude(V);
10       Return_V : Vector3;
11    begin
12       if M <= TOL then
13          M := 1.0;
14       end if;
15       Return_V.X := V.X / M;
16       Return_V.Y := V.Y / M;
17       Return_V.Z := V.Z / M;
18       if abs (V.X) < TOL then
19          Return_V.X := 0.0;
20       end if;
21       if abs (V.Y) < TOL then
22          Return_V.Y := 0.0;
23       end if;
```

```
24          if abs (V.Z) < TOL then
25             Return_V.Z := 0.0;
26          end if;
27          return Return_V;
28       end Normalize;
29
30       function Cross_Product(U, V: Vector3) return Vector3 is
31       begin
32          return Vector3'(U.Y * V.Z - U.Z * V.Y,
33                          -U.X * V.z + U.Z * V.X,
34                          U.X * V.Y - U.Y * V.X);
35       end Cross_Product;
36
37       --Operators-----------------------------------------
38       function "+" (LEFT, RIGHT: Vector3) return Vector3 is
39       begin
40          return Vector3'(LEFT.X + RIGHT.X,
41                          LEFT.Y + RIGHT.Y,
42                          LEFT.Z + RIGHT.Z);
43       end "+";
44
45       function "-" (LEFT, RIGHT: Vector3) return Vector3 is
46       begin
47          return Vector3'(LEFT.X - RIGHT.X,
48                          LEFT.Y - RIGHT.Y,
49                          LEFT.Z - RIGHT.Z);
```

```
50    end "-";
51
52    function "*" (LEFT, RIGHT: Vector3) return Scalar is
53    begin
54       return LEFT.X * RIGHT.X + LEFT.Y * RIGHT.Y +
55         LEFT.Z * RIGHT.Z;
56    end "*";
57
58    function "/" (LEFT: Vector3; RIGHT: Scalar)
59         return Vector3 is
60    begin
61       return Vector3'(LEFT.X / RIGHT,
62                       LEFT.Y / RIGHT,
63                       LEFT.Z / RIGHT);
64    end "/";
65
66    function "*" (LEFT: Vector3; RIGHT: Scalar)
67         return Vector3 is
68    begin
69       return Vector3'(LEFT.X * RIGHT,
70                       LEFT.Y * RIGHT,
71                       LEFT.Z * RIGHT);
72    end "*";
73
74    function "-" (V: Vector3) return Vector3 is
75    begin
```

```
76         return Vector3'(-V.X, -V.Y, -V.Z);
77      end "-";
78   end Vector;
```

Appendix B. Naive Prim's Algorithm Implementation: Full Source Code

Listing B.1: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with MST_Prim; use MST_Prim;
3
4  procedure Main is
5     --  G: MST_Prim.Graph := ((0, 2, 0, 6, 0),
6     --                        (2, 0, 3, 8, 5),
7     --                        (0, 3, 0, 0, 7),
8     --                        (6, 8, 0, 0, 9),
9     --                        (0, 5, 7, 9, 0));
10    G: MST_Prim.Graph :=
11    -------a---b---c---d---e---f---g---h---i-------+
12          ((0,  4,  0,  0,  0,  0,  0,  8,  0),---- a
13           (4,  0,  8,  0,  0,  0,  0,  11, 0),---- b
14           (0,  8,  0,  7,  0,  4,  0,  0,  2),---- c
15           (0,  0,  7,  0,  9,  14, 0,  0,  0),---- d
16           (0,  0,  0,  9,  0,  10, 0,  0,  0),---- e
17           (0,  0,  4,  14, 10, 0,  2,  0,  0),---- f
18           (0,  0,  0,  0,  0,  2,  0,  1,  6),---- g
19           (8,  11, 0,  0,  0,  0,  1,  0,  7),---- h
20           (0,  0,  2,  0,  0,  0,  6,  7,  0));--- i
21    mst : MST_Prim.MST;
22 begin
23    mst := MST_Prim.Mst_Prim(G, 1);
```

```
24     Put_Line("  Edge  Weight");
25     for I in MST_Prim.Vertex loop
26        Put_Line(mst.Edges(I)'Image & " -" & I'image &
27              "   " & mst.Weights(I)'Image );
28     end loop;
29  end Main;
```

