Formal Verification for High Assurance Software: A Case Study Using the SPARK Auto-Active Verification Toolset

THESIS

Ryan M Baity, Second Lieutenant, USAF

AFIT-ENG-MS-21-M-009

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-21-M-009

FORMAL VERIFICATION FOR HIGH ASSURANCE SOFTWARE: A CASE STUDY USING THE SPARK AUTO-ACTIVE VERIFICATION TOOLSET

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 Cyber Operations

Ryan M Baity, B.S.C.S.

Second Lieutenant, USAF

March 25, 2021

AFIT-ENG-MS-21-M-009

FORMAL VERIFICATION FOR HIGH ASSURANCE SOFTWARE: A CASE STUDY USING THE SPARK AUTO-ACTIVE VERIFICATION TOOLSET

THESIS

Ryan M Baity, B.S.C.S.
Second Lieutenant, USAF

Committee Membership:

Kenneth M. Hopkinson, Ph.D
Chair

Timothy H. Lacey, Ph.D
Member

Maj Richard Dill, Ph.D
Member

Laura R. Humphrey, Ph.D
Member

AFIT-ENG-MS-21-M-009

Abstract

Software is an increasingly integral and sophisticated part of safety- and mission-critical systems. Software lies at the backbone of medical devices, automobiles, mobile devices, Internet of Thing (IOT) devices, military weapons systems, and much more. Poorly written software can lead to information leakage, undetected cyber breaches, and even human injury in cases where the software directly interfaces with components of a physical system. These systems may range from power facilities to remotely piloted aircraft. Software bugs and vulnerabilities can lead to severe economic hardships and loss of life in these domains. As fast as software spreads to automate many facets of our lives, it also grows in complexity. The complexity of software systems combined with the nature of the critical domains dependent on those systems results in a need to verify and validate the security and functional correctness of such software to a high level of assurance. The current generation of formal verification tools make it possible to write code with formal, machine-checked proofs of correctness. This thesis demonstrates the process of proving the correctness of code via a formal methods toolchain. It serves as a proof of concept for this powerful method of safety- and mission-critical software development.

Table of Contents

Page

List of Figures

List of Tables

FORMAL VERIFICATION FOR HIGH ASSURANCE SOFTWARE: A CASE STUDY USING THE SPARK AUTO-ACTIVE VERIFICATION TOOLSET

I. Introduction

1.1 Problem & Objective

The world of formal verification is new to many software developers. Large public and private organizations such as The National Aeronautics and Space Administration (NASA), Microsoft, Airbus, and Amazon have openly utilized formal methods in one way or another. Many other organizations use formal methods but do not openly discuss them. Even for those that do, they are not always able to disclose their precise implementations to the public. For private companies, this ensures a competitive advantage over one another. Because the particulars of formal verification as used in many private projects are not publicly available, this thesis attempts to bring this formal verification process to the forefront by applying formal methods to different algorithms. The goal is to demonstrate that modern verification tools, such as SPARK, provide practical formal methods frameworks for accomplishing the verification of code within Air Force research and development circles. This Thesis makes use of AdaCore's SPARK[1] due to its maturity and use within the Air Force Research Laboratory, but other formal methods tools and frameworks do exist. The SPARK framework takes a software implementation and checks whether it is free from run-time errors and whether it satisfies user-specified contracts; if not, it provides the user with feedback on potential issues that must be corrected to achieve various levels of proof. Built upon several publications that demonstrate the benefits

[1] `https://www.adacore.com/about-spark`

of formal methods, this research enumerates the SPARK Verification and Validation (V&V) process and demonstrates its viability to be used by the U.S. Air Force.

1.2 Overview

This section is based on a portion of a ready to publish Institution of Engineering and Technology (IET) chapter contribution and a portion of an already published paper.[2] [3] The increasing size and complexity of software running on safety-critical systems such as those developed in the aerospace industry has raised concerns about how to continue to ensure software quality and reliability, especially in a cost effective way [1, 2]. At the same time, more and more devices are becoming interconnected, opening new avenues for cyber security attacks. While any type of software fault is of concern in a safety-critical system, cyber security-related faults are particularly pernicious; whereas a general fault might never encounter the circumstances necessary to trigger it, a cyber security-related fault is actively sought out and exploited by attackers.

Even the smallest software faults can have severe safety and financial repercussions. For example, in 1990 AT&T lost more than $60 million in unconnected calls due to a misplaced break statement in a network switch software patch that remained undetected even after rigorous testing [3]. The 2014 Heartbleed OpenSSL vulnerability, a simple defect caused by a single unchecked memory copy, allowed attackers to remotely read protected memory from an estimated 24–55% of HTTPS sites [4]. In 2017, at least 465,000 St. Jude's Medical RF-enabled cardiac devices were recalled due to possible cyber security vulnerabilities [5]. In the same year, the WannaCry

[2]Laura R Humphrey, Ryan Baity, Kenneth Hopkinson chapter 5 contribution to upcoming IET Textbook - Section: Introduction.

[3]Ryan Baity, Laura R Humphrey, Kenneth Hopkinson. Formal verification of a merge sort algorithm in spark. In *AIAA Scitech 2021 Forum*, page 0039, 2021. DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. Case #88ABW-2020-3580.

ransomware and NotPetya malware attacks, which made use of the EternalBlue exploit to target a vulnerability in Microsoft's implementation of the Server Message Block (SMB) protocol, caused an estimated $4 billion and $10 billion in damages, respectively [6, 7]. Recently, concerns have been raised about cyber security vulnerabilities in automobiles and aircraft, especially as driverless cars and unmanned air vehicles begin to see use [8].

Unfortunately, it is extremely difficult to ensure that software is free of faults. One of the primary mechanisms for finding software faults is testing, but testing can only achieve partial coverage of all possible software behaviors, a problem that is exacerbated as the size and complexity of software increases [9]. Because testing only achieves partial coverage, it can only reveal the presence of faults; it cannot prove their absence. But given that even a single cyber security-related fault will be sought out and exploited if found, it would be very much beneficial to do all that can be done to prove their absence!

A possible solution is the use of *formal methods* [10, 11], i.e. mathematically-based tools and techniques for design and verification of software and hardware, to provide evidence of software quality and reliability. Formal methods have their roots in formal logic, discrete mathematics, and computer readable languages. Using these as a foundation, they aim to provide mathematical guarantees about system behavior through proof and analysis rather than testing, similar to the way in which one can prove the Pythagorean Theorem is correct using geometric axioms rather than testing it against the set of all possible right triangles. Formal methods consist of two major activities, modeling and analysis, often done with the assistance of automated or semi-automated tools. They are analogous to mathematically-based approaches used in traditional engineering disciplines (e.g. statics for civil engineering, dynamics for mechanical engineering, and stoichiometry for chemical engineering). Formal methods

can be used to analyze a range of system artifacts, including high- and low-level requirements, architectures, source code, object code, and discrete logic hardware, for a variety of properties such as traceability, completeness, consistency, compliance, and robustness. Formal methods are an accepted means to satisfy verification objectives in certification standards across a variety of domains, for instance, in railway (EN 50128) [12] and industrial (IEC 61508) [13] processes and for avionics (DO-333 supplement to DO-178C) [14, 11, 15]. Certification standards for other domains such as the automotive (ISO 26262) [16] and nuclear (IEC 60880) and space (ECSS-QST-80C) [17] domains also recognize some uses of formal methods as a verification technique. Formal methods have been used by Airbus to determine worst-case execution time and max stack usage [18], by Dassault-Aviation to replace software robustness testing [19], by Microsoft to verify third-party drivers [20], by Amazon Web Services to verify whether user access control policies meet user security requirements [21] and whether specified computers are reachable from the outside world [22], and to prove security properties and functional correctness of capabilities of the seL4 microkernel [23], to name a few.

Given the variety of artifacts, properties, and underlying mathematical frameworks involved, formal methods are somewhat broad. [24] gives an extensive overview of formal methods that discusses different categories of approaches according to level of rigor; variations in the extent to which formal methods can be applied in terms of lifecycle stage, proportion of the system covered, and types of properties analyzed; the value of formalizing specifications and the importance of validating them; the benefits and fallibilities of formal methods; automated and semi-automated systems and tools; and a review of industrial applications. It also includes a quick introduction to mathematical logic, including propositional calculus, predicate calculus, first-order theories, modal logics, etc. The DO-333 [14] supplement to DO-178C [25]

gives a more recent overview of formal methods, with [11] and [15] each providing an overview of DO-333 and illustrating the application of various formal methods tools and approaches on concrete examples relevant to the aerospace domain.

Though formal methods have many benefits, adoption has been slow for a variety of reasons, including acceptance by certifiers, scalability, usability, and education [26]. However, given the recent inclusion of formal methods in certification standards, successful uses of formal methods in industrial applications, and continual improvements in tool scalability and usability, it is expected to see an increased use of formal methods.

Among previously stated research goals, this research aims to hit upon the last barrier: education. Specifically, it aims to give the reader an impression of what is required to formally prove functional correctness of source code (i.e. to verify compliance of source code with low-level requirements specifying its desired behavior). This research utilizes SPARK [27, 28, 29], which has been used to develop highly assured software in a variety of domains, including the aerospace domain [30, 31].

1.3 Roadmap

Here is a roadmap of this document's organization. Chapter II gives a brief background of the history of formal methods, a background of SPARK basics, and some previous SPARK projects. Chapter III uses the SPARK framework to formally verify three algorithms; interpolation, merge sort, and priority queue. Chapter IV analyzes the three algorithms implemented and formally proven. Chapter V concludes this thesis.

II. Background and Literature Review

2.1 Preamble

This chapter details information and context needed for understanding Chapter III. It starts with a timeline of papers with relation to formal methods and their applications in Section 2.2. Then in Section 2.3, this chapter gives a brief overview of the formal methods framework used in this thesis (SPARK). In Section 2.4, this chapter discusses a few projects that utilized and benefited from the SPARK framework. In Section 2.5, this chapter describes SPARK's relationship with the Common Weakness Enumeration (CWE). Section 2.6 presents a few formal methods frameworks similar to SPARK. Section 2.7 provides a deeper view of the SPARK framework. Section 2.8 touches on SPARK's role within the Software Development Lifecycle. With the context and information provided in this chapter, Chapter III can be much more easily understood and comprehended.

2.2 General Timeline

The idea of verifying code for correctness has been around for decades. Hoare brought up program correctness via formal methods in his 1969 paper "An Axiomatic Basis for Computer Programming" [32]. He writes about "Proofs of Program Correctness" in Section 5 of his paper. He mentions that programming is an exact science and if one creates a proof whose axioms align with the implementations of the programming language, then that proof "may be used to prove the correctness of the program." Interestingly, Hoare predicts that formal methods will not become widely used for nontrivial programs until better proof techniques become available. Also acknowledged by Hoare is that these proofs will be difficult to create. Although he mentions some drawbacks, Hoare stated that the benefits indeed outweigh the

drawbacks.

Hoare provided that proving programs would solve a few issues in the programming world. It can be difficult when a large project is created and someone is tasked to come in at a later date and make a change or add a subroutine. One of the hardest parts of creating this subroutine is defining its purpose or creating its specification. That is, what are the pre- and postconditions, etc. Hoare stipulates that once a subroutine is formally defined and proven, it is easy to plug into a larger whole when it satisfies the same "criterion of correctness." Another point he makes is that every application created in any programming language, especially those specific to a machine, is most likely taking advantage of a quirk or unique feature of the language and/or machine. When transferring that application to another machine, it may fail. That machine-dependent feature will always be revealed when attempting to "prove the program for machine-independent axioms."

Overall, Hoare pushed the idea of proving the correctness of code as far back as 1969. When Hoare spoke of programming languages he mentioned Algol, Fortran and Cobol. Today, familiarity with these legacy languages exists mainly with those that have been in the industry for decades, yet even today his message resonates. Today it would be said that he wanted to verify code via formal methods. He finishes off his paper with an open invitation: "it is hoped that many of the fascinating problems involved will be taken up by others." According to the Google Scholar search engine,[1] this paper has been cited over 7,748 times. This paper was truly a prominent one describing the possibilities and benefits of formal verification of code.

Twenty one years later, Kemmerer published his paper and found that integrating formal methods during development is "faster and more cost-effective" than trying to verify the code later [33]. An interesting point about the reasoning for utilizing

[1] `https://scholar.google.com/`

formal methods in the development of his project was made by Kemmerer. He talked about his motivations for said utilization of which there were two. First, to be able to receive the needed certification from the Department of Defense (DoD) for his project. Second, he wanted to be sure that his team could deliver on the security requirements set for in the project specifications. This desire is interesting because it shows that in 1990 the DoD already required some form of formal evidence of software correctness and it shows that at least some developers were utilizing the ideas that Hoare alluded to in his seminal 1969 paper [32].

In 1996, the NASA/WVU Software IV & V Facility, Software Research Laboratory, released a technical report named "Experiences Using Formal Methods for Requirements Modeling" [34]. This report looks into three case studies in which the team applied formal methods to requirements modeling for their spacecraft's fault protection systems. The team mentioned that at the time, studies have shown that formal methods may be very beneficial for "improving the safety and reliability of large software systems." Their technique was simple, take three problems the team was already facing, work with the requirements analysts, model the code, and apply formal methods. In all three cases the team found that the benefits observed due to the early modeling of requirements "more than outweigh the effort needed to maintain multiple representations." The takeaway is that modeling is an effective way to look at a program and apply formal methods that can bring issues to light that were not even thought of. It was also interesting that the team concluded it is easier to have a small team of formal methods experts, rather than to train developers.

A little over a decade later, in 2008, Heitmeyer, et al. published "Applying Formal Methods to a Certifiably Secure Software System" [35]. They addressed the cost of verifying code. They argued that validating large software projects is extraordinarily costly when done in its entirety. In their approach, the authors separate code into

three categories and only actually formally verify the first category. The first category is what the team called the Event Code, which represents the code that implements a "Top Level Specification" (TLS) and touches one or more of what the authors call "Memory Areas of Interest" (MAIs). The other two categories are Trusted Code and Other Code. Trusted Code is self-evident and Other Code is any code that is not trusted, but does not interact with a TLS or MAI. This strategy is much more effective because the first category is only 10% of the total code, according to the team. The team applied this to an "evaluation of the separation kernel of an embedded software system." The authors also go into to some open problems in Section 7 of their paper. The takeaway from this paper: in many cases it is not feasible to formally specify an entire project. Instead, there are key parts of the project that should get more formal methods attention. In the authors' eyes, as mentioned above, these key parts are what the team defined as Event Code, which can be interpreted as critical code.

In the same year, "A Survey of Automated Techniques for Formal Software Verification" was written [36]. The reader is presented with three techniques for verifying software via static analysis. The first being abstract domains, the second being model checking, and the third being bounded model checking. The paper describes the three techniques and how they are different. The authors conclude that Model Checking tools with abstraction can track invariants, such as loop invariants, and Bounded Model Checkers are "unable to prove even simple properties if the program contains deep loops." These results were interesting in the fact that many papers seem to apply formal methods to models of the code it represents. It would be interesting to see formal methods applied to the actual program code via a formal methods framework. An example of this form of formal methods will be introduced in two paragraphs.

Woodcock, et al. discussed general experiences with Formal Methods in 2009 [37]. They detailed how utilizing formal methods has helped software development

across the board. Additionally, the team advocated for developing a global Verified Software Repository. This is part of the Verified Software Initiative. The technique of their paper is similar to a survey paper. The team investigates numerous projects that have utilized formal methods and analyze the effects on the project itself. The effort collected data from 62 industrial projects. The paper finds that three times as many projects "reported a reduction in time, rather than and increase" when applying formal methods. More impressively, 92% found that the quality of their project was improved and the other 8% was no change in quality. These results mean that applying formal methods did not worsen the quality of the code in any of the 62 sampled projects. In fact the team found that with regard to time spent, it only worsened 12% of the time and with regard to cost, it only worsened 7% of the time. In summary, formal methods were found to have improved the overall quality of product, with minimal effect on time and cost.

Moving much closer to modern day, "Development and Verification of a Flight Stack for a High-Altitude Glider in Ada/SPARK 2014" was written [30]. This paper introduces a weather balloon that is slightly different than the typical weather balloon used today. A traditional weather balloon gains many kilometers of altitude, logs data via its sensors, pops and the data come back to Earth's surface via a parachute. The researchers' balloon was designed with a glider that actually transports the data back to the take off point. Their goal was to test out the SPARK 2014 framework and use it in their balloon software verification process. This means that they are not specifically checking models of the code, instead they check the executable code itself. The paper introduces the GNATprove feature of SPARK, which is the static analyzer for SPARK. The paper also looks into problems they had with SPARK and some workarounds. It discusses some design limitations, such as limits with access types, limited polymorphism, etc. The authors concluded in their results that SPARK 2014

"raises the bar for formal verification and its tools, but developers still have to be aware of limitations."

The one point that all of the publications have in common is the use of some form of formal methods as a means to creating high quality software and verifying program correctness, but they do not disclose their actual formal methods implementations. This need for the use of formal methods applies to the U.S. Air Force. It should always utilize high quality and functionally correct software; SPARK can potentially help with that.

2.3 A Brief Overview of SPARK

This section is based on a portion of a ready to publish IET chapter contribution.[2] SPARK is both a programming language and associated toolset for formal verification [28]. The SPARK language is based on the Ada programming language [38, 39]. Ada is a statically-typed, imperative, object-oriented language with a strong type system, and the principles underlying its design are intended to encourage reliability, maintainability, and efficiency. These characteristics are inherited by SPARK, which both removes a few features from Ada that make verification difficult and adds a small number of features that facilitate verification. Several good resources are available for learning SPARK and Ada [28, 39, 29, 38]. In what follows, it is assumed that the reader has some familiarity with SPARK/Ada or is otherwise able to deduce the meaning of basic SPARK/Ada language features.

In terms of formal verification, SPARK performs two forms of sound static analysis on source code. The first is flow analysis, which checks initialization of variables and compliance with user-specified data flow contracts. Flow analysis can also identify unused assignments and unmodified variables, which often indicate extraneous/un-

[2]Laura R Humphrey, Ryan Baity, Kenneth Hopkinson chapter 5 contribution to upcoming IET Textbook - Section: Background on SPARK.

necessary code. The second is proof, which checks for both functional correctness, i.e. compliance with user-specified behavioral contracts, and the absence of run-time errors. It should be noted that these checks and contracts can be optionally compiled as executable assertions that are evaluated at run-time, (e.g. to debug code during development or to verify code through testing if static analysis cannot be fully completed).

There are different levels to which one can use SPARK to verify a program [40]. Generally, the level of verification performed during development is incremental, going from lowest to highest. Colloquially, the levels are referred to in order as "stone," "bronze," "silver," "gold," and "platinum." Verification at the stone level is achieved when code is accepted by SPARK, since the SPARK language has stricter legality rules than Ada. Bronze is achieved when flow analysis returns with no error. Silver ensures that no run-time errors, (e.g. division by zero or numeric overflow/underflow), will be encountered when executing the program. Gold consists of verifying compliance with key user contracts/specifications; however, at this level, the specifications only partially describe the desired behavior of the code. Platinum consists of verifying compliance of the code against a complete set of specifications. Briefly noted, these levels have a rough correspondence with levels of assurance in certification standards, (e.g. "software level" in DO-178C and "software integrity level" in IEC 61508, EN 50128, et al.). As discussed in [40], silver is the baseline level for all software levels except Level E, i.e. software whose anomalous behavior would have no effect on aircraft operational capability or pilot workload. Software at Level A or Level B may aim for gold or platinum depending on whether key properties or full functional correctness can be more easily verified to a sufficient level by other means. Targeting platinum becomes less likely at Level C or Level D, since verification by testing can be more easily argued to be sufficient. For software at Level E, silver

might still be considered but could be weakened to bronze if other means were used to build sufficient confidence that the software is free of run-time errors or such errors were sufficiently mitigated.

Though SPARK aims to perform fully automated verification at all of these levels, manual adaptation of the source code is in general necessary. For example, if one were to translate source code originally written in Ada to SPARK, one would have to remove features unsupported by SPARK discussed in Section 2.7.2. Verifying functional correctness also requires that the user formalize and manually write the requirements at the level of the source code in the form of contracts: type invariants, data dependency and flow contracts, assertions, loop invariants, loop variants, pre- and postconditions on subprograms, etc. In order to guide the underlying provers toward a proof of certain contracts, (e.g. pre- and postconditions on a subprogram), it is often necessary to write additional contracts (e.g. loop invariants on loops inside the subprogram).

2.4 Previous SPARK Projects

This section is based on a portion of a ready to publish IET chapter contribution.[3] Many projects have or are currently using SPARK to help provide evidence of software security and correctness, often in conjunction with other formal methods and more traditional review-based and test-based approaches. "Are we there yet? 20 years of industrial theorem proving with SPARK" gives a historical overview of several such projects [41]. Here, a more detailed description of some of those projects along with other SPARK-related projects that have an aerospace or cyber security focus is given.

One of the first major applications of SPARK was to the Ship Helicopter Operating Limits Information System (SHOLIS), a safety-critical system that aids the safe

[3]Laura R Humphrey, Ryan Baity, Kenneth Hopkinson chapter 5 contribution to upcoming IET Textbook - Section: SPARK aerospace & cyber security projects.

operation of helicopters on Naval vessels [42]. The work was carried out under the 1991 version of UK Interim Defence Standards 00-55 and 00-56, which required the use of formal methods for safety-critical applications. In addition to SPARK, the effort used the Z notation or Z [43] to write parts of the specification. Z is based on set theory and a first-order predicate calculus and is therefore amenable to proof, either manually or with tool assistance. In this effort, the software requirement specification was written in a combination of English and Z. The software design specification, which refines the software requirement specification by adding implementation details, was written in a combination of English, Z, and SPARK. The code was written in SPARK. The effort, therefore, consisted of two categories of proof activity: Z proof and SPARK proof. The Z proof activities were mainly manual and included checking the consistency of global variables and constants, the existence of valid initial states, and that operator preconditions calculated from the Z specifications [44] matched the specifier's expectations. Based on a software safety analysis, some software components were classified as safety-critical. For every safety-critical subprogram, SPARK pre- and postconditions were written based on the Z specifications, and data flow and information flow contracts were written to check for separation between safety-critical and non-safety-critical code. SPARK tools were then used to statically prove these contracts and the absence of run-time errors. Informal feedback found that the process of proving Z specifications was more efficient at finding faults than testing, and the use of formal specification led to simpler code that was easier to understand and maintain. The effort also found that SPARK proof was favorable in comparison with unit testing, particularly for proving the absence of run-time errors.

SPARK was also used in the development of the Lockheed C130J "Hercules" [45, 46], in particular to implement the core of the mission computer, which performs the majority of the aircraft's mission critical functions. Formal modeling methods

were used to develop the requirements, including the use of Parnas Tables [47] to write specifications relating inputs to required outputs. The requirements were then used to write data and information flow contracts and postconditions for SPARK subprograms. Prior to modified condition/decision coverage (MC/DC) testing, only basic flow analysis with SPARK was performed. Lockheed found that coding proceeded at near normal Ada rates, yet there was an 80% savings in the expected budget allocated to MC/DC testing, in part because the code had an unusually low fault density of less than one tenth the expected industry norm for safety critical software. SPARK was later used to perform a more thorough analysis [41].

In terms of security, SPARK was used in the development of the Certification Authority for the Multos smart card [48]. The system's security-enforcing kernel was written in SPARK, the infrastructure in Ada, and the graphical user interface in C++. Other parts of the system were built using commercial off-the-shelf (COTS) components to save time and money. The effort used a rigorous process to develop the requirements, including a Formal Security Policy Model (FSPM), which was formalized in Z. The FSPM was manually reviewed for correctness, and a typechecker was used to check its consistency. A formal top-level specification (FTLS) was created to describe the functionality of the system. The FTLS was also formalized in Z, a typechecker was used to check that it was well formed, and some special features allowed it to be checked against the FSPM. A high-level design was developed to describe the system's internal structure and explain how the components work together. This was particularly important for establishing confidentiality and integrity of the system through separation between COTS components and the security-enforcing kernel. A detailed design refined the requirements, assigning functionality to specific software modules. The process structure was modeled by mapping sets of Z operations in the FTLS to actions in the communicating sequential processes (CSP) language [49].

The CSP model was used to check that the overall system was deadlock-free and that there was no concurrent processing of security-critical functions. Rigorous coding standards were used for all code. Manual reviews were carried out to check for compliance with requirements, conformance to standards, and internal consistency. SPARK was used to carry out some proofs of absence of run-time errors and data flow errors, though it was not used to prove functional correctness. Instead, manual reviews and tests were used to check correctness. In the first year of use, reported faults were far better than the industry average for new systems.