Listing B.2: mst_prim.ads

```
 1  package MST_Prim with SPARK_Mode is
 2     Subtype Weight
 3       is Integer range 0 .. Integer'Last;
 4     Subtype Extended_Vertex
 5       is Integer range 0 .. 9;
 6     Subtype Vertex
 7       is Extended_Vertex range 1 .. 9;
 8     type Destinations is array(Vertex) of Extended_Vertex;
 9     type Weights_List is array (Vertex) of Weight;
10     type MST is record
11        --  Weights: Weights_List;
12        --  Edges: Destinations;
13        Weights: Weights_List := (others => Integer'Last);
14        Edges: Destinations := (others => 0);
15     end record;
16     type Visited_Set is array(Vertex) of Boolean;
17     type Adj_List is
18       array (Vertex) of Weight;
```

```
19    type Graph is array (Vertex) of Adj_List;
20    function Extract_Min
21      (Weights: Weights_List; Visited: Visited_Set)
22      return Vertex;
23    function Mst_Prim (G: Graph; r: Vertex) return MST;
24 end MST_Prim;
```

Listing B.3: mst_prim.adb

```
 1 package body MST_Prim with SPARK_Mode is
 2    function Extract_Min(
 3         Weights: Weights_List; Visited: Visited_Set)
 4    return Vertex is
 5       min: Weight := Weight'Last;
 6       min_Index: Integer;
 7    begin
 8       min_Index := 1;
 9       for I in Vertex loop
10          if Weights(I) < min
11            and Visited(I) = False
12          then
13             min := Weights(I);
14             min_Index := I;
15          end if;
16       end loop;
17       return min_Index;
18    end Extract_Min;
```

```
19
20    function Mst_Prim (G: Graph; r: Vertex) return MST
21    is
22       M : MST;
23       Visited: Visited_Set := (others => False);
24       u: Vertex;
25    begin
26       M.Weights(r) := 0;
27       M.Edges(r) := Extended_Vertex'First;
28       for I in Vertex loop
29          u := Extract_Min(M.Weights, Visited);
30          Visited(u) := True;
31          for V in Vertex loop
32             if G(u)(V) > 0 and Visited(V) = False
33               and G(u)(V) < M.Weights(V)
34             then
35                M.Weights(V) := G(u)(V);
36                M.Edges(V) := U;
37             end if;
38          end loop;
39       end loop;
40       return M;
41    end Mst_Prim;
42 end MST_Prim;
```

Appendix C. Textbook Prim's Algorithm Implementation: Full Source Code

Listing C.1: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with MST_Prim; use MST_Prim;
3
4  procedure main is
5     G: MST_Prim.Graph :=
6     -------a---b---c---d---e---f---g---h---i-------+
7           ((0,  4,  0,  0,  0,  0,  0,  8,  0),---- a
8            (4,  0,  8,  0,  0,  0,  0,  11, 0),---- b
9            (0,  8,  0,  7,  0,  4,  0,  0,  2),---- c
10           (0,  0,  7,  0,  9,  14, 0,  0,  0),---- d
11           (0,  0,  0,  9,  0,  10, 0,  0,  0),---- e
12           (0,  0,  4,  14, 10, 0,  2,  0,  0),---- f
13           (0,  0,  0,  0,  0,  2,  0,  1,  6),---- g
14           (8,  11, 0,  0,  0,  0,  1,  0,  7),---- h
15           (0,  0,  2,  0,  0,  0,  6,  7,  0));--- i
16    mst : Graph(9, 8);
17 begin
18    Prim_MST(G_CLR, 1, mst);
19    Print_Graph(mst);
20 end main;
```

Listing C.2: mst_prim.ads

```
1  with Ada.Containers.Formal_Vectors; use Ada.Containers;
```

```
2  with Ada.Numerics.Big_Numbers.Big_Integers;
3  use Ada.Numerics.Big_Numbers.Big_Integers;
4
5  package MST_Prim with SPARK_Mode is
6     Subtype Extended_Vertex is Natural range 0 .. 100;
7     Subtype Vertex is Extended_Vertex range
8          1 .. Extended_Vertex'Last;
9     Subtype Weight is Natural range 0 .. 1000;
10    type Edge is record
11       From: Extended_Vertex;
12       To: Vertex;
13       W: Weight;
14    end record
15      with Dynamic_Predicate =>
16        From /= To;
17    package Vertices is new Formal_Vectors(
18         Element_Type => Vertex, Index_Type => Positive);
19    use Vertices;
20    package Edges is new Formal_Vectors(
21         Element_Type => Edge, Index_Type => Positive);
22    use Edges;
23    package Weights is new Formal_Vectors(
24         Element_Type => Weight, Index_Type => Positive);
25    use Weights;
26    type Adj_Matrix is array(
27         Positive range <>, Positive range <>)
```