SPARK was also used in Tokeneer [50, 51], an NSA-funded demonstrator of high-security software engineering approaches that was later made open-source [52]. The Tokeneer system consists of a secure enclave containing workstations that should only be accessible to users who can be biometrically authenticated. Furthermore, the level of user access allowed depends on factors such as the user's allowed roles, security clearance, time of day, etc. The Tokeneer ID Station (TIS) project re-developed a core component of the Tokeneer system. Similar to the efforts on SHOLIS and the Multos smart card CA, a formal specification describing system states and operations was written in Z. To validate this specification, existence of a valid initial state was proven and operator preconditions were calculated and checked against the specifier's expectations. A typechecker was also used to check for consistency of types in all expressions. Security properties were also expressed in Z, and the formal specification was manually proven to exhibit the security properties. A formal design describing the system in terms of concrete states and operations in Z was developed to refine the formal specification in areas where more details were needed to bring the specifications closer to an implementation. For operations where the refinement was non-trivial, proofs that the formal design correctly refined the formal specification were carried out. A rigorous design process was then used to develop a software

architecture, map Z specifications to software packages, and put bounds on values in the software implementation, (e.g. state components represented by unbounded integers in the formal design). The system was implemented in SPARK. Information flow contracts, data flow contracts, and gold-level functional specifications expressing the security properties in the form of pre- and postconditions were written and proven with SPARK. Platinum-level functional specifications were not written due to budget constraints and the fact that manual review of the code against the formal design was relatively straightforward. SPARK was also used to check for the absence of certain types of run-time errors. System testing was used to further test the code against the formal design. The effort found that the amount of time spent on system testing was significantly less than would normally be expected and that formal specification was useful for finding errors early in the design process. Independent system reliability testing found zero defects, and for a period of time after delivery, the number of defects found remained at zero. However, [53] discusses two defects in the SPARK code and other issues that were found later, in part due to new developments that allow SPARK to more efficiently check for certain run-time errors that were too time-consuming to check for in the original effort. This includes checks for arithmetic overflow, which was the cause of one of the defects not found in the original effort.

Other cyber security-related efforts include the Muen x86/64 separation kernel [54], which is written in SPARK. A separation kernel aims to ensure cyber security properties such as confidentiality, integrity, and availability by providing an execution environment in which processes only have access to specified resources, only communicate with each other according to a specified policy, and are otherwise isolated. For Muen, SPARK analysis was used to prove full absence of run-time errors and some functional specifications expressed as postconditions. Similarly, an effort to develop

a verified SPARK kernel for the Genode operating system framework is underway [55]. SPARK was also used to create SPARKSkein [56], a reference implementation of the Skein cryptographic hash algorithm, one of the hash functions considered for the SHA-3 standard.[4] SPARKSkein was implemented based on the Skein specification and existing C reference implementation, and SPARK was used to check for run-time errors, which uncovered a subtle corner-case error that persists in the C reference implementation. Reference test cases were also used to check the code, which uncovered an additional specification-related error due to a typo in a constant. A more extensive effort along these lines is SPARKNaCl, a SPARK implementation of the NaCl cryptographic library for which SPARK has been used to prove absence of run-time errors and certain functional correctness properties [57]. As a final example, the RecordFlux framework [58] provides a domain specific language (DSL) for formally modeling binary protocol message formats, including invariant relations and dependencies between message fields, and a method to automatically generate SPARK implementations of message parsers from message models. SPARK can then be used to prove absence of run-time errors in the message parsers and certain gold-level functional specifications, (e.g. that optional message fields can be accessed if and only if all requirements defined in the specification are met). Demonstrations of RecordFlux include Ethernet frames, with a model that covers both the IEEE 802.3 and Ethernet II standards; Fizz, a TLS 1.3 implementation developed and used by Facebook; and a correct implementation of the TLS Heartbeat extension that was responsible for the Heartbleed vulnerability.

[4] `http://www.skein-hash.info/`

2.5 CWEs and SPARK

This section is based on a portion of a ready to publish IET chapter contribution.[5] SPARK is able to verify that code is free of many types of security vulnerabilities, including many of those identified by the MITRE corporation. MITRE maintains two databases relevant to cyber security. One is the Common Weakness Enumeration (CWE) [59], a database that categorizes common types of software and hardware weaknesses, giving each a unique CWE number. The other is the Common Vulnerabilities and Exposures (CVE) [60], which tracks publicly known security vulnerabilities and their root causes in terms of CWEs. For example, the Heartbleed OpenSSL vulnerability is CVE-2014-0160 and is caused by CWE-126 (buffer over-read), CWE-125 (out-of-bounds read), CWE-130 (improper handling of length parameter inconsistency), and CWE-843 (access of resource using incompatible type). The CWE also contains a running list of the 25 most dangerous software weaknesses based on CVEs over the last two years. The current list contains many CWEs that map to common run-time errors, (e.g. out-of-bounds reads and writes, improper restriction of operations within the bounds of a memory buffer, and integer overflow or wraparound). These CWEs are easily eliminated by proving SPARK code to the silver level. In fact, "Adacore technologies for cyber security" shows how SPARK can be used to address many CWE weaknesses, some through fundamental stone-level language features and others by proving code to the silver level or beyond [61].

2.6 SPARK Alternatives

Frama-C is an open-source tool that performs static analysis on C programs. Static analysis tools can reason about code without ever executing the source code.

[5] Laura R Humphrey, Ryan Baity, Kenneth Hopkinson chapter 5 contribution to upcoming IET Textbook - Section: An example of SPARK for cyber security.

Frama-C's website claims that it aims to be a tool that will always indicate when a location may cause a run-time error, and allows the user to manipulate functional specifications.[6]

Dafny is another auto-active verification tool similar to Frama-C but utilizes its own intermediate language Boogie. It was created at Microsoft and predictably targets the C# programming language as opposed to Frama-C's, C. Although Frama-C, Dafny, and a few others not mentioned can be used for verification [62, 63], this thesis will utilize and focus on SPARK.

2.7 AdaCore's SPARK Framework

This section will give a deeper view of the SPARK framework. Much of this section is referenced from the book "Building High Integrity Applications with SPARK." This book is a great resource and should be reviewed for a any desired understanding beyond the scope of this thesis [28]. Although many researchers have looked into the use of formal methods, as previously mentioned, it has been slow to be heavily utilized in industry. The creators of SPARK write that this is due to three things:

- Claims that formal methods extend the development cycle.
- They require difficult mathematics.
- They have limited tool support.

SPARK was designed by the AdaCore company to help build high integrity applications that can be formally verified. SPARK also hopes to ease the three points addressed.

[6] `https://frama-c.com/html/overview.html`

2.7.1 SMT Solvers and Flow Analysis Contracts

This section is based on a portion of a ready to publish IET chapter contribution.[7] At a high level, SPARK performs formal verification by translating SPARK programs, including checks and contracts to be verified, to the Why3 deductive program verification platform [64]. Why3 then uses a weakest-precondition calculus to generate verification conditions (VCs) (i.e. logical formulas whose validity would imply soundness of the code with respect to its contracts). Why3 then uses multiple provers, including but not limited to satisfiability modulo theory (SMT) solvers Alt-Ergo [65], CVC4 [66], and Z3 [67], to attempt to prove the validity of the VCs. In this thesis, GNATprove which utilizes Why3 only ends up utilizing the CVC4 and Z3 provers throughout its proofs. To reiterate, though SPARK aims to perform fully automated verification through this process, it is often necessary for the user to provide additional assertions, (e.g. loop invariants and type invariants), to create additional VCs that help guide the provers toward a proof of the original checks and contracts to be verified.

```
1     procedure Append_To_Stream(Message : in String;
2                                Status  : out Boolean)
3     with  Global  => (In_out => Message_Stream),
4           Depends => (Message_Stream => (Message_Stream, Message)
5                       Status =>         (Message_Stream, Message));
```

Figure 1: Example data dependency and data flow contracts.

As previously mentioned, SPARK performs two types of analysis: flow analysis and proof. Proof will be covered in Chapter III, so this thesis will now briefly consider flow analysis. There are two types of contracts relevant to flow analysis: data dependency contracts and flow dependency contracts. Data dependency contracts

[7] Laura R Humphrey, Ryan Baity, Kenneth Hopkinson chapter 5 contribution to upcoming IET Textbook - Section: Background on SPARK.

describe what global data a subprogram depends on and whether that data is read, written, or both. The second is flow dependency contracts, which describe dependencies between a subprogram's inputs and outputs, including global data. Consider the example specification shown in Figure 1. for a procedure `Append_To_Stream` that takes a `Message` string as input, appends it to a global `Message_Stream`, and reports whether the operation was successful through boolean output `Status`. The data dependency contract specified with aspect `Global` indicates that `Message_Stream` has mode `In_Out`, meaning that it is both an input to the procedure and therefore must be initialized before the procedure is called and also an output that the procedure modifies. The flow dependency contract specified with aspect `Depends` indicates that the value of `Message_Stream` after the procedure is called depends both on the current value of `Message_Stream` and the value of `Message`, and the value of `Status` depends on both of these as well. SPARK flow analysis will verify whether or not these contracts hold `True`.

2.7.2 SPARK vs. Ada

Because the SPARK language is a stricter subset of the Ada programming language, the SPARK User's Guide lists out some notable Ada features excluded from SPARK. Adacore calls these restrictions global simplifications to Ada. The following exclusions listed are from Section 5.1.1 of the SPARK User's Guide [68]:

- "Uses of access types and allocators must follow an ownership policy, so that only one access object has read-write permission to some allocated memory at any given time, or only read-only permission for that allocated memory is granted to possibly multiple access objects."
- "All expressions (including function calls) are free of side-effects."

- "Aliasing of names is not permitted."
- "The backward `goto` statement is not permitted."
- "The use of controlled types is not permitted."
- "Handling of exceptions is not permitted."
- "Unless explicitly specified as (possibly) nonreturning, subprograms should always terminate when called on inputs satisfying the subprogram precondition."
- "Generic code is not analyzed directly."

2.7.3 GNAT Community Edition

GNAT Community Edition is a suite that provides an IDE called GNAT Programming Studio (GPS - changing name to GNATstudio), the Ada/SPARK languages, a prover, and a few more tools developed by AdaCore that can be utilized for formal verification of software.[8] It is free to software developers, hobbyists, and students. The IDE provided is similar to any other programming IDE.

2.7.4 GNAT Compiler and GNATprove

The GNAT compiler functions like a typical compiler by checking that written code conforms to the Ada programming language syntax and then generates executable code. The GNAT compiler is fully integrated into GPS upon installation. GNATprove is built specifically to be used with Ada's subset SPARK. GNATprove may be utilized in three distinct modes:

- Check
- Flow

[8] `https://www.adacore.com/community`

- Proof

Check mode simply checks that the code adheres strictly to the SPARK syntax. This is important due to the many differences between what is acceptable in the context of Ada as compared to that of its SPARK subset. As mentioned in Section 2.7.2, one of the most prominent examples being SPARK's lack of support for what AdaCore calls access types (pointers). The lack of access types in SPARK can be seen as a limitation and should be taken into account when debating whether or not to implement your project in SPARK. This also means that if one wanted to transcribe an application from its original language to SPARK, one may be required to restructure his or her code. This could be tedious.

Flow mode simply performs flow analysis on the supplied SPARK code. Proof mode is the most important feature of GNATprove. Proof mode will perform the formal analysis on your SPARK code. McCormick and Chapin write that SPARK checks for code that may raise run-time errors (divide-by-zero, bound checks, etc). GNATprove will also attempt to prove any assertions that the programmer has annotated within the SPARK code. These assertions are known as logical statements and are either `True` or `False` [28]. At a minimum, SPARK programs should be shown to be free of run-time errors. At this point, the program is known to have established an absence of run-time errors (AoRTE) [30].

When GNATprove analyzes both the SPARK code and the assertions annotated by the programmer, it produces logical statements known as conjectures to GNATprove [28]. Conjectures are statements that GNATprove believes to be "true but not yet proven." These conjectures are known as verification conditions (VCs) as mentioned in Section 2.7.1. For a program to be proven correct, all VCs must be discharged (proven correct). If a logical statement is proven true, it is now known as a theorem.

2.7.5 GNAT Academic Program

The GNAT Academic Program (GAP) is a program available to researchers, such as master's students.[9] If the student is utilizing SPARK in his or her research, he or she is encouraged to join GAP and receive access to all the AdaCore GNAT community tools (Ada, SPARK, GNAT Compiler, GNATprove, etc) for free. More notably, GAP members are granted access to AdaCore's SPARK experts and can submit problem tickets at any time. The GAP support team will review the ticket and get back to the student. This program is an invaluable resource for anyone who has research that may involve formal methods and SPARK specifically.

2.7.6 Assurance Levels - Levels of Proof

This subsection expands upon the levels of proof mentioned in Section 2.3. SPARK can give a program different levels of assurance as described by AdaCore and Thales in their "Implementation Guidance for the Adoption of SPARK" [69]. Chapter 2 Section 3 describes the following levels, which are all considered to be higher than the basic Brick Level (Ada code):

1. Stone - valid SPARK code
2. Bronze - initialization and correct data flow
3. Silver - absence of run-time errors (AoRTE)
4. Gold - proof of key integrity properties
5. Platinum - full functional proof of requirements

All five levels build on each other and each level requires all the previous to be met. Consequently, each level requires more effort to complete. The guide goes on to

[9] https://www.adacore.com/academia

articulate that the Stone Level is a must and should be considered an intermediate step during adoption. The developers should attempt to achieve a Bronze Level of assurance for most of the code. The Silver Level should be the goal and expectation for all "critical software," and the Gold Level should represent "a subset of the code subject to specific key integrity (safety/security) properties." Essentially, the more important the section of code, the higher the level of assurance that should be associated with it, but the importance of the code is decided by the developers.

2.7.7 SPARK Formal Method Examples

All of the examples in this section come from two SPARK guides because they are designed to be an introduction to SPARK [29, 68]. For more information on any of the following, look into the guides for more details [29, 68]. The following three items are discussed in this section:

- Contracts
- Loop Invariant
- Ghost Code

2.7.7.1 Contracts

Contracts are critical in specifying how a program should preform in SPARK. Contracts are applied to what Ada/SPARK calls subprograms. One can think of a subprogram as a function or method in any other language. This means that in Ada and SPARK, contracts are applied to procedures (which do not return anything - similar to a `void` function in other languages) and functions (do return something). Contracts are typically comprised of a combination of five parts. The following definitions are directly from AdaCore [68]:

1. (precondition) - uses `Pre` to "[specify] constraints on callers of the subprogram."
2. (postcondition) - uses `Post` to "[specify] (party or completely) the functional behavior of the subprogram."
3. (contract cases) - uses `Contract_Cases` to "partition the behavior of a subprogram. It can replace or complement a precondition and postcondition."
4. (data dependencies) - uses `Global` to "specify the global data read and written by the subprogram."
5. (flow dependencies) - uses `Depends` to "specify how subprogram outputs depend on subprogram inputs."

It is important to note that all the above optional components of a contract are typically used in testing and development. Although this is usually the case, one may want the contracts to be checked during regular execution as well. This check can be accomplished by applying the `-gnata` flag in GNAT when using the command line. It is also called when pressing "prove" in the GNAT Community IDE with assertions enabled. Checking the contracts during execution of a SPARK program that fails a check raises an exception if something like a precondition fails. This means that the code within the subprogram will not execute making any necessary debugging much more straight forward. Rather than trying to work through the subprogram to find the issue, the user will notice something like the following [29]:

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE: failed precondition form
```

Figure 2 has two examples pulled from the "SPARK 2014 User's Guide" that showcase pre- and postconditions and contract cases [68]. `Add_To_Total` is a simple function that adds a value (`Incr`) to a global variable named `Total`. The first procedure has a precondition which specifies the number to be incremented by must be 0 or greater before the start of the function and if that is `True` then check to see if

`Total` is in the range between 0 to `Integer'Last - Incr`, where `Integer'Last` is the largest possible value of an Integer, to avoid an Integer overflow. Using `and then` short circuits the logic whereas `and` would evaluate both sides.

```
1     procedure Add_To_Total (Incr : in Integer) with
2        Pre => Incr >= 0 and then Total in 0 .. Integer'Last - Incr;
3        Post => Total = Total'Old + Incr;
4
5     procedure Add_To_Total (Incr : in Integer) with
6        Contract_Cases =>
7           (Total+Incr < Threshold  => Total = Total'Old + Incr,
8           Total+Incr >= Threshold => Total = Threshold);
```

Figure 2: Example SPARK Contracts.

The second procedure (in Figure 2) is another approach at a contract for the `Add_To_Total` procedure. Instead of using pre- and postconditions, this implementation explores contract cases. To reiterate, contract cases are utilized when a subprogram has an input space that can be divided into mutually exclusive regions, where the desired behavior of the subprogram can be specified for each region. In this procedure there are two cases, delimited by a comma. This contract comprises two cases. First, after being incremented `Total` is still less than a set globally known `Threshold`, "in which case" the procedure adds `Incr` to `Total`. The second case is when the `Incr` + `Total` exceeds or is equal to the specified `Threshold`, "in which case" `Total` is set equal to `Threshold`.

2.7.7.2 Loop Invariants

SPARK and all other formal verification toolsets need additional annotations from the user due to the way that formal methods tools analyze loops. They process each iteration of a loop independently, which means that the tools cannot reason based on past or future iterations. A loop invariant works by checking that it holds for the first iteration, then for any iteration. This method is essentially proof by induction.

Sometimes providing a loop invariant helps formal methods tools prove an otherwise un-provable postcondition. Figure 3 is a simple example of a loop invariant in SPARK from the "Loop Invariant" section of a SPARK guide [29], which just checks that no elements of `A` are equal to `E`.

```
 1  function Find (A : Nat_Array; E : Natural) return Natural is
 2  begin
 3     for I in A'Range loop
 4        pragma Loop_Invariant
 5        (for all J in A'First .. I - 1 => A (J) /= E);
 6        if A (I) = E then
 7           return I;
 8        end if;
 9        end loop;
10     pragma Assert (for all I in A'Range => A (I) /= E);
11     return 0;
12  end Find;
```

Figure 3: Example SPARK Loop Invariant.

This loop and its loop invariant are followed by a simple `pragma Assert` to ensure check that no elements in `A` are equal to `E`. As mentioned earlier, assertions are predicates typically placed in code where the programmer thinks that it will always evaluate to true. An assertion is similar to a loop invariant, but a loop invariant is a specific type of assertion that can only be used within a loop and asserts something that should be `True` before, during, and after a loop.

2.7.7.3 Ghost Code

Ghost code in SPARK is code that has no effect on the functionality of the program. Instead, it is typically utilized for testing and verification. For this reason, when executing with the flag `-gnata` ghost code will be executed along with the functional code in the Ada/SPARK program. There are many different variations of ghost code to include ghost functions, used to express properties used in functions, ghost variables (global and local), to track of program state, and more [68]. To specify

something as ghost, one would type `with Ghost;` following what is to be considered ghost code. As an example, if one wanted a function `Double_Array` that takes in an integer array type called `IArray` and returns an `IArray` with each index doubled to be a ghost function, this is what that function specification would look like:

```
function Double_Array(A : IArray) return IArray with Ghost;
```

2.8 SPARK's role in the Software Development Lifecycle

This section will briefly discuss the role and location of SPARK development as it applies to the Software Development Life Cycle (SDLC).[10]

1. Requirement analysis
2. Planning
3. Software design
4. Software development
5. Testing
6. Deployment
7. Maintenance

Software development via SPARK is special because it integrates testing into the first four phases listed above. This is because in the requirements analysis and planning phases, the developer must already start thinking about the contracts required between the interfaces of the subprograms to be implemented. These contracts must then of course be incorporated in the software design as well, leading to the implementation of said contracts in the software development phase. Once the developers

[10] `https://stackify.com/what-is-sdlc/`

reach the testing phase, much of the pure functionality testing is complete. With respect to the testing phase, the type of formal verification preformed by SPARK is equivalent to exhaustive unit testing. It can therefore prove the absence of errors, whereas as traditional testing over a limited number of inputs can only reveal their presence (examples of this were discussed in Section 2.4). The developer focuses on integration testing, but he or she can rely upon the proven contracts developed and implemented in the first four steps as a means for certifying core functionality.

2.9 Summary

This chapter provided information about the formal methods tool SPARK, its makeup, its applications, and some of its previous projects. With the context and information provided in this chapter, Chapter III's use of the SPARK formal methods framework can be much more easily understood and comprehended.

III. Methodology

3.1 Preamble

This chapter proves the implementations of three well known algorithms. All formal verification in this chapter is accomplished strictly with the SPARK auto-active toolset for formal verification described in Chapter II. Each section first describes a basic implementation of the three algorithms. These basic implementations are first written in SPARK without any contracts (below the gold level). Each section then presents updated implementations with user-added contracts such as pre- and postconditions, loop invariants, and assertions. This chapter describes the formal verification of interpolation (Section 3.2), merge sort (Section 3.3), and priority queue (Section 3.4). The following steps outline the general process for what is required to implement an algorithms via the SPARK formal methods toolset:

1. Understand SPARK and its syntax.
2. Understand the algorithm to be proven with SPARK.
3. Implement the algorithm to a basic level in SPARK (core functionality).
4. In English, identify what the algorithm should do and should not do (define its specification).
5. Turn the English properties into SPARK contracts that can be proven by GNATprove.
6. Run SPARK's GNATprove and then make additional annotations as needed to prove contracts.
7. If applicable, integrate proven SPARK code back into its origin software system.

This general process is utilized on the three algorithms discussed in this chapter (Section 3.2, Section 3.3, Section 3.4). It is assumed that step one is already complete, so that the thesis can focus on the remaining steps. Each section in this chapter starts with a quick review of the algorithm and then moves on to defining the basic implementations in SPARK (step three) before moving through the remaining steps. Each section then concludes with a proven SPARK implementation of the algorithm.

3.2 Verifying an Interpolation Algorithm in SPARK

This section is based on a portion of a ready to publish IET chapter contribution.[1] To provide an entry-level conceptual understanding of SPARK and its process, this section goes through the process of implementing a simple interpolation algorithm to the gold level. While it is the case that SPARK is very useful when coding up a project from scratch, one may receive a snippet of Ada/SPARK code and have to formally verify it. This example starts with an imperfect implementation of the interpolation algorithm, goes through its faults, fix them, then moves into formally verifying the code. This example utilizes AdaCore's Ada/SPARK integrated development environment (IDE) mentioned in Section 2.7.3, GPS. GPS has all the needed tools built-in to modify the code, add formal contracts, run the provers on the SPARK interpolation implementation, and formally verify it.

Before moving into the interpolation example implementation, this thesis reviews what interpolation is. From a mathematical standpoint, interpolation utilizes a discrete set of data points to estimate the value of a hypothetical point that lies between two consecutive discrete points. For example, if you have points (0, 0) and (2, 2), interpolation would determine that the Y-value at $X = 1$ is $Y = 2$. Hence in this example, the discrete points (0, 0) and (2, 4) provide that an interpolation at $X = 1$

[1] Laura R Humphrey, Ryan Baity, Kenneth Hopkinson chapter 5 contribution to upcoming IET Textbook - Section: An example of SPARK for cyber security.

would estimate that for (1, ?), $X = 1$ lies equally between $X = 0$ and $X = 2$, so its Y-value would too lie equally between $Y = 0$ and $Y = 4$, respectively. Of course, for this to work, the distance of X from both closest known X-values from the discrete set of known points must be taken into account. This example uses equation Equation (1) to interpolate the desired Y-value based on a supplied and previously unknown X-value. Where a is the user-supplied X-value, where x_1 and y_1 are the closest known X and Y-value less than a, where x_2 and y_2 are the closest known X and Y-value greater than a, and finally, where y is the resulting interpolated Y-value.

$$y = y_1 + (a - x_1)\,\frac{(y_2 - y_1)}{(x_2 - x_1)} \tag{1}$$

To review, the mathematical function $y = f(x)$ is to be estimated as a piecewise linear function over a sequence of points (x_i, y_i) for $i = \{1, \ldots, n\}$. Suppose the points are ordered such that $x_i < x_{i+1}$ for $i = \{1, \ldots, n-1\}$. Then using linear interpolation, the value $f(a)$ is estimated as follows:

- If $a < x_1$, then $f(a) = f(x_1) = y_1$
- If $a > x_n$, then $f(a) = f(x_n) = y_n$
- If $a = x_i$ for some i, then $f(a) = f(x_i) = y_i$
- Otherwise, $f(a) = y_i + (a - x_i)\frac{(x_{i+1} - x_i)}{(y_{i+1} - y_i)}$ for $x_i < a < x_{i+1}$

3.2.1 Interpolation Implementation

This section now looks at the implementation. Figure 4, Figure 5, and Figure 6 represent the initial implementation that needs to be annotated with contracts and checked for correctness. Each of the three figures are called a package in Ada/SPARK. This is a modularization unit of code similar to the Java class and the C++ header and implementation pair. SPARK, like its superset Ada, uses an Ada specification

file (.ads), which starts with the first line `package [package_name]`. This serves as the specification file. The first code snippet is the specification file (Figure 4) in this example. It specifies types, subtypes, procedures, and functions. Note that the X-value is defined as an Integer subtype named `Arg` and the Y-values as an Integer subtype named `Value`. And when (`Arg`, `Value`) are paired together, it results in a `Point`. `Index` is a type defined as a range from 1-100. A type called `Func` is also defined. `Func` is defined as an array of `Points` where the index range can be from 1-100 as long as the first index starts at index 1. The `Increasing` function is also defined and should return `True` only if the `Value`s of each `Point` are monotonically increasing. Next, `Monotonic_Incr_Func` is defined by predicating it on the earlier defined `Increasing` function. The intent of this `Monotonic_Incr_Func` is to specify a monotonically increasing function. Next, in Figure 5 SPARK uses an Ada body file (.adb) to provide implementations of functions and procedures defined in the specification file. These .adb files start with the first line `package body [package_name]`. As can be seen in the Interpolation package body (Figure 5), the `Eval` function is implemented. This is where the interpolation calculation occurs in this example. Equation (1) is implemented in the `elsif` on lines 13–17 of the .adb file. Figure 6 shows a simple main package for exercising the interpolation function `Eval` through procedure `Test_Interpolation`. This package sets up a nominal test case with a constant `F` of type `Func` containing points ((-10, -10), (-1, -3), (0, 6), (5, 12), (12, 12), (18, 12), (20, 15)). Note that the Y-values of `F` satisfy `Increasing`, so that when `Eval` is called on `F`, `F` can be converted to subtype `Monotonic_Incr_Func` without violating its predicate and raising a run-time error. This test case returns the expected result, printing the following Y-values for X-values ranging from -20 to 10: -10, -10, -9, -8, -7, -7, -6, -5, -4, -3, 6, 7, 8, 9, 10, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 15.

```
1  package Interpolation with SPARK_Mode is
2
3     subtype Arg is Integer;
4     subtype Value is Integer;
5     type Point is record
6        X : Arg;
7        Y : Value;
8     end record;
9
10    type Index is new Integer range 1 .. 100;
11    type Func is array (Index range <>) of Point with
12       Predicate => Func'First = 1;
13
14    function Increasing (F : Func) return Boolean is
15       (for all I in F'Range =>
16          (for all J in F'Range =>
17             (if I < J then F(I).Y <= F(J).Y)));
18
19    subtype Monotonic_Incr_Func is Func with
20       Predicate => Increasing (Monotonic_Incr_Func);
21
22    function Eval (F : Monotonic_Incr_Func; A : Arg) return Value;
23
24 end Interpolation;
```

Figure 4: Initial `interpol.ads`

```
1  package body Interpolation with SPARK_Mode is
2
3     function Eval (F : Monotonic_Incr_Func; A : Arg) return Value is
4     begin
5        if A < F(1).X then
6           return F(1).Y;
7        end if;
8
9        for K in F'Range loop
10          if A = F(K).X then
11             return F(K).Y;
12          elsif A > F(K).X and A < F(K+1).X then
13             declare
14                DX : constant Integer := F(K+1).X - F(K).X;
15                DY : constant Integer := F(K+1).Y - F(K).Y;
16             begin
17                return F(K).Y + (A - F(K).X) * DY / DX;
18             end;
19          end if;
20       end loop;
21       return F(F'Last).Y;
22    end Eval;
23
24 end Interpolation;
```

Figure 5: Initial `interpol.adb`

```
 1  with Ada.Text_IO; use Ada.Text_IO;
 2  with Interpolation; use Interpolation;
 3
 4     procedure Test_Interpolation is
 5        F : constant Func :=
 6        ((-10, -10), (-1, -3), (0, 6), (5, 12), (12, 12), (18, 12), (20, 15));
 7     begin
 8        for A in -10 .. 20 loop
 9           Put_Line ("Eval X =" & A'Image & " Y =" & Eval(F, A)'Image);
10        end loop;
11  end Test_Interpolation;
```

Figure 6: Initial `Test_Interpolation.adb`

This section has now successfully presented a simple interpolation algorithm candidate. There are many issues with the initial interpolation implementation provided. At this point, it is advisable to run the SPARK provers on the code to highlight potential issues. On this initial implementation, SPARK and its prover engine, GNATprove, returned five items of concern. In conjunction with SPARK, CodePeer, a tool available to the SPARK Pro platform, can be used to find vulnerabilities and link them to a Mitre Common Weakness Enumeration (CWE) entry (discussed in Section 2.5). CodePeer also ran against the initial implementation and found the same five items as SPARK, but this time with mappings to a CWE. All of the SPARK prover warning items and CodePeer CWE flags are found when run on the initial `interpol.adb` as represented in Figure 5. The five items are as follows:

1. [Line 7] – [CWE 118]: array index check might fail if array empty
2. [Line 16] – [CWE 119]: array index check might fail at `F(K+1)`
3. [Line 18] – [CWE 190]: overflow check might fail at subtraction
4. [Line 19] – [CWE 190]: overflow check might fail at subtraction
5. [Line 21] – [CWE 190]: overflow check might fail at multiplication

It is now time to go through the issues and provide solutions as a means to ensure the correctness of this code via formal verification using SPARK. All the following changes to Figure 4 and Figure 5 are reflected in Figure 7 and Figure 8 and correspond to the five items SPARK enumerated. The main driver `Test_Interpolation.adb` is still the same as Figure 6. This interpolation example cannot be a formally verified example until a few flaws with the original setup and specification are addressed. First are the changes reflected in the updated body file (Figure 8), which includes the interpolation calculation. There are four changes to the flow of the code:

1. If `A` is equal to the first value in `F`, not just less than, this condition holds `True`. This update can be seen on lines 4-5 of Figure 8.

2. The code now takes into account if `A` (the provided X-value to interpolate) is greater than or equal to the last value in the `F`. This update can be seen on lines 6-7 of Figure 8.

3. The interpolated value is now fully calculated within the `declare` section. This update can be seen on line 18 of Figure 8.

4. The final return in the body of the `Eval` function is altered to raise a program error, because the code should never reach this point if the code is implemented correctly. Remember, at the beginning of the `Eval` function `A` is checked to see if it is at the end of the `Monotonic_Incr_Func`. This update can be seen on line 26 of Figure 8.

Next are the changes reflected in the updated specification file (Figure 7). There are three changes to the general specifications in this file:

1. The ranges for subtypes `Arg` and `Value` are restricted to the range `-20_000 .. 20_000` to prevent overflows in the interpolation arithmetic. This update can be seen on lines 3–4 of Figure 7.

2. The definition of the type `Func` is expanded to include that any sequence of `Point`s in `Func` must have increasing X-values to be considered a `Func`. In other words, X is always increasing from `Point` to `Point`, and there are no duplicate X-values. This update can be seen on lines 13–15 of Figure 7.

3. The name of the `Increasing` function is updated to `Monotonic_Increasing` to precisely describe the desired behavior it aims to check. This is because the `Func` that is passed into its parameter does not need to be strictly increasing. The value of a `Point` can remain steady as long as it does not decrease as it progresses through the range in a positive direction. This update can be seen on line 17 of Figure 7.

Now that the appropriate modifications have been made to the core functionality of the interpolation example code, this thesis can move into the formal specification of the `Eval` function. The goal is to formally specify this function specifically because this is the function that actually interpolates a Y-value given a particular X-value (`A`). To accomplish this task, there needs to be a postcondition that is user-defined on the function. The user has to communicate to SPARK what the correct functional behavior of code should be. First, add a contract in the form of a precondition that states that there must be at least one element in `Func` F. This prevents one possible error in the `Eval` body. In particular, line 4 of Figure 8 would result in an error if checking the first index of an empty `Func` array. This added precondition can be seen on line 26 of Figure 7. Additionally, in this interpolation example, there are only four desired interpolation results based on parameter `A`'s (supplied X-value) relationship to the `Func F` supplied to `Eval`. `A` is one of the following and checks in order:

1. Less than or equal to the first element's X-value

2. Greater than or equal to the last element's X-value

3. Equal to an internal element's X-value

4. Lies between two element's X-value

If `A` satisfies condition 1, then Y (the interpolation result) equals the Y-value of the first element. If `A` satisfies condition 2, then Y equals the Y-value of the last element. If `A` satisfies condition 3, then Y equals that specific internal element's Y-value. Finally, if `A` satisfies condition 4, then Y lies within the range of `A`'s two closest elements. Notice, the fourth condition only checks that the interpolated Y value is in the correct range but it does not check for the exact value as defined by the interpolation equation (Equation (1)). Because this criteria only checks the range of the result, this is a gold level proof rather than a full platinum level proof. These four conditions are implemented as a contract in the form of a postcondition. This update can be seen on lines 27-33 of Figure 7.

3.2.2 Loop Invariants and Assertions Needed

At this point, SPARK is run again to ensure that the postcondition just added is sound. As the code stands, SPARK returns the following about line 27 in Figure 7: "postcondition might fail." This warning could mean one of three things. One, it could mean the programmer must provide the provers with more information. Two, it could mean that the code itself has been unintentionally implemented incorrectly. Three, it could mean that the specification is something other than what was intended. But if the provers just need more information, one common issue is that the function corresponding to the SPARK "postcondition might fail" warning contains a loop within its implementation. Recall, `Eval` is implemented in Figure 5 (initial `interpol .adb`) and does indeed contain a for loop. SPARK provers have issues working with loops because they can only reason about each iteration of the loop independently. In other words, without help, the provers do not know what happened on any other

```
1  package Interpolation with SPARK_Mode is
2
3     subtype Arg is Integer range -20_000 .. 20_000;
4     subtype Value is Integer range -20_000 .. 20_000;
5     type Point is record
6        X : Arg;
7        Y : Value;
8     end record;
9
10    type Index is new Integer range 1 .. 100;
11    type Func is array (Index range <>) of Point with
12       Predicate => Func'First = 1 and
13       (for all I in Func'Range =>
14          (for all J in Func'Range =>
15             (if I < J then Func(I).X < Func(J).X)));
16
17    function Monotonic_Increasing (F : Func) return Boolean is
18       (for all I in F'Range =>
19          (for all J in F'Range =>
20             (if I < J then F(I).Y <= F(J).Y)));
21
22    subtype Monotonic_Incr_Func is Func with
23       Predicate => Monotonic_Increasing (Monotonic_Incr_Func);
24
25    function Eval (F : Monotonic_Incr_Func; A : Arg) return Value with
26       Pre => F'Length > 0,
27       Post => (if A <= F(1).X then Eval'Result = F(1).Y
28          elsif A >= F(F'Last).X then Eval'Result = F(F'Last).Y
29          elsif (for some K in 1..F'Last => A = F(K).X and then
30             Eval'Result = F(K).X) then True
31             else (for some K in 1..F'Last - 1 =>
32                A in F(K).X..F(K+1).X and then
33                Eval'Result in F(K).Y..F(K+1).Y));
34
35    pragma Annotate (GNATprove, Terminating, Eval);
36 end Interpolation;
```

Figure 7: Corrected and extended specification `interpol.ads`

arbitrary iteration of the loop. The user must supply user contracts such as loop invariants and/or assertions to help push the provers to an inductive proof of the loop. In this example, the loop works by iterating through each element of the array of `Points`. In order for SPARK to inductively prove this section of code, the lower bound at each iteration is needed to make sense of the past iterations of the loop. For this reason, the loop invariant `A >= F(K).X` is added at the start of the loop on line 11 of Figure 8. This loop invariant helps because it establishes that the value of `A` could not have been between `F(K).X` and `F(K+1).X` for values of `K` in earlier iterations of the loop (i.e. that the loop did not somehow erroneously "skip over" the points

that `A` lies between on a previous iteration). Additionally, it is important to indicate to the provers that the second condition of the loop, which handles an X-value that lies between two known elements, should indeed return an interpolated Y-value that lies between the Y-value of those two surrounding `Points`. This assertion can be seen on line 20 of the updated interpolation body file (Figure 8).

```
1  package body Interpolation with SPARK_Mode is
2     function Eval (F : Monotonic_Incr_Func; A : Arg) return Value is
3     begin
4        if A <= F(1).X then
5           return F(1).Y;
6        elsif A >= F(F'Last).X then
7           return F(F'Last).Y;
8        end if;
9
10       for K in F'Range loop
11          pragma Loop_Invariant(A >= F(K).X);
12          if A = F(K).X then
13             return F(K).Y;
14          elsif (A > F(K).X and then A < F(K+1).X) then
15             declare
16                DX : constant Integer := F(K+1).X - F(K).X;
17                DY : constant Integer := F(K+1).Y - F(K).Y;
18                R  : constant Integer := F(K).Y + (A - F(K).X) * DY / DX;
19             begin
20                pragma Assert(R in F(K).Y..F(K+1).Y);
21                return R;
22             end;
23          end if;
24       end loop;
25
26       raise Program_Error;
27    end Eval;
28 end Interpolation;
```

Figure 8: Corrected `interpol.adb`

When running the SPARK provers once more on the newly updated code (Figure 7 & Figure 8), the provers terminate successfully. This means that the provers are able to prove the user-defined contracts and specifications and guarantee key functional correctness as defined in the pre- and postconditions. Additionally, CodePeer reanalyzed the formalized code and found no CWE risks are present.

3.3 Verifying a Merge Sort Algorithm in SPARK

This section is based on a previously published paper [70].[2] This section goes through the process of verifying SPARK code to the gold level through an example, namely a recursive merge sort algorithm. Though verification of such a simple, well-known algorithm may seem like merely an academic exercise, it is interesting to note that subtle errors in similar algorithms have gone undetected for long periods of time. For example, the Java Development Kit (JDK) implementation of a binary search over arrays contained a run-time error due to overflow that went undetected for nine years [71]. Verification at the silver level or higher in SPARK would have revealed the error.

3.3.1 Merge Sort Implementation

Figure 9 depicts a recursive merge sort algorithm. The underlying procedure splits an array into left and right halves, then recursively calls itself on each half. The base case is reached when the procedure is called on an array of size 1. Left and right halves are merged into a sorted array by taking elements in order from the left and right halves, and the merged and sorted array is returned. Note this means that when the procedure goes to merge the left and right halves returned by lower-level calls, it assumes each half is sorted.

Consider a SPARK implementation of this algorithm that sorts arrays of integers. The implementation of the driver (`main.adb`) can be seen in Appendix B under Listing B.1. SPARK is a strongly typed language, so the first step is to begin by considering the types to be used in this implementation. A common SPARK design pattern is to have a separate package that specifies the types used in a program. Here, this thesis defines the types for the merge sort algorithm in a `mergesort_types`

[2] Ryan Baity, Laura R Humphrey, Kenneth Hopkinson. Formal verification of a merge sort algorithm in SPARK. In *AIAA Scitech 2021 Forum*, page 0039, 2021. DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. Case #88ABW-2020-3580.

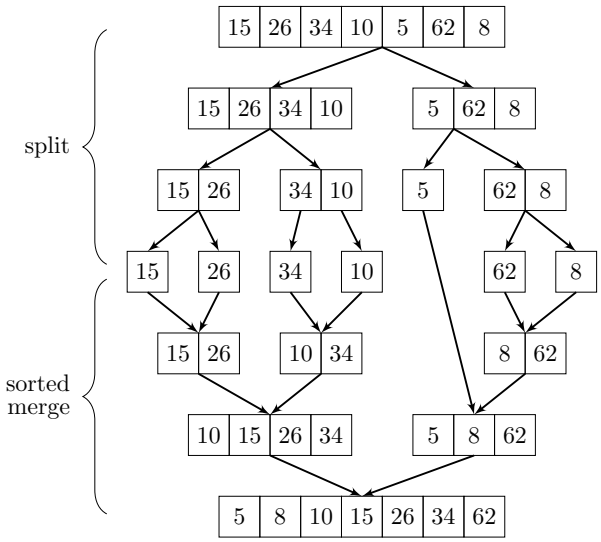


Figure 9: A visual depiction of a merge sort algorithm.

```
1  package mergesort_types with SPARK_Mode is
2     subtype Sort_Index is Integer range 0 .. 99;
3     type Sort_Array is array(Sort_Index range <>) of Integer;
4     procedure Print(A : Sort_Array);
5  end mergesort_types;
```

Figure 10: SPARK package specification for the types used by this merge sort algorithm.

package, shown in Figure 10. Line 1 defines the keywords `with SPARK_Mode` in the package specification and indicates that it should contain valid SPARK code. Line 2 defines the subtype `Sort_Index` as a subtype of standard type `Integer` that is constrained to range from 0 to 99. Line 3 defines the type `Sort_Array` as an array type indexed by type `Sort_Index` that holds elements of type `Integer`. The notation `Sort_Index range <>` indicates that this is an unconstrained array type (i.e. a variable of this type can be instantiated with arbitrary first and last index values from subtype `Sort_Index`). For example,

```
1  A: Sort_Array(0 .. 3) := [2, 4, 6, 8];
2  B: Sort_Array(51 .. 52) := [3, 4];
```

```
3  C: Sort_Array(5 .. 9);
```

are all valid. It would be possible to make this a generic package so that a user of the package could choose the range constraints for `Sort_Index`, enabling `Sort_Array` types of arbitrary maximum size. But for simplicity, the arrays are limited to have indices with a maximum range of 0 to 99 (i.e. arrays with a maximum size of 100 elements). Line 4 defines the procedure `Print` as a function that prints supplied `Sort_Index` array `A`. This `Print` procedure is for console output and is implemented in the `mergesort_types` body file and can be viewed in Appendix B in Listing B.5.

```
1  with mergesort_types; use mergesort_types;
2
3  package mergesort_algorithm with SPARK_Mode is
4
5     procedure recursive_mergesort(A: in out Sort_Array; L, R: Sort_Index);
6
7     procedure merge(A: in out Sort_Array; L: Sort_Index;
8                     M: Sort_Index; R: Sort_Index);
9
10 end mergesort_algorithm;
```

Figure 11: Basic SPARK package specification for procedure `recursive_mergesort` and helper procedure `merge`.

Now consider the package for implementing the merge sort algorithm itself, called `mergesort_algorithm`. While the `mergesort_types` package consists of only a specification, packages containing subprograms more commonly consist of both a specification and a body. The basic specification at the bronze level for procedures in package `mergesort_algorithm` with no contracts is shown in Figure 11. The *with clause* on line 1 provides access to package `mergesort_types`, and the *use clause* provides direct visibility to public declarations within the package without having to prefix them with the package name. As in the `mergesort_types` package, the keywords `with SPARK_Mode` on line 3 specify that the package is to contain SPARK code. Line 5 defines the specification for procedure `recursive_mergesort`. Its first formal

parameter `A` is of type `Sort_Array` and has mode `in out`, meaning that `A` must be initialized before the procedure is called, and `A` is modified by the procedure. The second and third formal parameters `L` and `R` are of type `Sort_Index` and implicitly have mode `in` by default, meaning they cannot be modified by the procedure. The goal of `recursive_mergesort` is to sort the values in subarray `A(L .. R)`. Lines 7-8 define the specification for helper procedure `merge`. The formal parameters are the

```
1  with mergesort_types; use mergesort_types;
2
3  package body mergesort_algorithm with SPARK_Mode is
4
5     procedure recursive_mergesort(A: in out Sort_Array; L, R: Sort_Index) is
6                                   M: Sort_Index;
7     begin
8        if (L < R) then
9           M := L+(R-l)/2;
10          recursive_mergesort(A, L, M);
11          recursive_mergesort(A, M+1, R);
12          merge(A, L, M, R);
13       end if;
14    end recursive_mergesort;
15
16    procedure merge(A: in out Sort_Array; L: in Sort_Index;
17                    M: in Sort_Index; R: in Sort_Index) is
18       n1: constant Natural := M - L + 1;
19       n2: constant Natural := R - M;
20       L_temp: constant Sort_Array(0..n1-1) := A(L .. M);
21       R_temp: constant Sort_Array(0..n2-1) := A(M+1 .. R);
22       ii, jj, kk: Natural := 0;
23    begin
24       while ii < n1 and jj < n2 loop
25          if L_temp(ii) <= R_temp(jj) then
26             A(L + kk) := L_temp(ii);
27             ii := ii + 1;
28          else
29             A(L + kk) := R_temp(jj);
30             jj := jj + 1;
31          end if;
32          kk := kk + 1;
33       end loop;
34
35       if ii < n1 then
36          A(L + kk .. R) := L_temp(ii .. n1-1);
37       elsif jj < n2 then
38          A(L + kk .. R) := R_temp(jj .. n2-1);
39       end if;
40    end merge;
41
42 end mergesort_algorithm;
```

Figure 12: Basic SPARK package body for procedure `recursive_mergesort` and helper procedure `merge`.

same, except there is an additional parameter `M` of type `Sort_Index`. The goal of this procedure, which assumes the contents of `A(L .. M)` and `A(M+1 .. R)` are each already separately sorted, is to merge the two into `A(L .. R)` so that the contents of `A(L .. R)` are sorted after the procedure completes.

Now consider the body of package `mergesort_algorithm`, shown in Figure 12. The procedure `recursive_mergesort` calculates the midpoint of the array, then recursively calls `recursive_mergesort` to sort the left and right array halves. It then calls `merge` to combine the resulting sorted halves back into a sorted array. Procedure `merge` declares several local variables on lines 18-22. `Natural` constants `n1` and `n2` store the size of the left and right subarrays to be merged. `Sort_Array` constants `L_temp` and `R_temp` store copies of the left and right portions of `A` to be merged. `Natural` variables `ii`, `jj`, and `kk` track the position of the algorithm as it increments through values stored in `L_temp`, `R_temp`, and `A`. Note that these are standard type `Natural` rather than type `Sort_Index`, since on the last iteration of the main loop, they can potentially be one larger than the maximum value allowed by `Sort_Index`. The main loop puts values from `L_temp` and `R_temp` back into `A` starting at `A(L)` until it reaches the end of exactly one of either `L_temp` and `R_temp`. The final `if` statement takes the remaining elements from the temp array that still has un-merged elements and copies them into the end of `A`.

Now consider how to formally specify key behaviors of these procedures using pre- and postconditions in SPARK. For convenience, first, add a function to the specification for package `mergesort_algorithm` that returns `True` if and only if an array is sorted. This function can be entirely defined in the specification using what is known in SPARK as an expression function. Its definition is:

```
1 function Is_Ascending(A: Sort_Array) return Boolean is
2    (if A'Length > 1 then (for all I in A'Range =>
3       (if I < A'Last then A(I) <= A(I + 1))))
```

```
4     with Ghost;
```

This function is marked with aspect `Ghost` to indicate that it is mainly used for proof purposes. That is, it is only compiled and included in an executable if the user indicates to the compiler that it should be (e.g. to check the code through test rather than verify it through proof).