```
28    of Weight
29      with Dynamic_Predicate =>
30        Adj_Matrix'First(1) = 1 and
31        Adj_Matrix'Last(1) >= Adj_Matrix'First(1) and
32        Adj_Matrix'First(2) = 1 and
33        Adj_Matrix'Last(2) >= Adj_Matrix'First(2) and
34        Adj_Matrix'Last(1) = Adj_Matrix'Last(2);
35    type Graph (Num_Vertices, Num_Edges: Count_Type)
36    is record --Def 1.1
37       V: Vertices.Vector(Num_Vertices);
38       E: Edges.Vector(Num_Edges);
39    end record
40      with Dynamic_Predicate =>
41        Num_Vertices >= 0 and
42        Num_Edges >= 0;
43
44    --Functions and Procedures-----------------------------
45    procedure Prim_MST(
46         G: Adj_Matrix; r: Integer; MST: in out Graph)
47      with
48        Pre =>
49          G'Length <= 100 and then
50          G'Length = Capacity(MST.V) and then
51          r in G'Length and then
52          Is_Tree(MST) and then
53          MST.Num_Vertices = MST.Num_Edges + 1;
```

```
54    procedure Print_Graph(M : Graph);
55
56    --Min-Priority-Queue-----------------------------------
57    type Queue_Item is record
58       ID: Vertex;
59       Value: Edge;
60       Priority: Weight;
61    end record
62      with Dynamic_Predicate =>
63        Value.To /= Value.From and
64        ID in Vertex and
65        Value.From in Extended_Vertex and
66        Value.To in Vertex and
67        Value.W = Priority and
68        Priority in Weight;
69    Package PQ is new Formal_Vectors(
70       Element_Type => Queue_Item, Index_Type => Positive);
71    use PQ;
72    procedure Update_Item(
73         Q: in out PQ.Vector; new_item: Queue_Item);
74    procedure extract_min(
75         Q: in out PQ.Vector; pair: out Queue_Item)
76      with
77        pre =>
78          not PQ.Is_Empty(Q) and then
79          Integer(Length(Q)) in Vertex,
```

```
80          Post =>
81              Did_Queue_Decrease(Q'Old, Q) and
82              Is_Min(Q, pair) and
83              Is_First_Extracted(Q'Old, Q, pair);
84      function Is_In_Queue(Q: PQ.Vector; V: Vertex)
85           return Boolean is (for some I of Q => I.ID = V);
86      function Get_Item_Priority(Q: PQ.Vector; V: Vertex)
87           return Weight
88        with
89          pre =>
90            Is_In_Queue(Q, V);
91
92      --Ghost Code-------------------------------------------
93      function Is_Tree(G: Graph) return Boolean is
94        (
95            G.Num_Vertices <= 100 and then
96            G.Num_Edges <= 99 and then
97            Length(G.V) = Length(G.E) + 1 and then
98            Capacity(G.V) = Capacity(G.E) + 1 and then
99            (for all I of G.E => I.From /= I.To))
100       with Ghost;
101
102     ---Min-Priority-Queue Ghost Code----------------------
103     function Is_Min(
104          Orig_Queue: PQ.Vector;
105          Min_Priority_Found: Queue_Item) return Boolean
```

```
106       with Ghost,
107       Pre => Min_Priority_Found.ID in Vertex;
108    function Did_Queue_Decrease(
109          Orig_Queue, Result_Queue: PQ.Vector)
110    return Boolean
111       with Ghost;
112    function Is_First_Extracted(
113          Orig_Queue, Result_Queue: PQ.Vector;
114          extracted_pair: Queue_Item)
115    return Boolean
116       with Ghost,
117       Pre => Length(Result_Queue) = Length(Orig_Queue) - 1;
118 end MST_Prim;
```

Listing C.3: mst_prim.adb