```
1  with mergesort_types; use mergesort_types;
2
3  package mergesort_algorithm with SPARK_Mode is
4
5     function Is_Ascending(A: Sort_Array) return Boolean is
6        (if A'Length > 1 then (for all I in A'Range =>
7           (if I < A'Last then A(I) <= A(I + 1))))
8     with Ghost;
9
10    procedure recursive_mergesort(A: in out Sort_Array; L, R: Sort_Index)
          with
11       Pre => A'Length >= 1
12          and then (L in A'Range and R in A'Range)
13          and then L <= R,
14       Post => Is_Ascending(A(L..R))
15          and (for all I in A'Range => (if I not in L..R then A(I) = A'Old(I)
                ))
16          and (for all I in A'Range => (for some J in A'Range => A(I) = A'Old
                (J)));
17
18    procedure merge(A: in out Sort_Array; L: Sort_Index;
19                    M: Sort_Index;        R: Sort_Index) with
20       Pre => (L in A'Range and R in A'Range)
21          and then L <= R
22          and then M in L..R
23          and then Is_Ascending(A(L..M))
24          and then Is_Ascending(A(M+1..R)),
25       Post => Is_Ascending(A(L..R))
26          and (for all I in A'Range => (if I not in L .. R then A(I) = A'Old(
                I)))
27          and (for all I in A'Range => (for some J in A'Range => A(I) = A'Old
                (J)));
28
29 end mergesort_algorithm;
```

Figure 13: SPARK package specification for procedure `recursive_mergesort` and helper procedure `merge` with behavioral contracts expressed using ghost function `Is_Ascending`.

With this ghost function defined, it is possible to write pre- and postconditions for procedures `recursive_mergesort` and `merge` using the function `Is_Ascending`. Figure 13 shows the modified specification for package `mergesort_algorithm`. On

lines 11–13, the precondtion for `recursive_mergesort` states that `A` should contain at least 1 element, that `L` and `R` should be within the `Range` of indices used by `A`, and `L` should be less than or equal to `R`. On lines 14–16, the postcondition states that after the procedure executes, the values stored in subarray `A(L .. R)` should be in ascending order, that values stored in indices outside the range `L .. R` should be the same as they are before execution of the procedure (aspect `Old` in a postcondition refers to the value of a subprogram parameter before execution), and that every value stored in `A` after execution of the procedure should correspond to some value stored in `A` before execution of the procedure. On lines 20–24, the precondition for `merge` states that `L` should be less than or equal to `R`, `M` should be between `L` and `R`, and that the left and right halves of the subarray to be merged should each be sorted. On lines 25–27, the postcondition states the same thing as it does for `recursive_mergesort`. This is not surprising since `recursive_mergesort` is using `merge` as a helper function to achieve its own postcondition. Briefly note that pre- and postconditions in SPARK must be free of potential run-time errors. This is why the short-circuit `and then` construct is used in the precondition of `merge`. If `L` or `M` were outside `A'Range`, then the expression `Is_Ascending(A(L..M))` or `Is_Ascending(A(M+1..R))` would raise a run-time error. The short-circuit `and then` construct ensures that these expressions are not evaluated if `L` or `M` is outside `A'Range`. The short-circuit construct is not necessary for the precondition of `recursive_mergesort`; it is used merely for efficiency reasons.

Note that these postconditions are at the gold level and not the platinum level. This is because they hold for the desired behavior but still allow for additional undesired behaviors. Specifically, both postconditions state that all values stored in `A` before execution of the subprogram should have a corresponding value stored in `A` after execution, but this does not guarantee that `A` is a one-to-one mapping/permu-

tation of `A'Old`. For instance, when `A'Old = [1 1 5 5 5 2 2]`, the desired value `A = [1 1 2 2 5 5 5]` satisfies the postcondition, but so does the undesired value `A = [1 2 5 5 5 5 5]`. Additional postconditions would be required to rule out such behaviors (i.e. to ensure that `A` is a permutation of `A'Old`). Section 3.3.3 discusses why proving this particular property is difficult for this particular algorithm.

3.3.2 Loop Invariants Needed

With no additional annotations, SPARK is able to prove that the body of the procedure `recursive_mergesort` satisfies the postcondition of `recursive_mergesort` based on the precondition of `recursive_mergesort`, the simple logic required to compute the midpoint index `M`, and the pre- and postconditions of internal calls to `recursive_mergesort` and `merge`. However, without additional annotations, SPARK is not able to prove the postcondition of `merge`. In particular, user-provided loop invariants inside the body of `merge` are needed. These create extra verification conditions (i.e. assertions that SPARK must prove), that can help guide proof of the postcondition. Subprograms containing loops almost always require loop invariants, which express properties that the user believes to be `True` at every iteration of the loop. These are needed because each execution of a loop can be viewed as a separate path through the subprogram, and the number of times the loop executes cannot be determined statically (i.e. there are potentially an infinite number of paths through a loop). SPARK analyzes loops by splitting them into three parts: the path that enters the loop for the first time and terminates on the invariant, the path the goes from the invariant and terminates on the invariant again, and the path that goes from the invariant and leaves the loop. By splitting the loop into parts, SPARK is able to prove loop invariants through induction. More specifically, SPARK proves that the loop invariant holds in the first iteration of the loop, then it proves the loop

invariant holds in an arbitrary iteration assuming it held in the previous iteration. The loop invariant and properties of the path out of the loop are then used to help prove subsequent properties, including subprogram postconditions. Note that a loop invariant in SPARK can appear anywhere at the top level of a loop (i.e. not nested inside another control structure), though it is usually placed at the beginning or end of the loop. It can also be broken across multiple `pragma Loop_Invariant` statements, though these must be grouped together without any other intervening statements or declarations. Internally, SPARK creates a loop invariant that is a conjunction of all loop invariant `pragma`s in the loop.

```
1  pragma Loop_Invariant(ii <= n1 and jj <= n2);
2  pragma Loop_Invariant(kk = ii + jj);
3  pragma Loop_Invariant(Is_Ascending(A(L..L+(kk-1))));
4  pragma Loop_Invariant(for all I in L..L+(kk-1) =>
5     ((if ii < n1 then A(I) <= L_temp(ii)) and (if jj < n2 then A(I) <= R_temp
          (jj))));
6  pragma Loop_Invariant(for all I in A'Range => (if I not in L..L+(kk-1) then
       A(I) = A'Loop_Entry(I)));
7  pragma Loop_Invariant(for all I in L..L+(kk-1) =>
8     (for some J in A'Range => A(I) = A'Loop_Entry(J)));
```

Figure 14: Loop invariant `pragma`s needed to prove the `merge` postcondition.

Figure 14 shows a loop invariant that enables SPARK to prove the postcondition of `merge`. This loop invariant is broken into six individual loop invariant `pragma`s that would be placed between lines 32 and 33 of the subprogram body shown in Figure 12. The first loop invariant `pragma` on line 1 specifies upper bounds `n1` and `n2` on variables `ii` and `jj`, which correspond to the sizes of `L_temp` and `R_temp`, respectively. The second loop invariant `pragma` on line 2 specifies that `kk = ii + jj`. Note that the size of `A(L..R)` is `n1` + `n2`. Also, recall from Figure 12 that `ii`, `jj`, and `kk` are used to index into `L_temp`, `R_temp`, and `A`, respectively, as the loop proceeds. These two loop invariant `pragma`s help prove several important properties of these variables. First, they help prove that they remain within the bounds of their type, `Natural`, and also

that they stay within the index range of their respective arrays. These checks are necessary to prove absence of run-time errors. Second, the fact that `kk = ii + jj` also helps prove that elements are placed in `A(L..R)` in order. Each iteration of the loop evaluates whether `L_temp(ii)` or `R_temp(jj)` is smaller, and the smaller value is stored in `A(L + kk)`. Then either `ii` or `jj` is incremented by 1, depending on which corresponded to the smaller value, and `kk` is always incremented by 1. So, each iteration of the loop places the next smallest value at the next location in `A`.

The third loop invariant `pragma` on line 3 states that all elements placed in `A` up to this point are sorted, (i.e. `Is_Ascending(A(L..L+kk-1))`), which is critical for proving the part of the `merge` postcondition that states `Is_Ascending(A(L..R))`. Note that it is necessary to subtract 1 from `kk` in this expression, since `kk` is incremented by 1 in anticipation of the next loop iteration right before this `pragma` is evaluated. Unfortunately, SPARK is unable to prove this part of the loop invariant without additional information. This information is provided by the fourth loop invariant `pragma`, which broken across lines 4 and 5. Conceptually, it states that all elements placed in `A` up to this point, (i.e. in the range `L..L+kk-1`), are less than the next values of `L_temp` and `R_temp` indexed by `ii` and `jj`. Note that since either `ii` or `jj` is incremented by 1 in anticipation of the next iteration of the loop right before this `pragma` is evaluated, whichever is incremented would be outside the index range of its corresponding array on the last iteration of the loop (i.e. `n1-1` for `L_temp` and `n2-1` for `R_temp`). This is why line 5 only evaluates whether `A(I) <= L_temp(ii)` if `ii < n1` and whether `A(I) <= R_temp(jj)` if `jj < n2`.

The fifth loop invariant `pragma` on line 6 conceptually states that all elements of `A` that have yet to be changed, (i.e. those outside of indices `L..L+kk-1`), have the same value as those in `A'Loop_Entry`, which refers to the value of `A` at the start of the loop. This helps prove the part of the `merge` postcondition that states `(for all`

`I in A'Range => (if I not in L .. R then A(I)= A'Old(I))).`

The sixth loop invariant `pragma` broken across lines 7 and 8 conceptually states that for all values that have been placed in `A` so far, (i.e. in indices `L..L+kk-1`), have a corresponding value somewhere in `A'Loop_Entry`. This helps prove the part of the `merge` postcondition that states `(for all I in A'Range => (for some J in A'Range => A(I)= A'Old(J)))`.

With these six loop invariant `pragma`s in place, SPARK is able to prove the postcondition on `merge`. As previously mentioned, SPARK is already able to prove the postcondition on `recursive_mergesort` without any additional annotations, so it is now able to prove the whole `mergesort_algorithm` package.

3.3.3 Discussion (Limitations to proof)

There are two additional issues to discuss with regards to the SPARK implementation of the recursive mergesort algorithm. The first issue is that the postcondition is only gold level. Specifically, the postcondition does not require that the value of `A` after `merge` executes is a one-to-one mapping/permutation of `A'Old`, only that every value stored in `A` has a corresponding value stored in `A'Old`. In theory, it would be possible to add the permutation property to the postcondition of `merge` and prove it, raising the level to platinum. There are in fact example SPARK implementations of various sorting algorithms in the spark-by-example[3] repository, such as an insertion sort and a selection sort algorithm, that include the permutation property in their postconditions. However, this property is easier to prove for those particular sorting algorithms, since they work by iteratively swapping elements of the array to be sorted in a loop. This makes it easy to write a loop invariant expressing that the permutation property holds in every iteration of the loop. For merge sort, a loop invariant

[3]https://github.com/tofgarion/spark-by-example/

that captures the permutation property is not as easy to write, since the algorithm incrementally adds new values to the array to be sorted, pulling from either the left or right half of the original array. Work is being done on using something like the multiset specifications in the spark-by-example[3] repository to model the array to be sorted and the left and right halves of the array as multisets to write a loop invariant that expresses that the number of occurrences of each element of `A(L..L+kk-1)` is the same as the number of occurrences of each element of `L_temp(0..ii-1)` unioned with `R_temp(0..jj-1)`. This should allow the permutation property to be proven for this merge sort algorithm.

The second issue has to do with subprogram termination. In general, SPARK does not try to verify termination of subprograms; rather, by default, it proves properties such as postconditions assuming that subprograms terminate. It is possible to add `pragma`s that instruct SPARK to attempt to prove subprogram termination. For example, the following `pragma`s can be added to the package specification for `mergesort_algorithm`:

```
pragma Annotate (GNATprove, Terminating, merge);
pragma Annotate (GNATprove, Terminating, recursive_mergesort);
```

This tells GNATprove, one of the main tools used by SPARK, to attempt to prove that `recursive_mergesort` and `merge` terminate. Subprograms like `merge` that include a loop often require loop variants (i.e. assertions about scalar variables modified by the loop that the user believes to be monotonically increasing or decreasing), to help prove that the loop terminates if they can be proven to be `True`. For SPARK to prove the first termination `pragma`, the following loop variant must be added before or after the loop invariant `pragma`s in the body of `merge`:

```
pragma Loop_Variant(Increases => ii + jj);
```

SPARK is able to prove the loop variant is `True`. Since the loop termination condition

is `ii < n1 and jj < n2`, which must be reached eventually if `ii + jj` is monotonically increasing, SPARK can then prove that the loop and `merge` terminates. Unfortunately, `recursive_mergesort` is a recursive procedure, which is harder to prove. SPARK is unable to prove termination for this procedure. In such a case, other forms of verification such as third party review are appropriate.

3.4 Verifying a Priority Queue Algorithm in SPARK

This section moves through the process of verifying SPARK code to the platinum level through an example, namely a minimum priority queue algorithm with insert and extract functionality.

3.4.1 Priority Queue Implementation

This section implements a minimum priority queue with two functionalities, `insert` and `extract`. Figure 15 shows a visual depiction of the implemented priority queue data structure with size 10. It is implemented in such a way that each element of the structure has a descriptor and corresponding priority. It is a priority queue represented as an array of pairs. Because this is a minimum priority queue implementation, the lower the assigned priority value, the higher the priority. For SPARK implementation, this array data structure must remain static in memory usage. It must remain a fixed size. In this case, that fixed size is 10 elements. In order to keep track of the current scope of the fixed length priority queue, there is a tracker variable declared and initialized in the priority specification file called `REAR`. `REAR` marks the last element in the current scope of the array data structure, and therefore, the last reachable and relevant pair within the priority queue. When a pair is inserted, `REAR` is moved up one spot, and when a pair is extracted, `REAR` is moved down one spot - more detail on this further on.

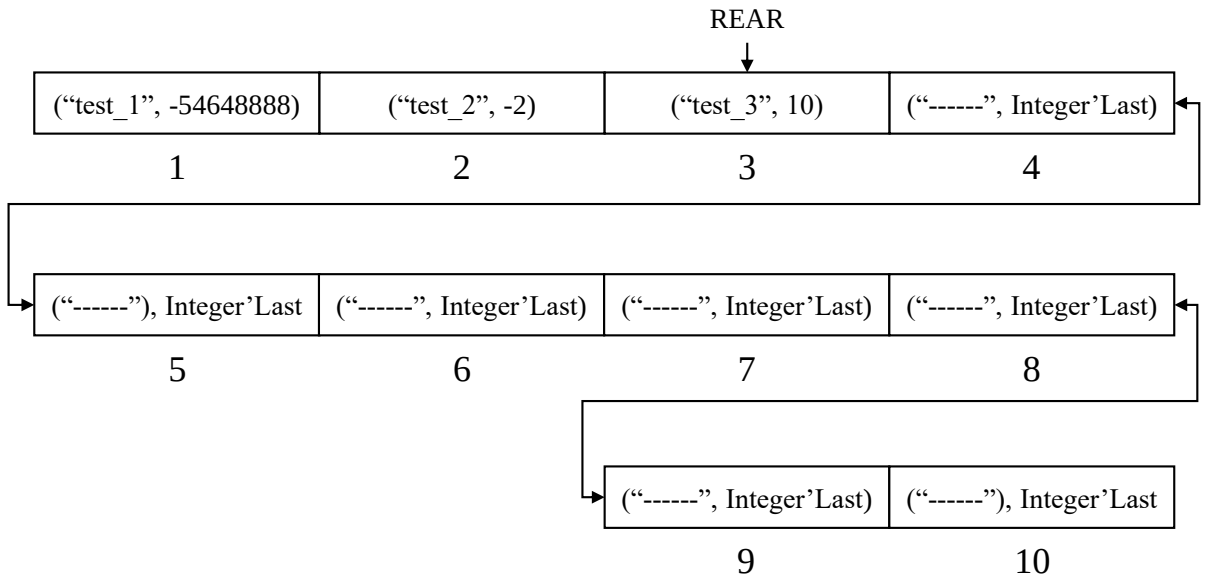


Figure 15: A visual depiction of the priority queue implementation with fixed size 10.

Briefly, the implementation of the driver (`main.adb`) is in Appendix C under Listing C.1. There are two helper subprograms, `insert` and `extract`, that operate on a priority queue. These `test_helper` subprograms are not part of the provable code and only act as helpers for `main.adb` cleanliness and execution as well as priority queue data structure initialization and can be seen in Appendix C under Listing C.4 and Listing C.5.

Consider a SPARK implementation of algorithms to insert and extract (Item, Priority) pairs. The types for this priority queue implementation are now defined in the `priority` package, shown in Figure 16. Line 1 includes the keywords `with SPARK_Mode` in the package specification to indicate that it should contain valid SPARK code. Line 3 defines the subtype `Item` as a subtype of standard type `String` that is constrained to a length of 6 characters. Line 5 defines the subtype

```
1  package priority with SPARK_Mode is
2
3     subtype Item is String(1..6);
4
5     subtype Given_Priority is Integer;
6
7     type Item_Priority_Pair is record
8        X: Item;
9        P: Given_Priority;
10    end record;
11
12    Initializer_Item : constant Item_Priority_Pair := (("------", Integer'
          Last));
13
14    type Priority_Queue is private;
15
16    procedure Print_Queue(PQ : in Priority_Queue);
17
18    procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair);
19
20    procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair
          );
21
22 private
23
24    subtype Index_Range is Natural range 1 .. 10;
25    subtype REAR_Index_Range is Natural range 0..Index_Range'Last;
26
27    type Item_Priority_Array is array(Index_Range) of Item_Priority_Pair;
28
29    type Priority_Queue is record
30       PQ: Item_Priority_Array;
31       REAR: REAR_Index_Range := 0;
32    end record;
33
34 end priority;
```

Figure 16: Basic SPARK package specification for various `priority` specific type definitions as well as specification of procedures `insert` and `extract`.

`Given_Priority` as a subtype of standard type `Integer`. `Given_Priority` represents the priority of each (Item, Priority) pair in the array from `Integer'First` to `Integer'Last`. Lines 7–10 define the type `Item_Priority_Pair` as a record of both `Item` and `Given_Priority`. `Item_Priority_Pair` is the type that will represent the (Item, Priority) pair that has been mentioned many times previously. Line 12 defines the constant `Initializer_Item` as a `Item_Priority_Pair` with generic values `(("------", Integer'Last))`. `Initializer_Item` acts as a placeholder for unfilled elements of the fixed array upon its initialization. In other words, if the user only

supplies the following three `Item_Priority_Pair`s,

```
1     (("test_1", -54648888))
2     (("test_2", -2))
3     (("test_3", 10))
```

then as previously seen in Figure 15, the other seven elements of the fixed size array representing the priority queue is initialized with `Initializer_Item`. Line 14 defines the type `Priority_Queue` as a `private` entity that is defined within the corresponding `private` field.

Before describing procedures `Print_Queue`, `insert`, and `extract`, this thesis discusses the private type `Priority_Queue` (lines 22-33). Line 24 defines the subtype `Index_Range` as a subtype of standard type `Natural` that is constrained to range from 1 to 10. `Index_Range` defines the range of elements in the fixed array of pairs. Line 25 defines the subtype `REAR_Index_Range` as a subtype of standard type `Natural` that is constrained to range from 0 to 10. `REAR_Index_Range` defines the range of that `REAR` can be equal to. The value 0 represents that the priority queue has no elements. Line 27 defines the type `Item_Priority_Array` as an array type constrained by type `Index_Range` that holds elements of type `Item_Priority_Pair`. `Item_Priority_Array` is the fixed length array of `Item_Priority_Pair`s that represents the data structure for the priority queue. Lines 29–32 now define the previously mentioned `Priority_Queue` type. `Priority_Queue` is a record that encapsulates the current priority queue (`Item_Priority_Array`) and its corresponding `REAR` value into one concise record. This queue/REAR value pair can now be passed around as one easy to reference unit. For example, if a `Priority_Queue` called `PQ` is defined, then to reference the priority queue the `PQ` field within the record (line 30) would be referenced. This would look like `PQ.PQ`. To reference the `REAR` value, one would reference the `REAR` field within the record (line 31). This would look like `PQ.REAR`.

This thesis now goes back to the procedures `Print_Queue`, `insert`, and `extract`

on lines 16, 18, and 20, respectively. The basic specification at the bronze level for procedures in package `priority` with no contracts is shown in Figure 16. On line 16, the procedure `Print_Queue` is defined. This procedure takes in the current priority queue and current REAR via the `Priority_Queue` record. The `Print_Queue` procedure is implemented in the `priority` body file, Figure 17, along with the following two procedures `insert` and `extract`.

Line 18 defines the specification for the procedure `insert`. Its first formal parameter `PQ` is type `Priority_Queue`. `PQ` is of mode `in out`, meaning that the parameter value must be initialized before the procedure is called, and then be modified by the procedure. It is important to note that with the stricter subset SPARK, `in out` can only be used for a procedure, not a function. This is because formal verification is simpler if functions can be assumed to be free of side effects (i.e. if they do not modify the values of their parameters). Its second parameter `pair` is of type `Item_Priority_Pair` and implicitly has mode `in`, meaning that the parameter should be initialized before being passed into the procedure and cannot be modified by the procedure. To recap, if there is no `in out` or `out` descriptor present on the parameter, it is implied to be treated as an `in` mode parameter. The goal of `insert` is to insert the provided `Item_Priority_Pair` called `pair` to the provided `Priority_Queue` named `PQ`. The `insert` procedure then updates the `Priority_Queue` record with both the updated `REAR` and the updated `Item_Priority_Array` that is holding the priority queue elements.

Line 20 defines the specification for the procedure `extract`. Its first formal parameter `PQ`, just like in the `insert` declaration, is of type `Priority_Queue`. Its second parameter `pair`, just like in `insert`, is of type `Item_Priority_Pair`, but this time has mode `out`, indicating that any input value will not be used, and an output value is guaranteed. This `out` parameter conceptually acts as a return value that can be

referenced in the body files where `extract` is used. The goal of `extract` is to extract the highest priority (lowest `Given_Priority` value) `Item_Priority_Pair` from the provided priority queue. The `extract` procedure then updates the `Priority_Queue` record with the updated queue and `REAR` value and output the extracted pair to the `out` parameter.

Now consider the basic body of package `priority`, shown in Figure 17. The function `Print_Queue` prints the provided `Priority_Queue`. The procedure `insert` places the provided `Item_Priority_Pair pair` in the index directly following the original queue's `REAR`. The new `Item_Priority_Pair` has been added into the priority queue (line 22). At this point, `insert` updates the `Priority_Queue` record fields with the current priority queue and `REAR` values.

The procedure `extract` is slightly more complicated. `extract` looks to find the minimum `Given_Priority` of all of the `Item_Priority_Pairs` in the provided `Priority_Queue` priority queue in the loop on lines 32-36. This loop updates the `min_index` if it finds an `Item_Priority_Pair` with a lower `Given_Priority`. This `min_index` can be used later in the procedure to reference the minimum pair, which is extracted. Next, in lines 38-39, `extract` removes the identified `Item_Priority_Pair`. Line 38 in `extract` takes everything from the beginning of the priority queue array, `PQ.PQ`, to one before `min_index` and places it in `new_array` covering the same indexes. Line 39 then takes everything in `PQ.PQ` from one above `min_index` to `REAR` and places the priority queue elements in `new_array` from `min_index` to one less than the original `REAR`. This `new_array` essentially is the same as the original `Priority_Queue` record, `PQ`, without the `Item_Priority_Pair` at `min_index`. That pair is removed, and all items to the right of that extracted minimum `Item_Priority_Pair` are moved to the left by one element. This means that `REAR` also moves to the left by one element. Now the new `REAR` needs to move to the left by one element. On lines 41–42,

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body priority with SPARK_Mode is
4
5     procedure Print_Queue(PQ : in Priority_Queue) is
6     begin
7        if Length(PQ) = 0 then
8           Put_Line("EMPTY_Queue");
9           Put_Line("");
10       else
11          for I in 1..PQ.REAR loop
12             Put("Element " & I'Img & ": ");
13             Put("(" & (PQ.PQ(I).X) & ", ");
14             Put_Line(Integer'Image(PQ.PQ(I).P) & ") ");
15          end loop;
16          Put_Line("");
17       end if;
18    end Print_Queue;
19
20    procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair) is
21    begin
22       PQ.PQ(PQ.REAR + 1) := pair;
23       PQ.REAR := PQ.REAR + 1;
24    end insert;
25
26    procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair
          ) is
27       new_priority_queue : Priority_Queue;
28       new_array : Item_Priority_Array := PQ.PQ;
29       min_index : Index_Range := 1;
30       orig_queue : constant Priority_Queue := PQ;
31    begin
32       for I in 1..PQ.REAR loop
33          if PQ.PQ(I).P < PQ.PQ(min_index).P then
34             min_index := I;
35          end if;
36       end loop;
37
38       new_array(1..min_index-1) := PQ.PQ(1..min_index-1);
39       new_array(min_index..PQ.REAR-1) := PQ.PQ(min_index+1..PQ.REAR);
40
41       new_priority_queue.PQ := new_array;
42       new_priority_queue.REAR := PQ.REAR - 1;
43
44       pair := PQ.PQ(min_index);
45       PQ := new_priority_queue;
46
47    end extract;
48 end priority;
```

Figure 17: Basic SPARK package body for procedure `insert` and procedure `extract`.

`extract` populates the two fields (`PQ` and `REAR` as defined in Figure 16). On line 44, the `out` parameter `pair` is given the value of the minimum priority queue pair found. On line 45, the provided `Priority_Queue` is updated with the new values.

Now consider how to formally specify key behaviors of these two functions using

pre- and postconditions in SPARK. The updated version of Figure 16 is one file but is split into Figure 18 and Figure 19 so that it can fit within this section. For convenience, this thesis first adds seven functions to the specification for package `priority` that return `True` if and only if desired behaviors are met. These seven functions (in Figure 18) can usually be entirely defined in the specification file using what is known in SPARK as an expression function as done in Section 3.3.1. The

```
1  package priority with SPARK_Mode is
2
3     subtype Item is String(1..6);
4
5     subtype Given_Priority is Integer;
6
7     type Item_Priority_Pair is record
8        X: Item;
9        P: Given_Priority;
10    end record;
11
12    Initializer_Item : constant Item_Priority_Pair := (("------", Integer'
          Last));
13
14    type Priority_Queue is private;
15
16    procedure Print_Queue(PQ : in Priority_Queue);
17
18    function Is_Not_Full(PQ: Priority_Queue) return Boolean
19       with Ghost;
20
21    function Is_Not_Empty(PQ: Priority_Queue) return Boolean
22       with Ghost;
23
24    function Is_Min(Orig_Queue: Priority_Queue; Min_Priority_Found:
          Given_Priority) return Boolean
25       with Ghost;
26
27    function Is_At_End_Of_Queue(Orig_Queue, Result_Queue: Priority_Queue;
          inserted_pair: Item_Priority_Pair) return Boolean
28       with Ghost,
29       Pre => Is_Not_Empty(Result_Queue);
30
31    function Is_First_Extracted(Orig_Queue, Result_Queue: Priority_Queue;
          extracted_pair: Item_Priority_Pair) return Boolean
32       with Ghost;
33
34    function Did_Queue_Increase(Orig_Queue, Result_Queue: Priority_Queue)
          return Boolean
35       with Ghost;
36
37    function Did_Queue_Decrease(Orig_Queue, Result_Queue: Priority_Queue)
          return Boolean
38       with Ghost;
```

Figure 18: PART 1: SPARK package specification of ghost functions.

```
39
40     procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair)
           with
41        Pre =>
42           Is_Not_Full(PQ),
43        Post =>
44           Did_Queue_Increase(PQ'Old, PQ)
45           and Is_At_End_Of_Queue(PQ'Old, PQ, pair);
46
47     procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair
           ) with
48        Pre =>
49           Is_Not_Empty(PQ),
50        Post =>
51           Did_Queue_Decrease(PQ'Old, PQ)
52           and Is_Min(PQ'Old, pair.P)
53           and Is_First_Extracted(PQ'Old, PQ, pair);
54
55     pragma Annotate (GNATprove, Terminating, insert);
56     pragma Annotate (GNATprove, Terminating, extract);
57
58  private
59
60     subtype Index_Range is Natural range 1 .. 10;
61     subtype REAR_Index_Range is Natural range 0..Index_Range'Last;
62
63     type Item_Priority_Array is array(Index_Range) of Item_Priority_Pair;
64
65     type Priority_Queue is record
66        PQ: Item_Priority_Array;
67        REAR: REAR_Index_Range := 0;
68     end record;
69
70  end priority;
```

Figure 19: PART 2: SPARK package specification for procedure `insert`, procedure `extract` with behavioral contracts expressed using ghost functions.

issue here is that `Priority_Queue` is a private entity. This means its private fields cannot be referenced in the public portion of the specification file. To reconcile this, the expression function is implemented in the body file of the `priority` package where the private fields can be referenced freely (body implementatinos can be seen in Appendix C Listing C.3), rather than the public portion of the specification file. Their specification file definitions are seen in Figure 18, lines 18–38.

These seven functions (in Figure 18) are marked with aspect `Ghost` to indicate that they are used mainly for proof purposes. That is, the seven ghost functions are only compiled and included in an executable if the user indicates to the compiler that

it should be (e.g. to check the code through test rather than verify the code through proof). On line 18, `Is_Not_Full` returns `True` if and only if the queue is not full. That is, it does not have a `REAR` value equal to the size of `REAR_Index_Range'Last`. This is checked with the expression: `PQ.REAR < REAR_Index_Range'Last`. On line 21, `Is_Not_Empty` does the exact opposite and returns `True` if the `REAR` value of the queue is not the lowest possible value. This is accomplished with the following expression: `PQ.REAR > REAR_Index_Range'First`. On line 24 `Is_Min` checks that the priority of the extracted `Item_Priority_Pair` is indeed the minimum priority within the original priority queue, `Orig_Queue`, provided. It compares the `Min_Priority_Found` by `extract` to the `Given_Priority` of each `Item_Priority_Pair` from 1 to the original `REAR`. This is done with the expression:

```
(for all I in 1 .. Orig_Queue.REAR => Orig_Queue.PQ(I).P >=
   Min_Priority_Found)
```

On line 27, `Is_At_End_Of_Queue` returns `True` if the pair inserted is inserted at the end of the resulting queue and if the rest of the original queue up to the point of the inserted pair is preserved. This is accomplished by the following expression:

```
((Result_Queue.PQ(Result_Queue.REAR) = inserted_pair) and (for all I
    in 1 .. Orig_Queue.REAR => Orig_Queue.PQ(I) = Result_Queue.PQ(I)
   ))
```

It is important to note that `Is_At_End_Of_Queue` requires a precondition unlike the rest of the ghost functions. The precondition utilizes an earlier ghost function and is as follows: `Pre => Is_Not_Empty(Result_Queue);` This is needed because without it, SPARK notices that the `REAR` value within the provided `Priority_Queue` record is of type `REAR_Index_Range`. Remember from Figure 16 and its updated version Figure 19 that `REAR_Index_Range` has a range of `0..Index_Range'Last`. This means that `REAR` could technically be at index 0, which is not in the range of `Index_Range` of which `Priority_Queue`'s `Item_Pair_Array` is constrained by. This means that

the expression within `Is_At_End_Of_Queue` could refer to an index out of range. Specifying that `REAR` cannot be 0 as a precondition with `Is_Not_Empty` means that `REAR` must be within `1..Index_Range'Last`, which is the legal range for the priority queue field (`PQ`) references made in the `Is_At_End_Of_Queue` expression.

On line 31, `Is_First_Extracted` returns `True` if a few conditions are met. The first condition being that there is a value within the original queue that matches the extracted pair. This is expressed in the following way:

```
(for some I in 1 .. Orig_Queue.REAR => extracted_pair = orig_queue.
   PQ(I))
```

The second condition is that all `Item_Priority_Pair`s located in an index lower than that point should have a `Given_Priority` larger than the `Given_Priority` of the extracted pair. This ensures that if there were `Item_Priority_Pair`s with the same `Given_Priority`, the first one is extracted. This is the way that the loop in the `extract` procedure operates. Recall, the loop does not update the `min_index` value unless a pair's `Given_Priority` is less than the current `min_index`, meaning the first element of the queue with that `Given_Priority` is extracted. This is expressed in the following way:

```
(for all X in 1 .. I-1 => (Result_Queue.PQ(X).P > extracted_pair.P))
```

where `I` is the index of the extracted pair in the original queue. The third condition is the preservation property. All elements/pairs from the beginning of the queue up to one below the extracted pair in the original queue should remain the same between the original queue and resulting queue. And all elements in the index directly after the extracted pair in the original queue should equal that of the elements in the resulting queue from the index of the extracted pair (as defined by the original queue) to the `REAR` of the resulting queue, which is one lower than the `REAR` of the original queue from with the pair is extracted. This is expressed with the following expression:

```
(Orig_Queue.PQ(1..I-1) = Result_Queue.PQ(1..I-1)) and then (
    Orig_Queue.PQ(I+1..Orig_Queue.REAR) = Result_Queue.PQ(I..
    Result_Queue.REAR))
```

where `I` is the index of the extracted pair in the original queue once more. Those three conditions are checked in that order and must all evaluate to `True` for `Is_First_Extracted` to evaluate to `True`. The body of this ghost function can be seen in Figure 21.

Finally, lines 34 and 37, still located in Figure 18, specify the two ghost functions, `Did_Queue_Increase` and `Did_Queue_Decrease`, respectively. They verify that the value of `REAR` either went up by one or down by one. `Did_Queue_Increase` evaluates to `True` if: `(Result_Queue.REAR = Orig_Queue.REAR + 1)`. `Did_Queue_Decrease` evaluates to `True` if: `(Result_Queue.REAR = Orig_Queue.REAR - 1)`. Recall, all of these seven ghost functions can be seen implemented in the body file of the `priority` package in Appendix C under Listing C.3.

It is now possible to write pre- and postconditions for procedures `insert` and `extract` using the seven ghost functions previous defined in Figure 18. Recall, Figure 18 and Figure 19 show the modified specification for package `priority`. The pre- and postconditions for both procedures `insert` and `extract` are seen in Figure 19. Recall, for a pre- and/or postcondition to pass, it must evaluate to `True`. This thesis starts with procedure `insert`'s contracts. In Figure 19, line 42 is the precondition `Is_Not_Full`. This precondition means that before this procedure can execute, it must be `True` that the the parameter `Priority_Queue` contains a queue that is not full. This is because if the queue is already full, then there would be no way to add a new pair to the queue. Line 44 is the postcondition `Did_Queue_Increase`. Recall, this checks that the `REAR` field within record `Priority_Queue` is indeed one higher than the `REAR` in the original `Priority_Queue` that can be referenced with the `'Old` attribute. The original queue (`PQ'Old`) and current queue (`PQ`) are supplied and their

REAR values are compared. Line 45 is the postcondition Is_At_End_Of_Queue, which checks that the parameter pair is inserted at the end of the new queue and it also checks for the preservation of the original queue within in the resulting queue. All pairs up to the inserted pair, which sits at the end of the new queue, should be equal.

Now move to extract's contracts (still in Figure 19). Line 49 is the precondition Is_Not_Empty. This precondition means that before this procedure can execute, it must be True that the the parameter Priority_Queue contains a queue that is not empty. This is because if the queue is already empty, there would be no way to extract a new pair from the queue. Line 51 is the postcondition Did_Queue_Decrease. This line checks that the REAR value decreased by one after execution of the extract procedure. Line 52 is the postcondition Is_Min, which ensures that the Item_Priority_Pair with the minimum Given_Priority is indeed the lowest Given_Priority value within the original queue (PQ'Old). In other words, there should be no Item_Priority_Pair within PQ'Old with a lower value than that of pair.P. Line 53 is the postcondition Is_First_Extracted, which ensures that if there were multiple Item_Priority_Pairs with the same minimum Given_Priority , the first is the one that is extracted from the original queue. It also checks for the preservation of the original queue within the resulting queue minus the extracted pair. Lines 55 and 56 will be discussed in Section 3.4.3.

3.4.2 Loop Invariants and Assertions Needed

As with merge sort, with no additional annotations, SPARK is able to prove that the body of insert satisfies the postcondition of insert based on the precondition of insert and the simple logic required to insert the new Item_Priority_Pair. However, without additional annotations, SPARK is not able to prove the postcondition of extract. In particular, a user-provided loop invariant inside the body of extract

is needed.

```
1  for I in 1..REAR loop
2     if PQ(I).P < PQ(min_index).P then
3        min_index := I;
4     end if;
5     pragma Loop_Invariant(min_index in 1 .. I);
6     pragma Loop_Invariant(for all E in 1 .. I => PQ.PQ(E).P >= PQ.PQ(
          min_index).P);
7     pragma Loop_Invariant(for all E in 1 .. min_index-1 => PQ.PQ(E).P > PQ.PQ
          (min_index).P);
8  end loop;
```

Figure 20: Loop invariant `pragma`s needed to prove the `extract` postcondition.

To enable SPARK to prove the postcondition of `extract`, a loop invariant is needed in the loop of `extract`'s body. This loop invariant would be placed between lines 35 and 36 of the basic subprogram body shown in Figure 17 just before the end of the loop. Figure 20 shows the loop invariant split among three loop invariant `pragma`s that enable SPARK to prove the postcondition of `extract` positioned within the loop that finds the minimum `Item_Priority_Pair` based on its `Given_Priority` . The first loop invariant `pragma` proves that `min_index` is in the current scope by ensuring that `min_index` is not outside of `1..I`. Because SPARK proves loop invariants through induction, SPARK must be provided with the information needed for an inductive proof. The second loop invariant `pragma` is useful to prove the `Is_Min` postcondition criteria. It essentially tells the prover, which only reasons about the current iteration, that up to this point `min_index` represents the element corresponding to the `Item_Priority_Pair` with the minimum `Priority_Given` value. This is accomplished by proving that every member in the priority queue up to the current loop iteration, `I`, is indeed greater than or equal to the element that is currently deemed the minimum. Note the if-condition and its minimum value finding procedure is already complete by the time the loop invariant `pragma` is evaluated. This allows the current iteration's loop invariant to be evaluated with `min_index` updated accord-

ingly. Because of the updated `min_index`, each iteration of the loop up to the current `REAR` ends with `PQ(min_index).P` representing the minimum `Given_Priority` and therefore the minimum pair to extract from the supplied priority queue. The third loop invariant `pragma` is useful for `Is_First_Extracted`. It is very similar to the second loop invariant, but instead checks for the condition in `Is_First_Extracted` where every `Given_Priority` prior to the current `min_index` is greater than that of the current `PQ.PQ(min_index).P`. This is because in the postcondition that this loop invariant `pragma` is trying to prove, it wants to ensure that the first occurrence of an `Item_Priority_Pair` with the found minimum `Given_Priority` is indeed the one extracted. The third loop invariant fails if `min_index` is assigned to the hypothetical second, or third occurrence of the shared minimum `Given_Priority`.

These `Loop_Invariant pragam`s alone are not enough to help SPARK prove the postcondition of `extract`. Specifically, `Is_First_Extracted` in the `extract` postcondition is still not successfully proving. To help the SPARK provers with this postcondition, it may be helpful to provide more specific information as an assertion before exiting the `extract` procedure itself. First, recall the expression to be proven. Figure 21 represents the expression that SPARK cannot prove with only the three loop invariants.

```
1  function Is_First_Extracted(Orig_Queue, Result_Queue: Priority_Queue;
       extracted_pair: Item_Priority_Pair) return Boolean is
2      (for some I in 1 .. Orig_Queue.REAR =>
3         extracted_pair = orig_queue.PQ(I)
4      and then (for all X in 1 .. I-1 => (Result_Queue.PQ(X).P > extracted_pair
           .P))
5      and then (Orig_Queue.PQ(1..I-1) = Result_Queue.PQ(1..I-1))
6      and then (Orig_Queue.PQ(I+1..Orig_Queue.REAR) = Result_Queue.PQ(I..
           Result_Queue.REAR)));
```

Figure 21: Ghost function `Is_First_Extracted` body expression function implementation.

To help the provers with `Is_First_Extracted` in the postcondition the two as-

sertions in Figure 22 are added at the very end of the `extract` procedure. They are a copy from the body of `Is_First_Extracted` expressed in two different ways. The first one using the `min_index` found earlier within the scope of the `extract` procedure and the second by using the same expression that is found within `Is_First_Extracted`.

```
1  pragma Assert(pair = orig_queue.PQ(min_index)
2     and then (for all X in 1 .. min_index-1 => (PQ.PQ(X).P > orig_queue.PQ(
          min_index).P))
3     and then (orig_queue.PQ(1..min_index-1) = PQ.PQ(1..min_index-1))
4     and then (orig_queue.PQ(min_index+1..orig_queue.REAR) = PQ.PQ(min_index..
          PQ.REAR)));
5
6  pragma Assert(for some I in 1 .. orig_queue.REAR =>
7     pair = orig_queue.PQ(I)
8     and then (for all X in 1 .. I-1 => (PQ.PQ(X).P > orig_queue.PQ(I).P))
9     and then (orig_queue.PQ(1..I-1) = PQ.PQ(1..I-1))
10    and then (orig_queue.PQ(I+1..orig_queue.REAR) = PQ.PQ(I..PQ.REAR)));
```

Figure 22: Assert `pragma`s needed to prove the `extract` postcondition.

These three loop invariants (Figure 21) and two assertions (Figure 22) are enough for SPARK to reason that `Is_Min` and `Is_First_Extracted` are `True` and therefore allows the postcondition of `extract` to successfully prove. To summarize Figure 22, the prover cannot recognize that the `for some I` statement is `True` without knowing exactly which value of `I` makes the statement `True`. So the exact value is asserted in a nearly identical expression giving the proves the information they need to prove the `Is_First_Extracted` postcondition criteria (Figure 21) on the `extract` procedure. As previously mentioned, SPARK is already able to prove the postcondition on `insert` without any additional annotations, so it is now able to prove the whole `priority` package.

3.4.3 Discussion (Limitations to proof)

Like with the previous merge sort implementation, there is an additional point to discuss with regards to the SPARK implementation of this section's minimum priority

queue algorithm. Again, the issue has to do with subprogram termination. To help with this, the following `pragma`s can be added, as seen on lines 61–62 of Figure 18, to our package specification for `priority`:

```
pragma Annotate (GNATprove, Terminating, insert);
pragma Annotate (GNATprove, Terminating, extract);
```

This tells GNATprove to attempt to prove that `insert` and `extract` terminate. Subprograms like `extract` that include a loop often require loop variants, (i.e. assertions about scalar variables modified by the loop that the user believes to be monotonically increasing or decreasing), to help prove that the loop terminates if they can be proven to be `True`. Unlike that of the merge sort implementation, SPARK is able to prove that both `insert` and `extract` prove without any additional loop variants.

3.5 Summary

This chapter utilized the SPARK formal methods toolset to prove three SPARK implementations at or above the gold level of proof.

IV. Results and Analysis

4.1 Preamble

This research has resulted in three additional algorithms added to universal library of proven algorithms in SPARK. Additionally, this research has demonstrated the process of utilizing an auto-active prover, in this case SPARK, as a means to the formal verification of software. The educational aspect of showcasing all relevant source code cannot be overstated. This research serves as a starting point for many prospective developers that are looking to formally verify algorithms and/or software projects. Because the algorithms themselves and how they are implemented are the results, this section will discuss time and effort information and a proof analysis on levels of proof achieved along with what prevented a higher level proof.

The GNATprove output can be seen for each algorithm in the Appendix (Appendix D, Appendix E, and Appendix F). Code highlighted in the color blue indicates the SPARK's GNATprove has proven something about that line. The line by line breakdown shows the check proven. Section 4.2 will discuss the time-sinks and effort involved in this research. And finally, Section 4.3 will discuss the level of proof achieved, as defined in Section 2.7.6, and will discuss what prevented a higher level of proof from being achieved.

4.2 Time and Effort

Upon following the steps in Section 3.1, most time went into steps four and five, which is thinking through the specification, and then creating the pre- and postconditions as well as the loop invariants to help push the prover towards an inductive proof. This time sink is true for those subprograms with internal loops. This time sink is especially true in the case of the merge sort formalization. This algorithm has

recursion and loops found within the `merge` body. The biggest issue centered itself around the loop invanriants needed to guide SPARK's GNATprove towards a proof of the relevant postconditions. The provers have to reason and make sense of numerous changing variables, two temporary arrays, the merged array, and index tracking. This is a large task for a prover and as shown in Section 3.3.2 requires manual manipulation of the code in the form of loop invariants. In order for SPARK to be able to reason about this complex problem, the user must reason about it first and find relationships between all the moving parts. This is a difficult task for someone new to formal methods and only gets easier through example, practice, and time. There comes the point where one starts to understand how to communicate with the provers. The loop invariant construction is the most difficult portion across all three algorithms, although slightly easier in the case of interpolation and priority queue. This is because there are fewer variables to account for in those two algorithm's loops. In each case, it would take some trial and error, depending on the user's SPARK experience, to create the correct loop invariant. One of the most critical things to be communicated to the provers is that the loop is indeed moving towards the termination condition. A generic formula must be presented in such a way that implies forward progress. Upon that realization, it is much easier to create effective loop invariants that lead to successfully proven subprogram postconditions.

It is also important to note that sometimes the SPARK and GNATprove simply needed more time. GNATprove has a timeout value associated with its attempt at a proof. There are occasions in which a proof would fail until the provers timeout is bumped up, upon which it would successfully prove. One example of this is the `Is_Min` postcondition of the `insert` subprogram within the priority queue implementation. This postcondition would fail on the lowest timeout setting, and when given more time, this postcondition would successfully prove. Additionally, there are instances

in which utilizing the test driver (main function) with relevant sample data could help identify areas that the prover is having issues with. Specifically, as discussed in Section 2.7.7.1, this driver should be run with assertions enabled. This is accomplished by adding the `-gnata` flag to the command line and can also be enabled from within the GPS IDE in "Edit" then "Project Properties" then "Build" then "Ada" then "Debugging" and finally "Enable Assertions." With assertions enabled, the execution of the algorithm will halt when an assertion, loop invariant, pre- and/or postcondition, etc fails. This may, in turn, demonstrate to the user that the code is not implemented correctly.

Overall, the greatest effort and time-sink present in this research is indeed the planning of the formal specification and the process of supplying SPARK with the needed manual annotations for a successful auto-active proof.

4.3 Proof Analysis

Table 1 breaks down the levels of proof achieved as well as what is missing for the three algorithms this research worked with. The priority queue implementation,

Table 1: Level of Proof Achieved

Algorithm	**Level of Proof Achieved**	**Keys to Higher Proof**
Interpolation	Gold	Exact result not checked
Merge Sort	Gold	Termination assumptions & permutation property
Priority Queue	Platinum	N/A

which has a `insert` and `extract` functionality built-in is proven to full functional correctness (platinum proof). This level of proof essentially means that SPARK is able to prove that the `insert` procedure actually inserted the pair by checking if the pair supplied to the `insert` procedure is found in the resulting priority queue and that it is found at the end of the queue. Similarly, SPARK is able to prove that

the `extract` procedure extracted not only the minimum value pair but also the first minimum pair present in the priority queue based on the particular comparable value (`Given_Priority`) of the pair. SPARK is also able to prove the preservation of the rest of the queue in both `insert` and `extract`.

Interpolation and merge sort, on the other hand, are both gold level proofs. The similarity between the two is that key functional properties are proven, hence the gold level proof. But they both fall short of a platinum level proof for different reasons. First, the interpolation algorithm is fully proven other than one important exception. The exact interpolated result is technically not checked in the postcondition. Instead, the result is checked to be in the range of the two surrounding `Point`'s Y-values. Because the result of the interpolation function is just checked to be in the proper range rather than the actual interpolated value, this lowers the proof from a platinum proof to a gold one. And finally, merge sort is neglecting to prove the permutation property and termination property as discussed in Section 3.3.3. Again, lowering the merge sort level of proof from platinum to gold.

4.4 Summary

Overall, each of the three algorithms are proven to at least the gold level (i.e. that key functional correctness is proven in all three algorithms).

V. Conclusions

5.1 General Conclusions

This section is based on a previously published paper [70].[1] This research has given some impression of what is required to prove functional correctness of a subprogram, i.e. that a subprogram satisfies user-generated specifications in the SPARK language and auto-active verification toolset by going through the process of implementing and verifying an interpolation, a recursive merge sort, and a priority queue algorithm. In addition to writing the subprograms comprising this algorithm, this required formally expressing their desired behavior as part of the subprogram specifications and adding loop invariants and loop variants to the subprogram bodies.

Formal methods can be used to analyze a variety of properties of software and other design artifacts, such as worst-case execution time, max stack usage, and absence of run-time errors in software and security properties of software and architectures, as discussed in Section 1.2. The use of formal methods to prove functional correctness of software seems less common than simpler properties such as absence of run-time errors, though functional correctness of software has been a focus in the past [72]. However, as systems continue to become more complex, verifying functional correctness of software to a sufficient level of assurance through traditional means such as testing and third-party reviews may become increasingly less tractable and more expensive. This is especially a concern for complex systems that consist of many interacting modules or implement complex decision-making logic. As an example, this research has used SPARK to find errors in synthesized software implementations of protocols for various systems, including teams of unmanned air vehicles, that went

[1] Ryan Baity, Laura R Humphrey, Kenneth Hopkinson. Formal verification of a merge sort algorithm in SPARK. In *AIAA Scitech 2021 Forum*, page 0039, 2021. DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. Case #88ABW-2020-3580.

undetected through casual simulation and testing [73].

5.2 Significance of Research

This research has ultimately led to the publication of a verified merge sort algorithm via the SPARK auto-active verification toolset (Section 3.3) [70]. It has also presented the formal verification of two additional ready to publish algorithm implementations, interpolation and priority queue (Section 3.2 and Section 3.4, respectively). Overall, this research has added three algorithmic implementations to the body of proven SPARK algorithms.