```
 1 with Ada.Text_IO; use Ada.Text_IO;
 2
 3 package body MST_Prim with SPARK_Mode is
 4    --Functions and Procedures-----------------------------
 5    procedure Prim_MST(
 6          G: Adj_Matrix; r: Integer; MST: in out Graph)
 7    is
 8       u : Queue_Item;
 9       w : Weight;
10       Q : PQ.Vector(G'Length);
11    begin
```

```
12        for I in 1 .. G'Length loop
13           Q.Append((I, (0, I, Weight'Last), Weight'Last));
14           pragma Loop_Invariant(Integer(Length(Q)) = I and
15                                 I <= Integer(Capacity(Q)));
16           pragma Loop_Invariant(
17              for all J in 1 .. I =>
18               Q.Element(J).ID = J and
19               Q.Element(J).Value.To = J);
20        end loop;
21        Update_Item(Q, (r, (0, r, 0), 0));
22        while not PQ.Is_Empty(Q) loop
23           extract_min(Q, u);
24           pragma Assume(if Length(MST.V) >= 1 then
25              Length(MST.E) = Length(MST.V) - 1);
26           pragma Assume(u.ID = u.Value.To);
27           pragma Assume(u.Value.From /= u.Value.To);
28           pragma Assume(if u.ID = r then
29              u.Value.From = 0 else u.Value.From > 0);
30           pragma Assume(if u.ID /= r then u.Priority > 0);
31           pragma Assume(if u.ID = r then
32              Length(MST.V) = 0 and Length(MST.E) = 0);
33           pragma Assert(if Integer(Length(MST.V)) <= 1 and
34              Length(MST.E) = 0 then Is_Tree(MST));
35           pragma Loop_Invariant(
36              for all I of Q => u.Priority <= I.Priority);
37           if Length(MST.V) < Capacity(MST.V) then
```

```
38             MST.V.Append(u.Value.To);
39          end if;
40          if u.Value.To /= r and
41             Length(MST.E) < Capacity(MST.E) and
42             u.Value.From /= u.Value.To
43          then
44             MST.E.Append(
45                 (u.Value.From, u.Value.To, u.Priority));
46          end if;
47          for I in 1 .. G'Length loop
48             pragma Assume(u.Value.To <= G'Length);
49             w := G(u.Value.To, I);
50             pragma Assume(Is_In_Queue(Q, I));
51             if w > 0 and w < Get_Item_Priority(Q, I) and
52                 I /= u.Value.To then
53                Update_Item(Q, (I, (u.Value.To, I, w), w));
54             end if;
55          end loop;
56       end loop;
57    end Prim_MST;
58
59    procedure Print_Graph (M : Graph) is
60    begin
61       Put ("Vertices: [");
62       for I in 1..Vertices.Length(M.V) loop
63          Put(Vertices.Element(
```

```
64                M.V, Integer(I))'Image & " ");
65       end loop;
66       Put_Line ("]");
67
68       Put ("Edges: [");
69       for I in 1..Edges.Length(M.E) loop
70          Put ("(" & Edges.Element(
71                M.E, Integer(I)).From'Image & "<->" &
72                Edges.Element(
73                    M.E, Integer(I)).To'Image & ")");
74       end loop;
75       Put_Line ("]");
76
77       Put ("Weights: [");
78       for I in 1..Edges.Length(M.E) loop
79          Put(Edges.Element(M.E,Integer(I)).W'Image & " ");
80       end loop;
81       Put_Line ("]");
82
83    end Print_Graph;
84
85    --Min-Priority-Queue-----------------------------------
86    procedure Update_Item(
87         Q: in out PQ.Vector; new_item: Queue_Item) is
88    begin
89       for I of Q loop
```

```
90          if I.ID = new_item.ID then
91             PQ.Replace_Element(
92                 Q, Q.Find_Index(I), new_item);
93          end if;
94       end loop;
95    end Update_Item;
96
97    procedure extract_min(
98         Q: in out PQ.Vector; pair: out Queue_Item)
99    is
100      min_index: Vertex := 1;
101   begin
102      for I in 1 .. Integer(Length(Q)) loop
103         if Q.Element(I).Priority <
104            Q.Element(min_index).Priority
105         then
106            min_index := I;
107         end if;
108         pragma Loop_Invariant(min_index in 1 .. I);
109         pragma Loop_Invariant(
110            for all E in 1 .. I =>
111                Q.Element(E).Priority >=
112                Q.Element(min_index).Priority);
113         pragma Loop_Invariant(
114            for all E in 1 .. min_index-1 =>
115                Q.Element(E).Priority >
```

```
116                 Q.Element(min_index).Priority);
117       end loop;
118       pair := Q.Element(min_index);
119       PQ.Delete(Q, min_index);
120    end extract_min;
121
122    function Get_Item_Priority(Q: PQ.Vector; V: Vertex)
123     return Weight
124    is
125       Res: Weight := Weight'Last;
126    begin
127       for I of Q loop
128          if I.ID = V then
129             Res := I.Priority;
130          end if;
131       end loop;
132       return Res;
133    end Get_Item_Priority;
134
135    ---Min-Priority-Queue Ghost Code-----------------------
136    function Is_Min(
137         Orig_Queue: PQ.Vector;
138         Min_Priority_Found: Queue_Item) return Boolean
139    is
140      (for all I of Orig_Queue =>
141         I.Priority >= Min_priority_Found.Priority);
```

```
142
143     function Did_Queue_Decrease(
144          Orig_Queue, Result_Queue: PQ.Vector)
145     return Boolean is
146          (Length(Result_Queue) = Length(Orig_Queue) - 1);
147
148     function Is_First_Extracted(
149          Orig_Queue, Result_Queue: PQ.Vector;
150          extracted_pair: Queue_Item) return Boolean
151     is
152       (for some I in 1 .. Integer(Length(Orig_Queue)) =>
153           extracted_pair = Orig_Queue.Element(I)
154        and then (for all X in 1 .. I - 1 =>
155              (Result_Queue.Element(X).Priority >
156                  extracted_pair.Priority))
157        and then (for all J in 1 .. I - 1 =>
158                       Orig_Queue.Element(J) =
159                          Result_Queue.Element(J))
160        and then (for all J in I .. Integer(
161                       Length(Result_Queue)) =>
162                  Orig_Queue.Element(J + 1) =
163                       Result_Queue.Element(J)));
164 end MST_Prim;
```

Bibliography

1. Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In *International Symposium on Leveraging Applications of Formal Methods*, pages 461–478. Springer, 2016.

2. AdaCore. Introduction to SPARK. `https://learn.adacore.com/courses/intro-to-spark/index.html`, 2021. (accessed: 05.19.2022).

3. AdaCore. SPARK user's guide: Applying SPARK in practice. `https://docs.adacore.com/spark2014-docs/html/ug/en/usage_scenarios.html`, 2022.

4. Matt Bishop, Brian Hay, and Kara Nance. Applying formal methods informally. In *2011 44th Hawaii International Conference on System Sciences*, pages 1–8. IEEE, 2011.

5. Laura Humphrey, Ryan Baity, and Kenneth Hopkinson. *Formal verification of safety-critical software using SPARK*. IET, 2021.

6. Aaron Dutle, Mariano Moscato, Laura Titolo, César Muñoz, Gregory Anderson, and François Bobot. Formal analysis of the compact position reporting algorithm. *Formal Aspects of Computing*, 33:65–86, 2021.

7. Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

8. John A Van der Poll. Formal methods in software development: A road less travelled. *South African Computer Journal*, 2010(45):40–52, 2010.

9. Quentin Ochem. Rust and SPARK: Software reliability for everyone. *Electronic Design - Industrial Automation*, 4 2017.

10. Steve Klabnik and Carol Nichols. *The Rust Programming Langauge*. 1 edition.

11. Department of Defense requirements for high order computer programming languages. 'Ironman'. Technical Report ADA100403, OUSE R&E, Washington, DC, January 1977.

12. Claire Dross and Johannes Kanig. Recursive data structures in SPARK. In *International Conference on Computer Aided Verification*, pages 178–189. Springer, 2020.

13. Maroua Maalej, Tucker Taft, and Yannick Moy. Safe dynamic memory management in Ada and SPARK. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 37–52. Springer, 2018.

14. Docs.rs. Crate contracts. Accessed Jun. 10, 2022 [Online].

15. Vytautas Astrauskas, Aurel Bílỳ, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. The Prusti project: Formal verification for Rust. In *NASA Formal Methods Symposium*, pages 88–108. Springer, 2022.

16. John W McCormick and Peter C Chapin. *Building high integrity applications with SPARK*. Cambridge University Press, 2015.

17. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *European Symposium on Programming (ESOP)*, pages 125–128. Springer, 2013.

18. Prusti. Prusti user guide. Accessed Jun. 11, 2022 [Online].

19. Prusti. Prusti. Accessed Jun. 11, 2022 [Online].