This research also validates the viability of the SPARK toolset as a useful tool to the research's sponsor at the Air Force Research Laboratory (AFRL). It accomplished this by demonstrating the SPARK formal verification process on relatively easy to grasp algorithms for further utilization by the research sponsor and thereon, the U.S. Air Force. Ultimately, an effective process for proving that software is functionally correct allows for the confidence that any algorithm or software deployed onto operational U.S. Air Force units will work, from a software perspective, precisely as intended and allows for the continuation of mission execution. That is precisely what formal methods tools can provide.

5.3 Limitations

Although SPARK has been shown to be a worthwhile tool for the formal verification of algorithms, it of course does have its limitations. The first and most obvious limitations manifest themselves in the limitations mentioned in Section 2.7.2. These limitations may have an effect on specific projects that may rely on some of SPARK's excluded language features. Although this is true, the SPARK User's Guide mentions that future releases of SPARK may relax some of those restrictions [68]. With that

noted, the limitations are still currently present, and that might mean the implementer must rewrite vast portions of an algorithm to be free of any of the SPARK limitations. This may not be possible to avoid in all scenarios. For example, if multiple access objects are required read-write access to allocated memory, there will be troubles rethinking the original scheme of that algorithm.

5.4 Future Work

The future possibilities are endless with formal methods. When it relates to this particular research involving three distinct algorithms, there are a few paths that can be followed. One of the simplest and most useful would be to implement even more algorithms in SPARK, or via any other formal methods toolset, and prove their functional correctness. In other words, continue to add to the body of proven algorithms. There will always be value in continuing to formally verify algorithms that can then be used with total confidence by other developers. All of the algorithms that have been proven functionally correct can then be placed in an open repository, such as the spark-by-example[2] repository.

An additional research path could be to compare formal verification tools side-by-side. Some of the available tools that can be looked at in depth are mentioned in Section 2.6. Rather than do what this research did, which utilized one formal methods tool (SPARK) on three different algorithms, run three tools on one algorithms, for example.

Another research path could be to look into generalizing the implementations. This would mean to create a generic implementation of the algorithm in question and then prove said algorithm. For this to work, it would be necessary to change many points of the proof. One example is in the merge sort proof; the `Is_Accending`

[2]https://github.com/tofgarion/spark-by-example/

function would have to be re-written to account for generic inputs. This would most certainly be a difficult and worthwhile problem to investigate.

A final research path would be to look at integrating formalized code into a larger software system. Because this work serves as a stepping-stone for more in depth works to come, it only makes sense to take the lessons learned and the algorithms proven from this research and expand its utility into larger projects. Along the lines of this research, it would also be interesting to look into the modularization of an existing large scale project, take critical sections of the code, translate them from their native language into Ada, then SPARK, then utilize formal methods to verify them, and then interface them back into their original position within their respective software system, and see how well they work. This could be an important piece of research because it would demonstrate that only the most critical portions of large software systems, such as an individual subprogram, can be formalized and have no negative effect on the rest of the system when reintegrated.

5.5 Overall Conclusions

This section is based on a previously published paper [70].[3] Not only will software become more difficult to verify, low level requirements themselves will become more complex, and it will become more difficult to ensure the low level requirements are correct through traditional means such as modeling and simulation and testing. For example, it will be challenging to verify that interactions between services in service-oriented architectures result in desired system-level behavior. Others have made progress on that front [74, 75], and there are attempts to make progress in that area by using SPARK to formalize low-level requirements and verify compliance of software

[3]Ryan Baity, Laura R Humphrey, Kenneth Hopkinson. Formal verification of a merge sort algorithm in spark. In *AIAA Scitech 2021 Forum*, page 0039, 2021. DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. Case #88ABW-2020-3580.

services in a software framework for intelligent control of teams of unmanned vehicles [76]. It will also be challenging to prove properties about the algorithms that underlie increasingly complex and/or autonomous systems. NASA has made strides in this direction by formally proving properties of DAIDALUS (Detect and Avoid Alerting Logic for Unmanned Systems), a reference implementation of a detect and avoid concept intended to support the integration of Unmanned Aircraft Systems into civil airspace [77]. Formal methods have also been used to prove mathematical conjectures that have defied manual proof, such as Keller's conjecture [78], and research has been done to use formal methods to find a subtle error in the "paper and pencil" proof of a previously published decentralized protocol for controlling a team of unmanned air vehicles [79]. Given the increasing complexity of both algorithms and software, this research concludes that analysis through formal methods or similar approaches will become increasingly important for ensuring system correctness within the U.S. Air Force and other industries.

Appendix A. Interpolation: Full Source Code

Listing A.1: test_interpolation.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Interpolation; use Interpolation;
3
4  procedure Test_Interpolation is
5     F : constant Func :=
6     ((-10, -10), (-1, -3), (0, 6), (5, 12), (12, 12), (18, 12), (20, 15));
7  begin
8     for A in -10 .. 20 loop
9        Put_Line ("Eval X =" & A'Image & " Y =" & Eval(F, A)'Image);
10    end loop;
11 end Test_Interpolation;
```

Listing A.2: interpolation.ads

```
1  package Interpolation with SPARK_Mode is
2
3     subtype Arg is Integer range -20_000 .. 20_000;
4     subtype Value is Integer range -20_000 .. 20_000;
5     type Point is record
6        X : Arg;
7        Y : Value;
8     end record;
9
10    type Index is new Integer range 1 .. 100;
11    type Func is array (Index range <>) of Point with
12       Predicate => Func'First = 1 and
13       (for all I in Func'Range =>
14          (for all J in Func'Range =>
15             (if I < J then Func(I).X < Func(J).X)));
16
17    function Monotonic_Increasing (F : Func) return Boolean is
18       (for all I in F'Range =>
19          (for all J in F'Range =>
20             (if I < J then F(I).Y <= F(J).Y)));
21
22    subtype Monotonic_Incr_Func is Func with
23       Predicate => Monotonic_Increasing (Monotonic_Incr_Func);
24
25    function Eval (F : Monotonic_Incr_Func; A : Arg) return Value with
26       Pre => F'Length > 0,
27       Post => (if A <= F(1).X then Eval'Result = F(1).Y
28          elsif A >= F(F'Last).X then Eval'Result = F(F'Last).Y
29          elsif (for some K in 1..F'Last => A = F(K).X and then
30             Eval'Result = F(K).X) then True
31             else (for some K in 1..F'Last - 1 =>
32                A in F(K).X..F(K+1).X and then
33                Eval'Result in F(K).Y..F(K+1).Y));
34
35    pragma Annotate (GNATprove, Terminating, Eval);
```

```
36 end Interpolation;
```

Listing A.3: interpolation.adb

```
1  package body Interpolation with SPARK_Mode is
2     function Eval (F : Monotonic_Incr_Func; A : Arg) return Value is
3     begin
4        if A <= F(1).X then
5           return F(1).Y;
6        elsif A >= F(F'Last).X then
7           return F(F'Last).Y;
8        end if;
9
10       for K in F'Range loop
11          pragma Loop_Invariant(A >= F(K).X);
12          if A = F(K).X then
13             return F(K).Y;
14          elsif (A > F(K).X and then A < F(K+1).X) then
15             declare
16                DX : constant Integer := F(K+1).X - F(K).X;
17                DY : constant Integer := F(K+1).Y - F(K).Y;
18                R  : constant Integer := F(K).Y + (A - F(K).X) * DY / DX;
19             begin
20                pragma Assert(R in F(K).Y..F(K+1).Y);
21                return R;
22             end;
23          end if;
24       end loop;
25
26       raise Program_Error;
27    end Eval;
28 end Interpolation;
```

Appendix B. Merge Sort: Full Source Code

Listing B.1: main.adb

```
1  with Mergesort_Types; use Mergesort_Types;
2  with Mergesort_Algorithm;
3
4  procedure Main is
5     A: Sort_Array(1 .. 10) := (90, 5, 200, 250, -1, 13, -72, 44, 47, 5);
6  begin
7     Print(A);
8     Mergesort_Algorithm.Recursive_Mergesort(A, A'First, A'Last);
9     Print(A);
10 end Main;
```

Listing B.2: mergesort_algorithm.ads

```
1  with Mergesort_Types; use Mergesort_Types;
2
3  package mergesort_algorithm with SPARK_Mode is
4
5     function Is_Ascending(A: Sort_Array) return Boolean is
6        (if A'Length > 1 then (for all I in A'Range =>
7           (if I < A'Last then A(I) <= A(I + 1))))
8     with Ghost;
9
10    procedure Recursive_Mergesort(A: in out Sort_Array; L, R: Sort_Index) with
11       Pre => A'Length >= 1
12          and then (L in A'Range and R in A'Range)
13          and then L <= R,
14       Post => Is_Ascending(A(L..R))
15          and (for all I in A'Range => (if I not in L..R then A(I) = A'Old(I)))
16          and (for all I in A'Range => (for some J in A'Range => A(I) = A'Old(J)));
17
18    procedure Merge(A: in out Sort_Array; L: Sort_Index;
19                    M: Sort_Index;        R: Sort_Index) with
20       Pre => (L in A'Range and R in A'Range)
21          and then L <= R
22          and then M in L..R
23          and then Is_Ascending(A(L..M))
24          and then Is_Ascending(A(M+1..R)),
25       Post => Is_Ascending(A(L..R))
26          and (for all I in A'Range => (if I not in L .. R then A(I) = A'Old(I)))
27          and (for all I in A'Range => (for some J in A'Range => A(I) = A'Old(J)));
28
29 end mergesort_algorithm;
```

Listing B.3: mergesort_algorithm.adb

```
1  with Mergesort_Types; use Mergesort_Types;
2
3  package body mergesort_algorithm with SPARK_Mode is
```

```
4
5     procedure Recursive_Mergesort(A: in out Sort_Array; L, R: Sort_Index) is
6                             M: Sort_Index;
7     begin
8        if (L < R) then
9           M := L+(R-l)/2;
10          Recursive_Mergesort(A, L, M);
11          Recursive_Mergesort(A, M+1, R);
12          Merge(A, L, M, R);
13       end if;
14    end Recursive_Mergesort;
15
16    procedure Merge(A: in out Sort_Array; L: in Sort_Index;
17                    M: in Sort_Index; R: in Sort_Index) is
18       n1: constant Natural := M - L + 1;
19       n2: constant Natural := R - M;
20       L_temp: constant Sort_Array(0..n1-1) := A(L .. M);
21       R_temp: constant Sort_Array(0..n2-1) := A(M+1 .. R);
22       ii, jj, kk: Natural := 0;
23    begin
24       while ii < n1 and jj < n2 loop
25          if L_temp(ii) <= R_temp(jj) then
26             A(L + kk) := L_temp(ii);
27             ii := ii + 1;
28          else
29             A(L + kk) := R_temp(jj);
30             jj := jj + 1;
31          end if;
32          kk := kk + 1;
33          pragma Loop_Invariant(ii <= n1 and jj <= n2);
34          pragma Loop_Invariant(kk = ii + jj);
35          pragma Loop_Invariant(Is_Ascending(A(L..L+(kk-1))));
36          pragma Loop_Invariant(for all I in L..L+(kk-1) =>
37             ((if ii < n1 then A(I) <= L_temp(ii)) and (if jj < n2 then A(I) <= R_temp(jj))));
38          pragma Loop_Invariant(for all I in A'Range => (if I not in L..L+(kk-1) then A(I) = A'
                Loop_Entry(I)));
39          pragma Loop_Invariant(for all I in L..L+(kk-1) =>
40             (for some J in A'Range => A(I) = A'Loop_Entry(J)));
41       end loop;
42
43       if ii < n1 then
44          A(L + kk .. R) := L_temp(ii .. n1-1);
45       elsif jj < n2 then
46          A(L + kk .. R) := R_temp(jj .. n2-1);
47       end if;
48    end Merge;
49
50 end mergesort_algorithm;
```

Listing B.4: mergesort_types.ads

```
1  package mergesort_types with SPARK_Mode is
2     subtype Sort_Index is Integer range 0 .. 99;
3     type Sort_Array is array(Sort_Index range <>) of Integer;
```

```
4       procedure Print(A : Sort_Array);
5   end mergesort_types;
```

Listing B.5: mergesort_types.adb

```
1   with Ada.Text_IO;
2
3   package body mergesort_types with SPARK_Mode is
4
5       procedure Print (A : in Sort_Array) is
6       begin
7           for I in A'Range loop
8               Ada.Text_IO.Put(Integer'Image(A(I)) & " ");
9           end loop;
10          Ada.Text_IO.Put_Line("");
11      end Print;
12
13  end mergesort_types;
```

Appendix C. Priority Queue: Full Source Code

Listing C.1: main.adb

```
1  with test_helpers; use test_helpers;
2  with priority; use priority;
3  with Ada.Text_IO; use Ada.Text_IO;
4
5
6  procedure Main is
7
8     PQ : Priority_Queue;
9
10    item_1 : Item_Priority_Pair := (("test_4", -100));
11    item_2 : Item_Priority_Pair := (("test_5", 400));
12    item_3 : Item_Priority_Pair := (("test_6", -888));
13    item_4 : Item_Priority_Pair := (("test_7", -1234));
14    item_5 : Item_Priority_Pair := (("test_8", 4));
15    item_6 : Item_Priority_Pair := (("test_9", -95669));
16    item_7 : Item_Priority_Pair := (("test10", 876543));
17
18 begin
19    test_insert(item_1, PQ);
20    test_insert(item_2, PQ);
21
22    Put_Line("");
23    Put_Line("RUN PROGRAM WITH ASSERTIONS ENABLED. ");
24    Put_Line("");
25    Put_Line("[INITIAL]: Here is the initial Priority Queue: ");
26    Print_Queue(PQ);
27
28
29    --add the seven equeues {item_1, item_2, ...., item_7}
30
31
32    test_insert(item_3, PQ);
33    test_insert(item_4, PQ);
34    test_insert(item_5, PQ);
35    test_insert(item_6, PQ);
36    test_insert(item_7, PQ);
37
38
39    -- remove elements from the Min Priority Queue
40
41    test_extract(PQ);
42    test_extract(PQ);
43    test_extract(PQ);
44    test_extract(PQ);
45    test_extract(PQ);
46    test_extract(PQ);
47    test_extract(PQ);
48
49
```

```
50      -- add two equeues {item_1, item_2}
51
52      test_insert(item_1, PQ);
53      test_insert(item_2, PQ);
54
55
56      -- remove elements from the Min Priority Queue
57
58      test_extract(PQ);
59      test_extract(PQ);
60
61  -- end of test demonstration
62
63  end Main;
```

Listing C.2: priority.ads

```
1   package priority with SPARK_Mode is
2
3      subtype Item is String(1..6);
4
5      subtype Given_Priority is Integer;
6
7      type Item_Priority_Pair is record
8         X: Item;
9         P: Given_Priority;
10     end record;
11
12     Initializer_Item : constant Item_Priority_Pair := (("------", Integer'Last));
13
14     type Priority_Queue is private;
15
16     procedure Print_Queue(PQ : in Priority_Queue);
17
18     function Is_Not_Full(PQ: Priority_Queue) return Boolean
19        with Ghost;
20
21     function Is_Not_Empty(PQ: Priority_Queue) return Boolean
22        with Ghost;
23
24     function Is_Min(Orig_Queue: Priority_Queue; Min_Priority_Found: Given_Priority) return Boolean
25        with Ghost;
26
27     function Is_At_End_Of_Queue(Orig_Queue, Result_Queue: Priority_Queue; inserted_pair:
           Item_Priority_Pair) return Boolean
28        with Ghost,
29        Pre => Is_Not_Empty(Result_Queue);
30
31     function Is_First_Extracted(Orig_Queue, Result_Queue: Priority_Queue; extracted_pair:
           Item_Priority_Pair) return Boolean
32        with Ghost;
33
34     function Did_Queue_Increase(Orig_Queue, Result_Queue: Priority_Queue) return Boolean
35        with Ghost;
```

```
36
37      function Did_Queue_Decrease(Orig_Queue, Result_Queue: Priority_Queue) return Boolean
38         with Ghost;
39
40      procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair) with
41         Pre =>
42            Is_Not_Full(PQ),
43         Post =>
44            Did_Queue_Increase(PQ'Old, PQ)
45            and Is_At_End_Of_Queue(PQ'Old, PQ, pair);
46
47      procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair) with
48         Pre =>
49            Is_Not_Empty(PQ),
50         Post =>
51            Did_Queue_Decrease(PQ'Old, PQ)
52            and Is_Min(PQ'Old, pair.P)
53            and Is_First_Extracted(PQ'Old, PQ, pair);
54
55      pragma Annotate (GNATprove, Terminating, insert);
56      pragma Annotate (GNATprove, Terminating, extract);
57
58   private
59
60      subtype Index_Range is Natural range 1 .. 10;
61      subtype REAR_Index_Range is Natural range 0..Index_Range'Last;
62
63      type Item_Priority_Array is array(Index_Range) of Item_Priority_Pair;
64
65      type Priority_Queue is record
66         PQ: Item_Priority_Array;
67         REAR: REAR_Index_Range := 0;
68      end record;
69
70   end priority;
```

Listing C.3: priority.adb

```
1   with Ada.Text_IO; use Ada.Text_IO;
2
3   package body priority with SPARK_Mode is
4
5      procedure Print_Queue(PQ : in Priority_Queue) is
6      begin
7         if Length(PQ) = 0 then
8            Put_Line("EMPTY_Queue");
9            Put_Line("");
10        else
11           for I in 1..PQ.REAR loop
12              Put("Element " & I'Img & ": ");
13              Put("(" & (PQ.PQ(I).X) & ", ");
14              Put_Line(Integer'Image(PQ.PQ(I).P) & ") ");
15           end loop;
16           Put_Line("");
```

```
17          end if;
18       end Print_Queue;
19
20       function Is_Not_Full(PQ: Priority_Queue) return Boolean is
21          (PQ.REAR < REAR_Index_Range'Last);
22
23       function Is_Not_Empty(PQ: Priority_Queue) return Boolean is
24          (PQ.REAR > REAR_Index_Range'First);
25
26       function Is_Min(Orig_Queue: Priority_Queue; Min_Priority_Found: Given_Priority) return Boolean
             is
27          (for all I in 1 .. Orig_Queue.REAR => Orig_Queue.PQ(I).P >= Min_Priority_Found);
28
29       function Is_At_End_Of_Queue(Orig_Queue, Result_Queue: Priority_Queue; inserted_pair:
             Item_Priority_Pair) return Boolean is
30          ((Result_Queue.PQ(Result_Queue.REAR) = inserted_pair)
31          and
32          (for all I in 1 .. Orig_Queue.REAR => Orig_Queue.PQ(I) = Result_Queue.PQ(I)));
33
34       function Is_First_Extracted(Orig_Queue, Result_Queue: Priority_Queue; extracted_pair:
             Item_Priority_Pair) return Boolean is
35          (for some I in 1 .. Orig_Queue.REAR =>
36             extracted_pair = orig_queue.PQ(I)
37          and then (for all X in 1 .. I-1 => (Result_Queue.PQ(X).P > extracted_pair.P))
38          and then (Orig_Queue.PQ(1..I-1) = Result_Queue.PQ(1..I-1))
39          and then (Orig_Queue.PQ(I+1..Orig_Queue.REAR) = Result_Queue.PQ(I..Result_Queue.REAR)));
40
41       function Did_Queue_Increase(Orig_Queue, Result_Queue: Priority_Queue) return Boolean is
42          (Result_Queue.REAR = Orig_Queue.REAR + 1);
43
44       function Did_Queue_Decrease(Orig_Queue, Result_Queue: Priority_Queue) return Boolean is
45          (Result_Queue.REAR = Orig_Queue.REAR - 1);
46
47       procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair) is
48       begin
49          PQ.PQ(PQ.REAR + 1) := pair;
50          PQ.REAR := PQ.REAR + 1;
51       end insert;
52
53       procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair) is
54          new_priority_queue : Priority_Queue;
55          new_array : Item_Priority_Array := PQ.PQ;
56          min_index : Index_Range := 1;
57          orig_queue : constant Priority_Queue := PQ;
58       begin
59          for I in 1..PQ.REAR loop
60             if PQ.PQ(I).P < PQ.PQ(min_index).P then
61                min_index := I;
62             end if;
63             pragma Loop_Invariant(min_index in 1 .. I);
64             pragma Loop_Invariant(for all E in 1 .. I => PQ.PQ(E).P >= PQ.PQ(min_index).P);
65             pragma Loop_Invariant(for all E in 1 .. min_index-1 => PQ.PQ(E).P > PQ.PQ(min_index).P);
66          end loop;
67
```

```
68          new_array(1..min_index-1) := PQ.PQ(1..min_index-1);
69          new_array(min_index..PQ.REAR-1) := PQ.PQ(min_index+1..PQ.REAR);
70
71          new_priority_queue.PQ := new_array;
72          new_priority_queue.REAR := PQ.REAR - 1;
73
74          pair := PQ.PQ(min_index);
75          PQ := new_priority_queue;
76
77          pragma Assert(pair = orig_queue.PQ(min_index)
78             and then (for all X in 1 .. min_index-1 => (PQ.PQ(X).P > orig_queue.PQ(min_index).P))
79             and then (orig_queue.PQ(1..min_index-1) = PQ.PQ(1..min_index-1))
80             and then (orig_queue.PQ(min_index+1..orig_queue.REAR) = PQ.PQ(min_index..PQ.REAR)));
81
82          pragma Assert(for some I in 1 .. orig_queue.REAR =>
83             pair = orig_queue.PQ(I)
84             and then (for all X in 1 .. I-1 => (PQ.PQ(X).P > orig_queue.PQ(I).P))
85             and then (orig_queue.PQ(1..I-1) = PQ.PQ(1..I-1))
86             and then (orig_queue.PQ(I+1..orig_queue.REAR) = PQ.PQ(I..PQ.REAR)));
87       end extract;
88    end priority;
```

Listing C.4: test_helpers.ads

```
1  with priority; use priority;
2
3  package test_helpers is
4
5     procedure test_insert(pair: Item_Priority_Pair; PQ: in out Priority_Queue);
6
7     procedure test_extract(PQ: in out Priority_Queue);
8
9  end test_helpers;
```

Listing C.5: test_helpers.adb

```
1  with priority; use priority;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  -- Below are helper functions that just provide the user visual output
5  -- These functions {initialize, test_insert, test_extract} are called by
6  -- the main driver function to demonstrate the priority queue implementation
7
8  -- These helper function call the priority queue implementations of insert and extract
9  -- Insert is called on [line 39]
10 -- Extract is called on [line 51]
11 package body test_helpers is
12
13    procedure test_insert(pair: Item_Priority_Pair; PQ: in out Priority_Queue) is
14    begin
15       Put_Line("[INSERT]: Here is the resulting Priority Queue: ");
16       Insert(PQ, pair); -- INSERT
17       Print_Queue(PQ);
```

```
18      end test_insert;
19
20      procedure test_extract(PQ: in out Priority_Queue) is
21          temp_extract_return : Item_Priority_Pair;
22      begin
23          Put_Line("[EXTRACT]: Here is the resulting Priority Queue: ");
24          extract(PQ, temp_extract_return); -- EXTRACT
25          Put_Line("The Item removed was: [" & temp_extract_return.X & "] with Priority: [" &
                temp_extract_return.P'Img & "] ");
26          Print_Queue(PQ);
27      end test_extract;
28  end test_helpers;
```

Appendix D. Interpolation: GNATprove Output