20. Ryan Baity, Laura R Humphrey, and Kenneth Hopkinson. Formal verification of a merge sort algorithm in SPARK. In *AIAA Scitech 2021 Forum*, 2021.

21. Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In *International Conference on Interactive Theorem Proving*, pages 17–26. Springer, 2014.

22. MITRE. Common Weakness Enumeration (CWE). `https://cwe.mitre.org`, 2021. (accessed: 05.19.2022).

23. Roderick Chapman and Yannick Moy. Cyber security. `https://www.adacore.com/uploads/books/pdf/AdaCore-Tech-Cyber-Security-web.pdf`, 2018.

24. David M Bourg and Bryan Bywalec. *Physics for Game Developers: Science, math, and code for realistic effects*. " O'Reilly Media, Inc.", 2013.

25. AdaCore. SPARK user's guide: How to view GNATprove output. `https://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_view_gnatprove_output.html#the-analysis-results-summary-file`, 2022.

26. Claire Dross and Johannes Kanig. Making proofs of floating-point programs accessible to regular developers. In *Software Verification*, pages 7–24. Springer, 2021.

27. John Rushby. Formal methods and their role in the certification of critical systems. In *Safety and reliability of software based systems*, pages 1–42. Springer, 1997.

28. Gerard O'Regan. *Concise guide to formal methods*. Springer, 2017.

29. Ghada Bahig and Amr El-Kadi. Formal verification of automotive design in compliance with ISO 26262 design verification guidelines. *IEEE Access*, 5:4505–4516, 2017.

30. Davide Basile, Maurice H ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari. On the industrial uptake of formal methods in the railway domain. In *International Conference on Integrated Formal Methods (iFM)*, pages 20–29. Springer, 2018.

31. RTCA. Formal methods supplement to DO-178C and DO-278A. Technical Report DO-333, RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71), 2011.

32. Jennifer A Davis, Matthew Clark, Darren Cofer, Aaron Fifarek, Jacob Hinchman, Jonathan Hoffman, Brian Hulbert, Steven P Miller, and Lucas Wagner. Study on the barriers to the industrial adoption of formal methods. In *International Workshop Formal Methods for Industrial Critical Systems*, pages 63–77. Springer, 2013.

33. Aifheli Nemathaga and John Andrew van der Poll. *Formal Methods Adoption in the Commercial World*. PhD thesis, University of South Africa, 2020.

34. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

35. Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. Continuous formal verification of Amazon s2n. In *International Conference on Computer Aided Verification (CAV)*, pages 430–446. Springer, 2018.

36. John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018.

37. John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification (CAV)*, pages 231–241. Springer, 2019.

38. AdaCore. Download GNAT Community Edition. `https://www.adacore.com/download`, 2022. (accessed: 05.19.2022).

39. Christophe Garion. SPARK by example. `https://github.com/tofgarion/spark-by-example`, 2019. (accessed: 05.19.2022).

40. Artur Mariano, Dongwook Lee, Andreas Gerstlauer, and Derek Chiou. Hardware and software implementations of Prim's algorithm for efficient minimum spanning tree computation. In *International Embedded Systems Symposium*, pages 151–158, 2013.

41. Maarten van Steen and Andrew S. Tanenbaum. Distributed systems. `https://www.distributed-systems.net/index.php/books/ds3/`, 2017.

42. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 233–247. Springer, 2012.

43. Eduardo Brito and Jorge Sousa Pinto. Program verification in SPARK and ACSL: A comparative case study. In *International Conference on Reliable Software Technologies*, pages 97–110, 2010.

44. Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. Verified Rust monitors for Lola specifications. In *International Conference on Runtime Verification*, pages 431–450. Springer, 2020.

45. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

46. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In *International Conference of B and Z Users*, pages 457–476, 2003.

47. ClearSy. Atelier B user manual version 4.0. `https://www.it.uu.se/edu/course/homepage/bkp/ht13/AB/documentation/manual/ManuelUtilisateurAtb4/uk/user_uk.pdf`, 2022.

48. Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. Functional correctness of c implementations of dijkstra's, kruskal's, and prim's algorithms. In *International Conference on Computer Aided Verification*, pages 801–826. Springer, 2021.

49. Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

50. Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–11, 2018.

51. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

52. AdaCore. SPARK user's guide: Overview of SPARK language. `https://docs.adacore.com/spark2014-docs/html/ug/en/source/language_restrictions.html`, 2022.

53. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

54. R. M. Baity. Formal verification for high assurance software: A case study using the SPARK auto-active verification toolset. Master's thesis, Air Force Institute of Technology, AFIT Scholar, 2021.

55. Peter Höfner and Bernhard Möller. A new correctness proof for Prim's algorithm. `http://www.Informatik.Uni-Augsburg.de`, 2019.

56. Claire Dross and Johannes Kanig. Recursive data structures in SPARK. In *International Conference on Computer Aided Verification (CAV)*, pages 178–189. Springer, 2020.

57. AdaCore. SPARK 2014. `https://github.com/AdaCore/spark2014`, 2023. (accessed: 01.17.2023).

58. Brian Wheelhouse, Laura Humphrey, and Kenneth Hopkinson. Formal verification of prim's algorithm in spark. In *56th Hawaii International Conference on System Sciences (HICSS)*, pages 6695–6703, 2023.

59. AdaCore. SPARK user's guide: Formal containers library. `https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/spark_libraries.html#formal-containers-library`, 2022.

Acronyms

ACSL ANSI/ISO C Specification Language. 20, 38

ADS-B Automatic Dependent Surveillance-Broadcast. v, 5

AFRL Air Force Research Laboratory. iv, 1

AoRTE absence of runtime errors. 16

AWS Amazon Web Services. 19

CPR Compact Position Reporting algorithm. v, 5

CWE Common Weakness Enumeration. 12, 20, 38

MC/DC modified condition/decision coverage. 11, 20, 38

MSF minimum spanning forest. 21, 39

MST minimum spanning tree. 1, 2, 19, 21, 24, 25, 27, 28, 29, 30, 37, 39, 42, 43, 44, 48, 52

NaN not a number. 16

SHOLIS Ship Helicopter Operating Limits Information System. 11, 20, 38

SMT satisfiability modulo theories. 5, 9, 21, 41

VC verification condition. 9, 21, 41

VST Verified Software Toolchain. 21, 34, 39

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2023	Master's Thesis	Sept 2021 — Mar 2023

4. TITLE AND SUBTITLE

SAFE AND RELIABLE SOFTWARE AND THE FORMAL VERIFICATION OF PRIM'S ALGORITHM IN SPARK

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Brian S. Wheelhouse, 2d Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-23-M-070

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL, Aerospace Systems Directorate, Autonomous Controls Branch
Building 146, Rm 300
WPAFB OH 45433-7765
DSN 713-7032, COMM 937-713-7032
Email: laura.humphrey@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RQQA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Despite evidence that formal verification helps produce highly reliable and secure code, formal methods, i.e., mathematically based tools and approaches for software and hardware verification, are not commonly used in software and hardware development. The limited emphasis on formal verification in software education and training suggests that many developers have never considered the benefits of formal verification. Despite the challenging nature of their mathematical roots, software verification tools have improved; making it easier than ever to verify software. SPARK, a programming language and a formal verification toolset, is of particular interest for the AFRL, and will be a primary focus of this thesis. This thesis provides an overview of two safe and reliable languages with verification tools, namely SPARK and Rust. Then, to demonstrate the benefits of modern software verification tools, two examples of software verification in SPARK are presented. These examples include a verified implementation of the quaternion data structure and two implementations of Prim's algorithm, to further demonstrate the usability and methodology of the SPARK verification toolset.

15. SUBJECT TERMS

formal methods, software verification, spanning trees, minimum spanning trees, highly reliable software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Kenneth M. Hopkinson, AFIT/ENG
U	U	U	UU	112	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, ext 4579; kenneth.hopkinson@afit.edu