```
1  package body Interpolation with SPARK_Mode is
2     function Eval (F : Monotonic_Incr_Func; A : Arg) return Value is
3     begin
4        if A <= F(1).X then
5           return F(1).Y;
6        elsif A >= F(F'Last).X then
7           return F(F'Last).Y;
8        end if;
9
10       for K in F'Range loop
11          pragma Loop_Invariant(A >= F(K).X);
12          if A = F(K).X then
13             return F(K).Y;
14          elsif (A > F(K).X and then A < F(K+1).X) then
15             declare
16                DX : constant Integer := F(K+1).X - F(K).X;
17                DY : constant Integer := F(K+1).Y - F(K).Y;
18                R  : constant Integer := F(K).Y + (A - F(K).X) * DY / DX;
19             begin
20                pragma Assert(R in F(K).Y..F(K+1).Y);
21                return R;
22             end;
23          end if;
24       end loop;
25
26       raise Program_Error;
27    end Eval;
28 end Interpolation;
```

Figure 23: Highlighted interpolation .adb GNATprove results on source code.

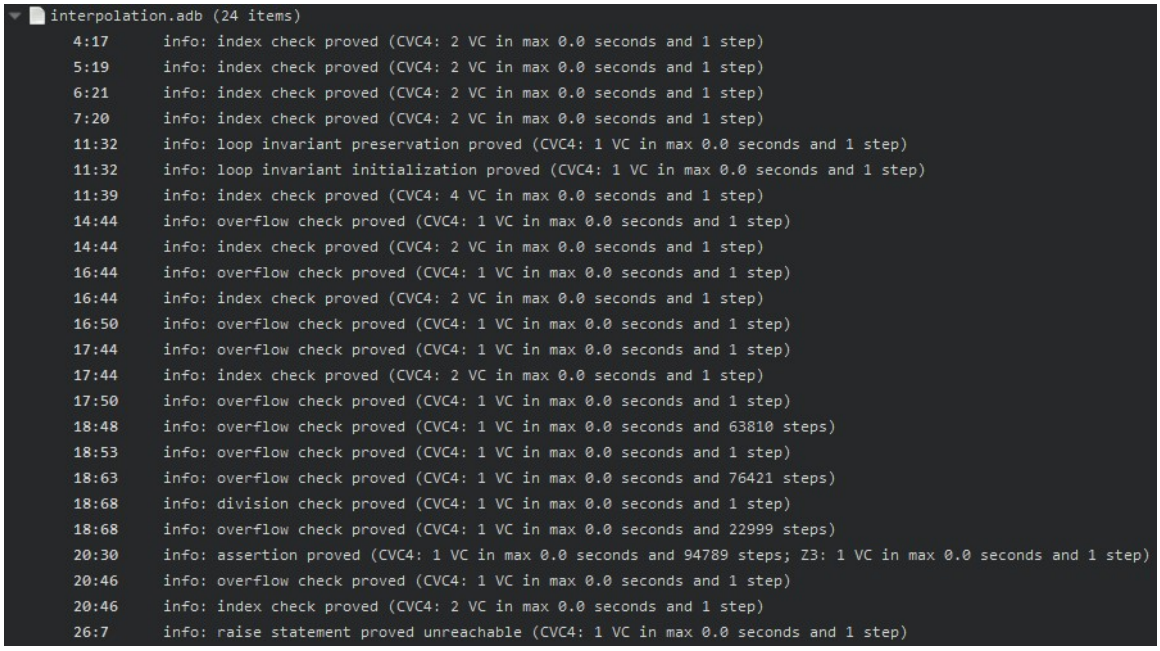
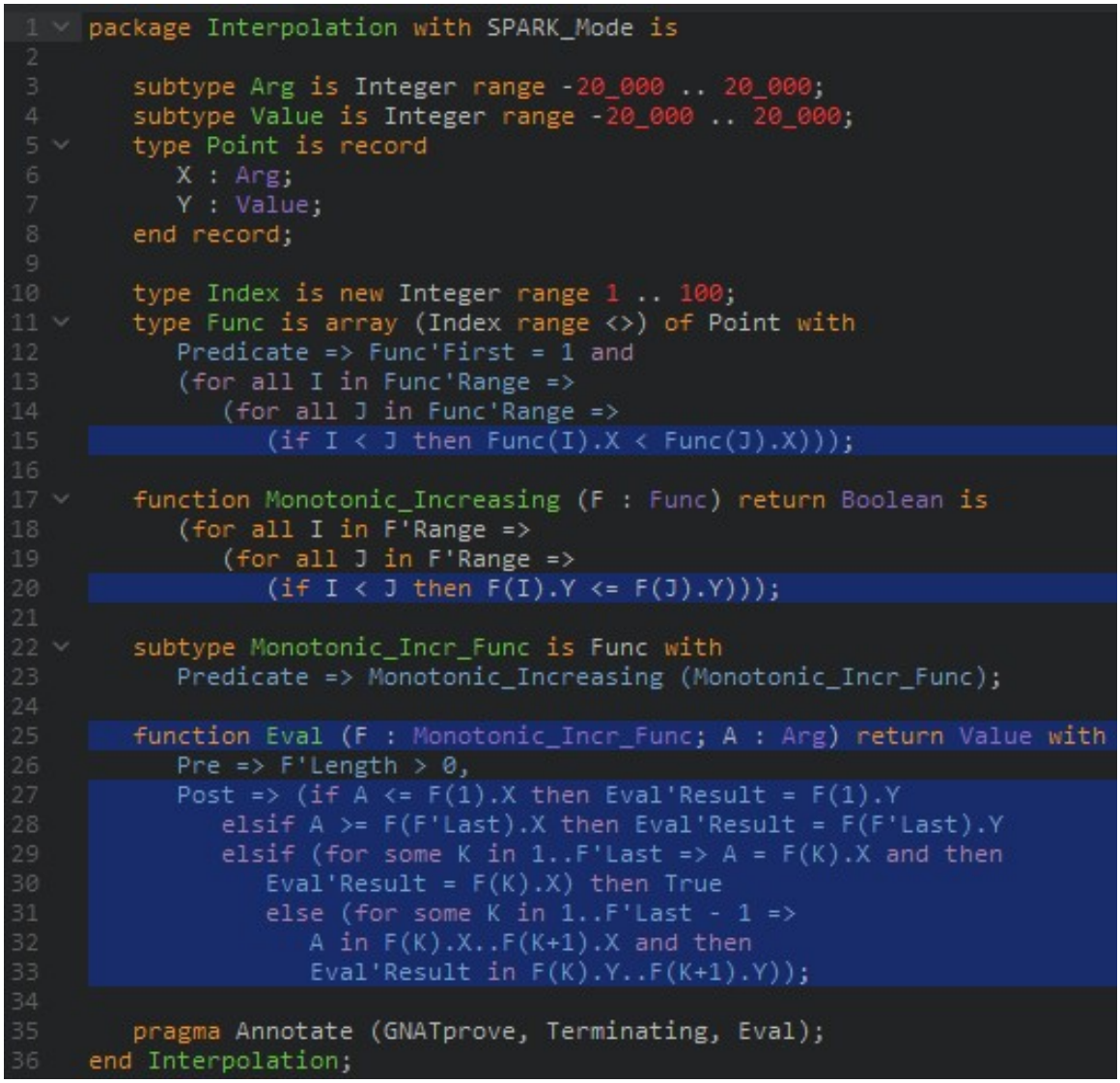


Figure 24: interpolation .adb GNATprove results.



```
package Interpolation with SPARK_Mode is

   subtype Arg is Integer range -20_000 .. 20_000;
   subtype Value is Integer range -20_000 .. 20_000;
   type Point is record
      X : Arg;
      Y : Value;
   end record;

   type Index is new Integer range 1 .. 100;
   type Func is array (Index range <>) of Point with
      Predicate => Func'First = 1 and
      (for all I in Func'Range =>
         (for all J in Func'Range =>
            (if I < J then Func(I).X < Func(J).X)));

   function Monotonic_Increasing (F : Func) return Boolean is
      (for all I in F'Range =>
         (for all J in F'Range =>
            (if I < J then F(I).Y <= F(J).Y)));

   subtype Monotonic_Incr_Func is Func with
      Predicate => Monotonic_Increasing (Monotonic_Incr_Func);

   function Eval (F : Monotonic_Incr_Func; A : Arg) return Value with
      Pre => F'Length > 0,
      Post => (if A <= F(1).X then Eval'Result = F(1).Y
         elsif A >= F(F'Last).X then Eval'Result = F(F'Last).Y
         elsif (for some K in 1..F'Last => A = F(K).X and then
            Eval'Result = F(K).X) then True
            else (for some K in 1..F'Last - 1 =>
               A in F(K).X..F(K+1).X and then
               Eval'Result in F(K).Y..F(K+1).Y));

   pragma Annotate (GNATprove, Terminating, Eval);
end Interpolation;
```

Figure 25: Highlighted interpolation .ads GNATprove results on source code.

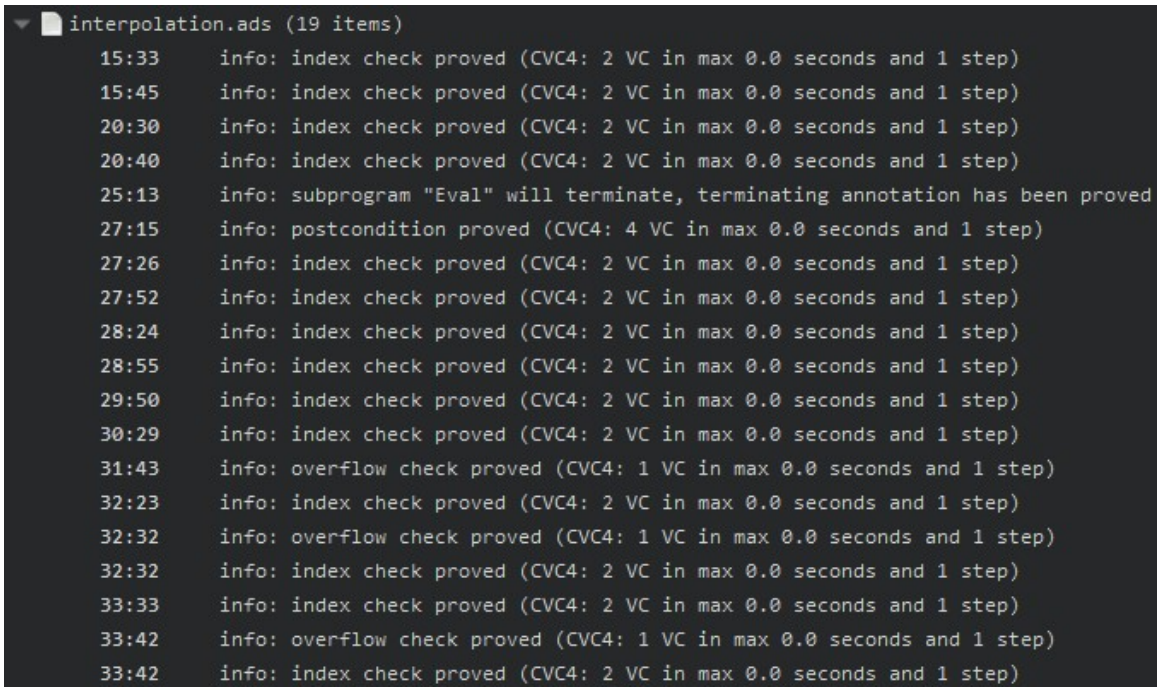


Figure 26: interpolation .ads GNATprove results.

Appendix E. Merge Sort: GNATprove Output

```
with Mergesort_Types; use Mergesort_Types;

package body mergesort_algorithm with SPARK_Mode is

   procedure Recursive_Mergesort(A: in out Sort_Array; L, R: Sort_Index) is
                                 M: Sort_Index;
   begin
      if (L < R) then
         M := L+(R-1)/2;
         Recursive_Mergesort(A, L, M);
         Recursive_Mergesort(A, M+1, R);
         Merge(A, L, M, R);
      end if;
   end Recursive_Mergesort;

   procedure Merge(A: in out Sort_Array; L: in Sort_Index;
                   M: in Sort_Index; R: in Sort_Index) is
      n1: constant Natural := M - L + 1;
      n2: constant Natural := R - M;
      L_temp: constant Sort_Array(0..n1-1) := A(L .. M);
      R_temp: constant Sort_Array(0..n2-1) := A(M+1 .. R);
      ii, jj, kk: Natural := 0;
   begin
      while ii < n1 and jj < n2 loop
         if L_temp(ii) <= R_temp(jj) then
            A(L + kk) := L_temp(ii);
            ii := ii + 1;
         else
            A(L + kk) := R_temp(jj);
            jj := jj + 1;
         end if;
         kk := kk + 1;
         pragma Loop_Invariant(ii <= n1 and jj <= n2);
         pragma Loop_Invariant(kk = ii + jj);
         pragma Loop_Invariant(Is_Ascending(A(L..L+(kk-1))));
         pragma Loop_Invariant(for all I in L..L+(kk-1) =>
            ((if ii < n1 then A(I) <= L_temp(ii)) and (if jj < n2 then A(I) <= R_temp(jj))));
         pragma Loop_Invariant(for all I in A'Range => (if I not in L..L+(kk-1) then A(I) = A'Loop_Entry(I)));
         pragma Loop_Invariant(for all I in L..L+(kk-1) =>
            (for some J in A'Range => A(I) = A'Loop_Entry(J)));
      end loop;

      if ii < n1 then
         A(L + kk .. R) := L_temp(ii .. n1-1);
      elsif jj < n2 then
         A(L + kk .. R) := R_temp(jj .. n2-1);
      end if;
   end Merge;

end mergesort_algorithm;
```

Figure 27: Highlighted mergesort_algorithm.adb GNATprove results on source code.

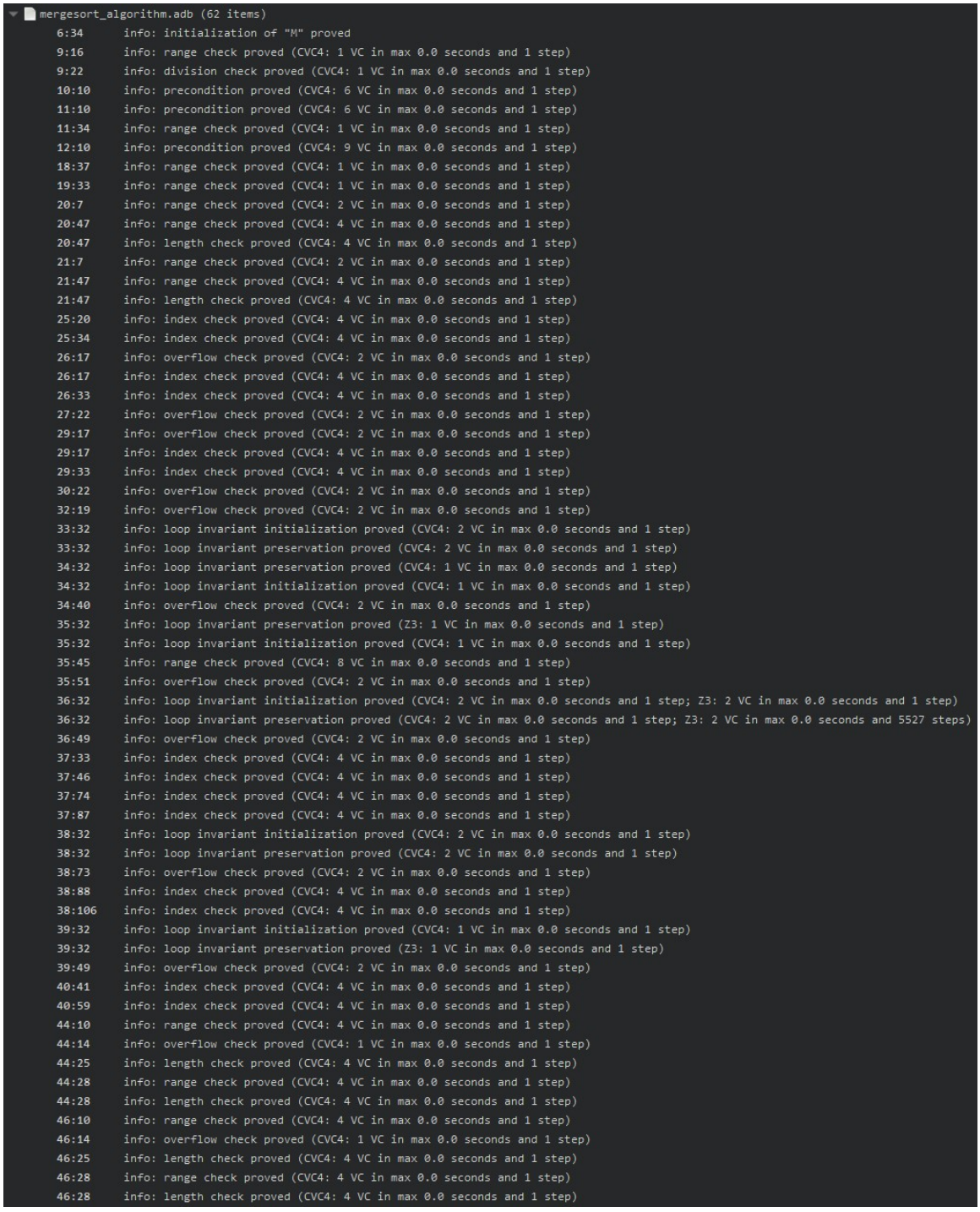
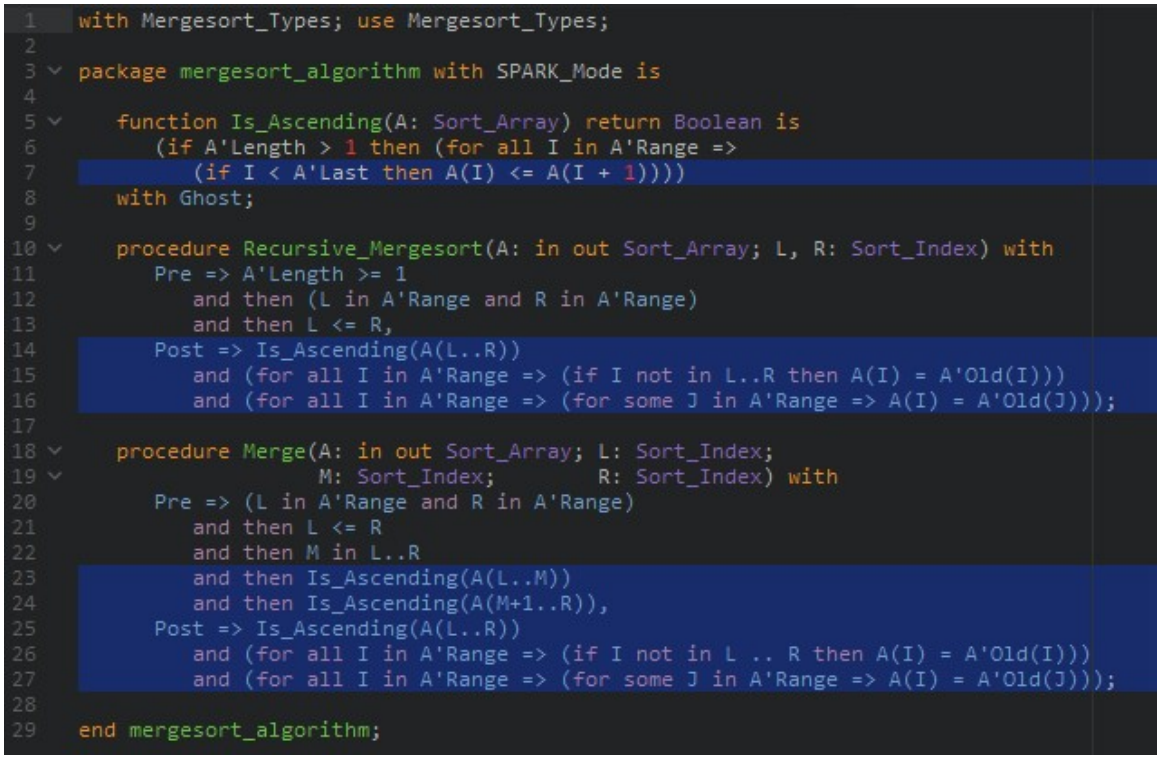
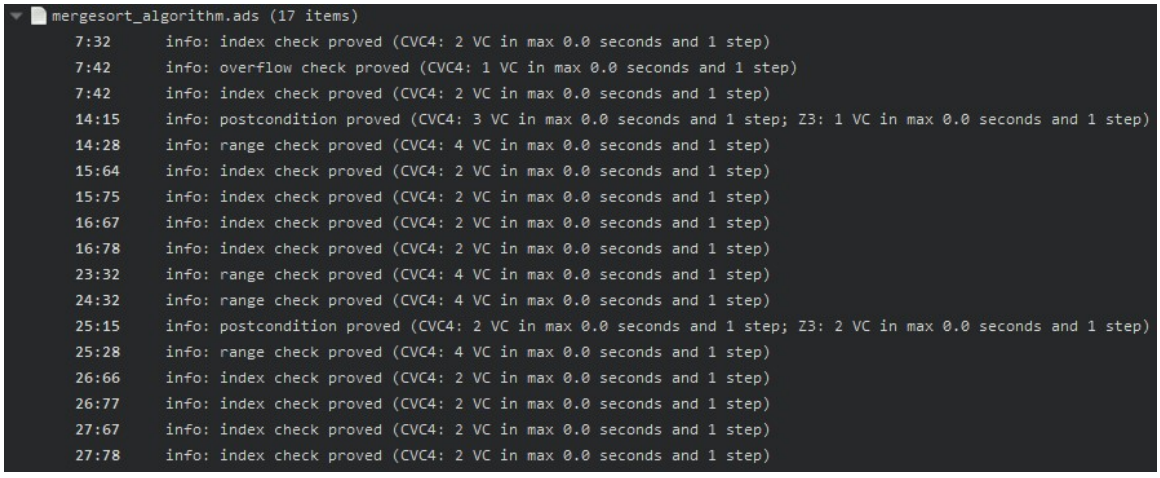


Figure 28: mergesort_algorithm.adb GNATprove results.



```
with Mergesort_Types; use Mergesort_Types;

package mergesort_algorithm with SPARK_Mode is

   function Is_Ascending(A: Sort_Array) return Boolean is
      (if A'Length > 1 then (for all I in A'Range =>
         (if I < A'Last then A(I) <= A(I + 1))))
   with Ghost;

   procedure Recursive_Mergesort(A: in out Sort_Array; L, R: Sort_Index) with
      Pre => A'Length >= 1
         and then (L in A'Range and R in A'Range)
         and then L <= R,
      Post => Is_Ascending(A(L..R))
         and (for all I in A'Range => (if I not in L..R then A(I) = A'Old(I)))
         and (for all I in A'Range => (for some J in A'Range => A(I) = A'Old(J)));

   procedure Merge(A: in out Sort_Array; L: Sort_Index;
                   M: Sort_Index;        R: Sort_Index) with
      Pre => (L in A'Range and R in A'Range)
         and then L <= R
         and then M in L..R
         and then Is_Ascending(A(L..M))
         and then Is_Ascending(A(M+1..R)),
      Post => Is_Ascending(A(L..R))
         and (for all I in A'Range => (if I not in L .. R then A(I) = A'Old(I)))
         and (for all I in A'Range => (for some J in A'Range => A(I) = A'Old(J)));

end mergesort_algorithm;
```

Figure 29: Highlighted mergesort_algorithm.ads GNATprove results on source code.



```
mergesort_algorithm.ads (17 items)
  7:32    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
  7:42    info: overflow check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
  7:42    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 14:15    info: postcondition proved (CVC4: 3 VC in max 0.0 seconds and 1 step; Z3: 1 VC in max 0.0 seconds and 1 step)
 14:28    info: range check proved (CVC4: 4 VC in max 0.0 seconds and 1 step)
 15:64    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 15:75    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 16:67    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 16:78    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 23:32    info: range check proved (CVC4: 4 VC in max 0.0 seconds and 1 step)
 24:32    info: range check proved (CVC4: 4 VC in max 0.0 seconds and 1 step)
 25:15    info: postcondition proved (CVC4: 2 VC in max 0.0 seconds and 1 step; Z3: 2 VC in max 0.0 seconds and 1 step)
 25:28    info: range check proved (CVC4: 4 VC in max 0.0 seconds and 1 step)
 26:66    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 26:77    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 27:67    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
 27:78    info: index check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
```

Figure 30: mergesort_algorithm.ads GNATprove results.

Appendix F. Priority Queue: GNATprove Output

```
with Ada.Text_IO; use Ada.Text_IO;

package body priority with SPARK_Mode is

   procedure Print_Queue(PQ : in Priority_Queue) is
   begin
      if PQ.REAR = 0 then
         Put_Line("EMPTY_Queue");
         Put_Line("");
      else
         for I in 1..PQ.REAR loop
            Put("Element " & I'Img & ": ");
            Put("(" & (PQ.PQ(I).X) & ", ");
            Put_Line(Integer'Image(PQ.PQ(I).P) & ") ");
         end loop;
         Put_Line("");
      end if;
   end Print_Queue;

   function Is_Not_Full(PQ: Priority_Queue) return Boolean is
     (PQ.REAR < REAR_Index_Range'Last);

   function Is_Not_Empty(PQ: Priority_Queue) return Boolean is
     (PQ.REAR > REAR_Index_Range'First);

   function Is_Min(Orig_Queue: Priority_Queue; Min_Priority_Found: Given_Priority) return Boolean is
     (for all I in 1 .. Orig_Queue.REAR => Orig_Queue.PQ(I).P >= Min_Priority_Found);

   function Is_At_End_Of_Queue(Orig_Queue, Result_Queue: Priority_Queue; inserted_pair: Item_Priority_Pair) return Boolean is
     ((Result_Queue.PQ(Result_Queue.REAR) = inserted_pair)
      and
     (for all I in 1 .. Orig_Queue.REAR => Orig_Queue.PQ(I) = Result_Queue.PQ(I)));

   function Is_First_Extracted(Orig_Queue, Result_Queue: Priority_Queue; extracted_pair: Item_Priority_Pair) return Boolean is
     (for some I in 1 .. Orig_Queue.REAR =>
         extracted_pair = orig_queue.PQ(I)
      and then (for all X in 1 .. I-1 => (Result_Queue.PQ(X).P > extracted_pair.P))
      and then (Orig_Queue.PQ(1..I-1) = Result_Queue.PQ(1..I-1))
      and then (Orig_Queue.PQ(I+1..Orig_Queue.REAR) = Result_Queue.PQ(I..Result_Queue.REAR)));

   function Did_Queue_Increase(Orig_Queue, Result_Queue: Priority_Queue) return Boolean is
     (Result_Queue.REAR = Orig_Queue.REAR + 1);

   function Did_Queue_Decrease(Orig_Queue, Result_Queue: Priority_Queue) return Boolean is
     (Result_Queue.REAR = Orig_Queue.REAR - 1);

   procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair) is
   begin
      PQ.PQ(PQ.REAR + 1) := pair;
      PQ.REAR := PQ.REAR + 1;
   end insert;

   procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair) is
      new_priority_queue : Priority_Queue;
      new_array : Item_Priority_Array := PQ.PQ;
      min_index : Index_Range := 1;
      orig_queue : constant Priority_Queue := PQ;
   begin
      for I in 1..PQ.REAR loop
         if PQ.PQ(I).P < PQ.PQ(min_index).P then
            min_index := I;
         end if;
         pragma Loop_Invariant(min_index in 1 .. I);
         pragma Loop_Invariant(for all E in 1 .. I => PQ.PQ(E).P >= PQ.PQ(min_index).P);
         pragma Loop_Invariant(for all E in 1 .. min_index-1 => PQ.PQ(E).P > PQ.PQ(min_index).P);
      end loop;

      new_array(1..min_index-1) := PQ.PQ(1..min_index-1);
      new_array(min_index..PQ.REAR-1) := PQ.PQ(min_index+1..PQ.REAR);

      new_priority_queue.PQ := new_array;
      new_priority_queue.REAR := PQ.REAR - 1;

      pair := PQ.PQ(min_index);
      PQ := new_priority_queue;

      pragma Assert(pair = orig_queue.PQ(min_index)
                    and then (for all X in 1 .. min_index-1 => (PQ.PQ(X).P > orig_queue.PQ(min_index).P))
                    and then (orig_queue.PQ(1..min_index-1) = PQ.PQ(1..min_index-1))
                    and then (orig_queue.PQ(min_index+1..orig_queue.REAR) = PQ.PQ(min_index..PQ.REAR)));

      pragma Assert(for some I in 1 .. orig_queue.REAR =>
                      pair = orig_queue.PQ(I)
                    and then (for all X in 1 .. I-1 => (PQ.PQ(X).P > orig_queue.PQ(I).P))
                    and then (orig_queue.PQ(1..I-1) = PQ.PQ(1..I-1))
                    and then (orig_queue.PQ(I+1..orig_queue.REAR) = PQ.PQ(I..PQ.REAR)));
   end extract;
end priority;
```

Figure 31: Highlighted priority .adb GNATprove results on source code.

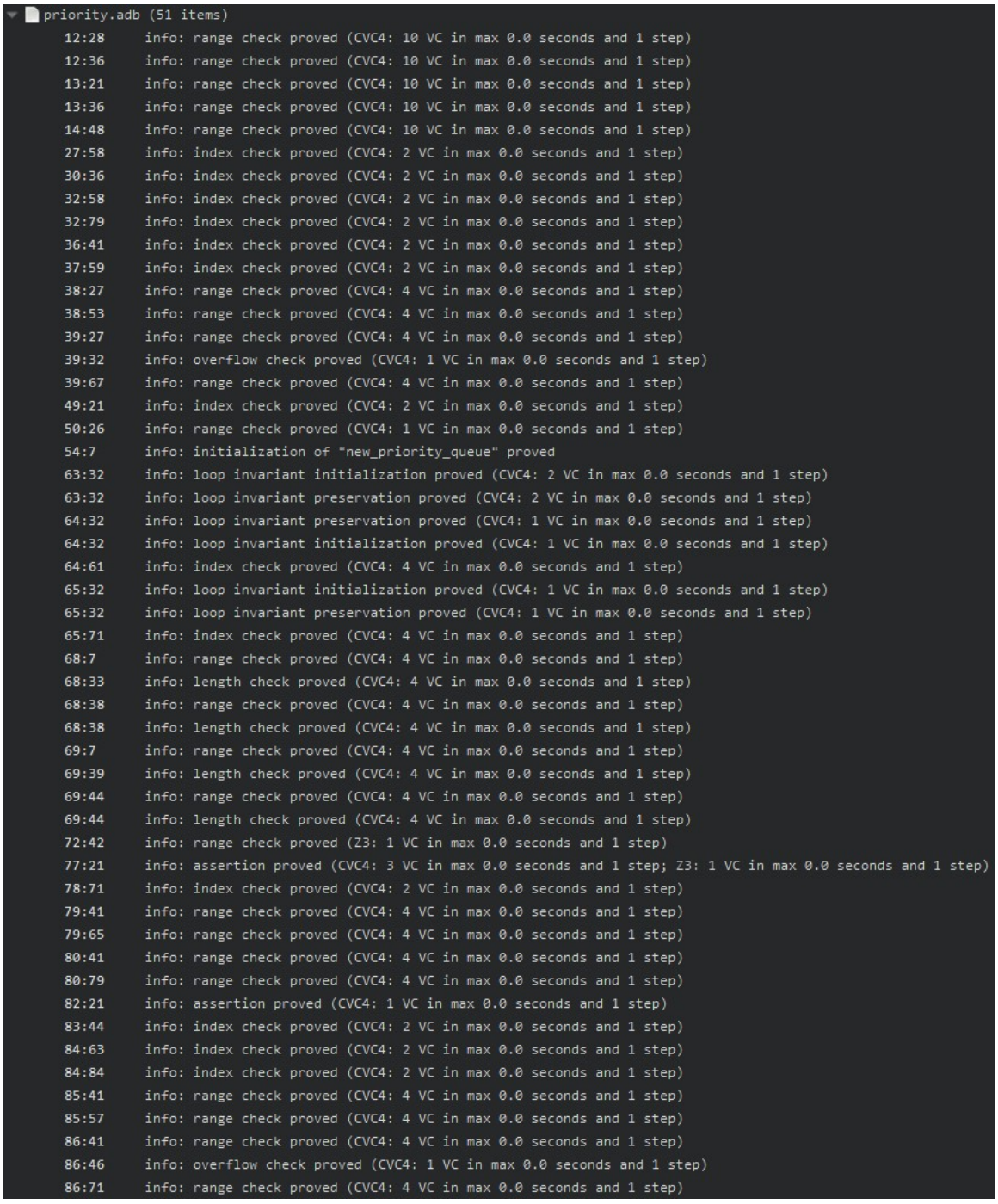
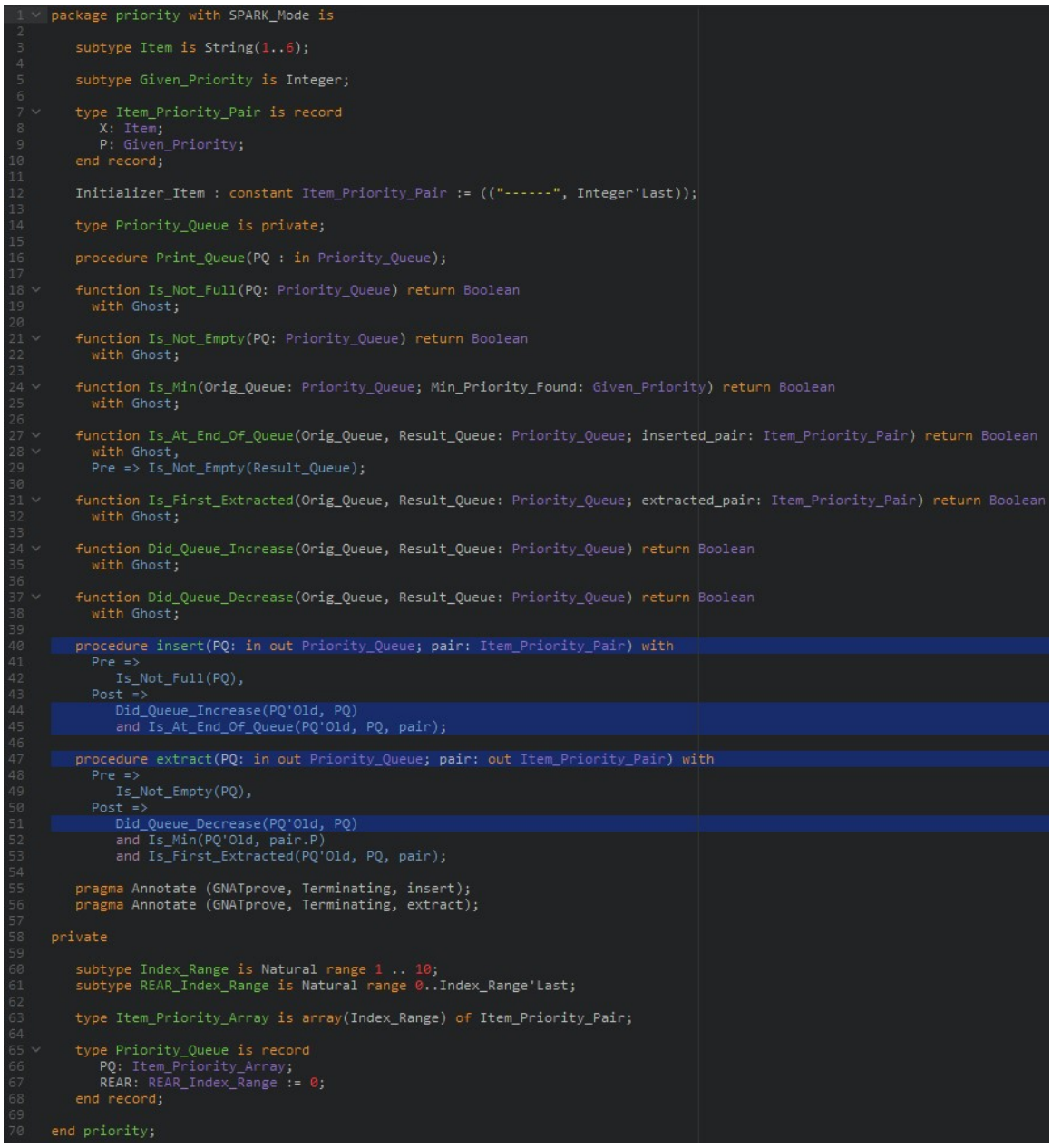


Figure 32: priority .adb GNATprove results.



```
package priority with SPARK_Mode is

   subtype Item is String(1..6);

   subtype Given_Priority is Integer;

   type Item_Priority_Pair is record
      X: Item;
      P: Given_Priority;
   end record;

   Initializer_Item : constant Item_Priority_Pair := (("------", Integer'Last));

   type Priority_Queue is private;

   procedure Print_Queue(PQ : in Priority_Queue);

   function Is_Not_Full(PQ: Priority_Queue) return Boolean
     with Ghost;

   function Is_Not_Empty(PQ: Priority_Queue) return Boolean
     with Ghost;

   function Is_Min(Orig_Queue: Priority_Queue; Min_Priority_Found: Given_Priority) return Boolean
     with Ghost;

   function Is_At_End_Of_Queue(Orig_Queue, Result_Queue: Priority_Queue; inserted_pair: Item_Priority_Pair) return Boolean
     with Ghost,
     Pre => Is_Not_Empty(Result_Queue);

   function Is_First_Extracted(Orig_Queue, Result_Queue: Priority_Queue; extracted_pair: Item_Priority_Pair) return Boolean
     with Ghost;

   function Did_Queue_Increase(Orig_Queue, Result_Queue: Priority_Queue) return Boolean
     with Ghost;

   function Did_Queue_Decrease(Orig_Queue, Result_Queue: Priority_Queue) return Boolean
     with Ghost;

   procedure insert(PQ: in out Priority_Queue; pair: Item_Priority_Pair) with
     Pre =>
        Is_Not_Full(PQ),
     Post =>
        Did_Queue_Increase(PQ'Old, PQ)
        and Is_At_End_Of_Queue(PQ'Old, PQ, pair);

   procedure extract(PQ: in out Priority_Queue; pair: out Item_Priority_Pair) with
     Pre =>
        Is_Not_Empty(PQ),
     Post =>
        Did_Queue_Decrease(PQ'Old, PQ)
        and Is_Min(PQ'Old, pair.P)
        and Is_First_Extracted(PQ'Old, PQ, pair);

   pragma Annotate (GNATprove, Terminating, insert);
   pragma Annotate (GNATprove, Terminating, extract);

private

   subtype Index_Range is Natural range 1 .. 10;
   subtype REAR_Index_Range is Natural range 0..Index_Range'Last;

   type Item_Priority_Array is array(Index_Range) of Item_Priority_Pair;

   type Priority_Queue is record
      PQ: Item_Priority_Array;
      REAR: REAR_Index_Range := 0;
   end record;

end priority;
```

Figure 33: Highlighted priority .ads GNATprove results on source code.

```
priority.ads (6 items)
   40:14     info: subprogram "insert" will terminate, terminating annotation has been proved
   44:9      info: postcondition proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
   45:13     info: precondition proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
   47:14     info: subprogram "extract" will terminate, terminating annotation has been proved
   47:49     info: initialization of "pair" proved
   51:9      info: postcondition proved (CVC4: 2 VC in max 0.0 seconds and 1 step; Z3: 1 VC in max 0.0 seconds and 1 step)
```

Figure 34: priority .ads GNATprove results.

Bibliography

1. Nidhi Kalra and Susan M. Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? Technical Report RR-1478-RC, RAND Corporation, Santa Monica, CA, 2016.

2. J. Hansson, S. Helton, and P. Feiler. ROI analysis of the System Architecture Virtual Integration initiative. Technical Report CMU/SEI-2018-TR-002, Software Engineering Institute, Carnegie Mellon University, 2018.

3. Dennis Burke. All circuits are busy now: The 1990 AT&T long distance network collapse. Technical Report CSC440-01, California Polytechnic State University, Nov 1995.

4. Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of Heartbleed. In *ACM Internet Measurement Conference (IMC)*, pages 475–488, 2014.

5. Bryce Alexander, Sohaib Haseeb, and Adrian Baranchuk. Are implanted electronic devices hackable? *Trends in Cardiovascular Medicine*, 29(8):476–480, 2019.

6. What is WannaCry ransomware?, 2020.

7. Andy Greenberg. The untold story of NotPetya, the most devastating cyberattack in history, 2018.

8. Eray Yağdereli, Cemal Gemci, and A Ziya Aktaş. A study on cyber-security of autonomous and unmanned vehicles. *The Journal of Defense Modeling and Simulation*, 12(4):369–381, 2015.

9. Yannick Moy and M. Anthony Aiello. When testing is not enough, 2020.

10. 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.

11. Zongyu Cao, Wanyou Lv, Yanhong Huang, Jianqi Shi, and Qin Li. Formal analysis and verification of airborne software based on DO-333. *Electronics*, 9(2):327, 2020.

12. 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.

13. Klaas Wijbrans, Franc Buve, Robin Rijkers, and Wouter Geurts. Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. In *International Symposium on Formal Methods (FM)*, pages 419–424. Springer, 2008.

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

15. Darren Cofer and Steven Miller. DO-333 certification case studies. In *NASA Formal Methods Symposium (NFM)*, pages 1–15. Springer, 2014.

16. 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.

17. Emmanuel Ledinot, Jean-Paul Blanquart, Jean-Marc Astruc, Philippe Baufreton, Jean-Louis Boulanger, Cyrille Comar, Hervé Delseny, Jean Gassino, Michel Leeman, Philippe Quéré, et al. Joint use of static and dynamic software verification techniques: A cross-domain view in safety critical system industries. In *Embedded Real Time Software and Systems (ERTS)*, pages 1–10, 2014.

18. Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In *International Symposium on Formal Methods (FM)*, pages 532–546. Springer, 2009.

19. Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Software*, 30(3):50–57, 2013.

20. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft Research, Jan 2004.

21. 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.

22. 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 *Int. Conf. Computer Aided Verification*, pages 231–241. Springer, 2019.

23. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael

Norrish, et al. seL4: Formal verification of an OS kernel. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

24. John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec 1993.

25. RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71). Software considerations in airborne systems and equipment certification. Technical Report DO-178C, RTCA, Inc., 2011.

26. 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.

27. Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and GNATprove. *International Journal of Software Tools for Technology Transfer*, 17(6):695–707, 2015.

28. John W McCormick and Peter C Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

29. Introduction to SPARK, 2020.

30. Martin Becker, Emanuel Regnath, and Samarjit Chakraborty. Development and verification of a flight stack for a high-altitude glider in Ada/SPARK 2014. In *International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 105–116. Springer, 2017.

31. Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. *European Congress on Embedded Real Time Systems (ERTS)*, 19, 2014.

32. Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

33. Richard A. Kemmerer. Integrating formal methods into the development process. *IEEE software*, 7(5):37–50, 1990.

34. S Easterbrook, R Lutz, JK Covington, J Kelly, M Ampo, and D Hamilton. Experiences using formal methods for requirement modeling. nasa/wvu software research lab, fairmont. *Techn. Report# NASA-IVV-96-018*, 1996.

35. Constance Heitmeyer, Myla Archer, Elizabeth Leonard, and John McLean. Applying formal methods to a certifiably secure software system. *IEEE Transactions on Software Engineering*, 34(1):82–98, 2008.

36. Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

37. Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

38. Introduction to Ada, 2020.

39. John Barnes. *Programming in Ada 2012*. Cambridge University Press, 2014.

40. Yannick Moy. Climbing the software assurance ladder – practical formal verification for reliable software. *Electronic Communications of the EASST*, 76, 2019.

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

42. Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, 2000.

43. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International, 1996.

44. David Garlan. Preconditions for understanding (formal specification). In *Proceedings of the Sixth International Workshop on Software Specification and Design*, pages 242–243. IEEE Computer Society, 1991.

45. Roderick Chapman. Industrial experience with SPARK. *ACM SIGAda Ada Letters*, 20(4):64–68, 2000.

46. Martin Croxford and James Sutton. Breaking through the v and v bottleneck. In *International Eurospace-Ada-Europe Symposium*, pages 344–354. Springer, 1995.

47. D Lorge Parnas, Jan Madey, and Michal Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, 1994.

48. Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE software*, 19(1):18–25, 2002.

49. Michael G Hinchey and Stephen A Jarvis. *Concurrent Systems: Formal Development in CSP*. McGraw-Hill, Inc., 1995.

50. Janet Barnes, Roderick Chapman, Randy Johnson, James Widmaier, David Cooper, and B Everett. Engineering the Tokeneer enclave protection software. In *International Symposium on Secure Software Engineering (ISSSE)*, pages 1–9. IEEE, 2006.

51. David Copper and Janet Barnes. Tokeneer ID station EAL5 demonstrator: Summary report. Technical Report S.P1229.81.1, Altran Praxis Limited, 2008.

52. Tokeneer, 2020.

53. Yannick Moy and Angela Wallenburg. Tokeneer: Beyond formal program verification. *Embedded Real Time Software and Systems (ERTS)*, 24, 2010.

54. Reto Buerki and Adrian-Ken Rueegsegger. Muen – an x86/64 separation kernel for high assurance. Technical report, University of Applied Sciences Rapperswil (HSR), 2013.

55. Martin Stein. Spunky, a Genode kernel in Ada/SPARK. *Ada User Journal*, 41(2):99–102, 2020.

56. Roderick Chapman, Eric Botcazou, and Angela Wallenburg. SPARKSkein: A formal and fast reference implementation of Skein. In *Brazilian Symposium on Formal Methods*, pages 16–27. Springer, 2011.

57. Roderick Chapman and Yannick Moy. SPARK 2014 re-implementation of the TweetNaCl crypto library, 2020.

58. Tobias Reiher, Alexander Senier, Jeronimo Castrillon, and Thorsten Strufe. RecordFlux: Formal message specification and generation of verifiable binary parsers. In *International Conference on Formal Aspects of Component Software (FACS)*, pages 170–190. Springer, 2019.

59. Common weakness enumeration (cwe®), Aug 2020.

60. Common vulnerabilities and exposures (cve®), Nov 2020.

61. R Chapman and Y Moy. Adacore technologies for cyber security, May 2018.

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

63. K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 348–370. Springer, 2010.

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

65. 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.

66. 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.

67. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

68. SPARK 2014 user's guide. https://docs.adacore.com/spark2014-docs/html/ug/index.html, 2011-2021. Accessed: 2020-4-12.

69. Adacore and Thales. *Implementation Guidance for the Adoption of SPARK*. AdaCore, 2018.

70. Ryan Baity, Laura R Humphrey, and Kenneth Hopkinson. Formal verification of a merge sort algorithm in SPARK. In *AIAA Scitech 2021 Forum*, page 0039, 2021.

71. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html, 2006. Accessed: 2020-11-16.

72. John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, 1993.

73. Laura R Humphrey, James Hamil, and Joffrey Huguet. End-to-end verification of initial and transition properties of GR(1) designs in SPARK. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 60–76. Springer, 2020.

74. Sanjai Rayadurgam, John Komp, Lian Duan, Baek-Gyu Kim, Oleg Sokolsky, and Insup Lee. From requirements to code: Model based development of a medical cyber physical system. In *Software Engineering in Health Care: 4th International Symposium, FHIES 2014, and 6th International Workshop, SEHC 2014, Revised Selected Papers*, volume 9062, page 96. Springer, 2017.

75. Jing Liu, John D Backes, Darren Cofer, and Andrew Gacek. From design contracts to component requirements verification. In *NASA Formal Methods Symposium (NFM)*, pages 373–387. Springer, 2016.

76. M Anthony Aiello, Claire Dross, Patrick Rogers, Laura Humphrey, and James Hamil. Practical application of SPARK to OpenUxAS. In *International Symposium on Formal Methods (FM)*, pages 751–761. Springer, 2019.

77. César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, María Consiglio, and James Chamberlain. DAIDALUS: Detect and avoid alerting logic for unmanned systems. In *34th Digital Avionics Systems Conference (DASC)*, pages 5A1–1–5A1–12. IEEE/AIAA, 2015.

78. Joshua Brakensiek, Marijn Heule, John Mackey, and David Narváez. The resolution of keller's conjecture. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 48–65. Springer, 2020.

79. Jennifer A Davis, Laura R Humphrey, and Derek B Kingston. When human intuition fails: Using formal methods to find an error in the 'proof' of a multi-agent protocol. In *International Conference on Computer Aided Verification (CAV)*, pages 366–375. Springer, 2019.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
25–03–2021	Master's Thesis	Aug 2019 — Mar 2021

4. TITLE AND SUBTITLE

Formal Verification for High Assurance Software: A Case Study Using the SPARK Auto-Active Verification Toolset

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ryan M. Baity, 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-21-M-009

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

Software is an increasingly integral and sophisticated part of safety- and mission-critical systems. Poorly written software can lead to information leakage, undetected cyber breaches, and even human injury in cases where the software directly interfaces with components of a physical system. These systems may range from power facilities to remotely piloted aircraft. Software bugs and vulnerabilities can lead to severe economic hardships and loss of life in these domains. As fast as software spreads to automate many facets of our lives, it also grows in complexity. The complexity of software systems combined with the nature of the critical domains dependent on those systems results in a need to verify and validate the security and functional correctness of such software to a high level of assurance. The current generation of formal verification tools make it possible to write code with formal, machine-checked proofs of correctness. This thesis demonstrates the process of proving the correctness of code via a formal methods toolchain. It serves as a proof of concept for this powerful method of safety- and mission-critical software development.

15. SUBJECT TERMS

formal methods, SPARK, software verification, V&V, interpolation, merge sort, priority queue

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	124	19b. TELEPHONE NUMBER *(include area code)* 937-255-3636x4579; kenneth.hopkinson@afit.edu