Formal Verification of Prim's Algorithm in SPARK

Brian S. Wheelhouse
Air Force Institute of Technology
bswheelhouse@gmail.com

Laura Humphrey
Air Force Research Laboratory
laura.humphrey@us.af.mil

Kenneth M. Hopkinson
Air Force Institute of Technology
kenneth.hopkinson@afit.edu

Abstract

Many distributed systems use a minimum spanning tree (MST) as the backbone of efficient communication within the system. Given its critical role, it is important that the MST be implemented correctly. One way to ensure its correctness with a high degree of confidence is to use formal methods, i.e. mathematically-based tools and approaches for design and verification of software and hardware. Toward this end, we implement Prim's algorithm for construction of MSTs in SPARK, which is both a programming language and associated set of formal verification tools. At the most basic levels, formal verification in SPARK requires proving that code satisfies contracts on data flow and initialization and is free of run-time errors, which often reveals rare or subtle errors that are hard to detect through testing alone. Once errors are corrected and formal verification is complete, the result is code that is mathematically proven to satisfy the verified properties. In this paper, we provide background on SPARK and describe the process of using it to implement and verify basic properties of MSTs constructed using Prim's algorithm.

Keywords: formal methods, program verification, spanning trees

1. Introduction

As software systems become increasingly complex, it becomes increasingly difficult to ensure their correctness. A major reason is that as system complexity increases, the proportion of system behaviors that can be feasibly covered through standard test-based verification approaches decreases, leaving more room for latent errors. A possible solution to this problem lies in the use of formal methods, i.e. mathematically-based tools and approaches for software and hardware verification [Rushby, 1997, O'Regan, 2017]. Whereas testing checks individual execution traces of a system, formal methods analyze a mathematical model of a system, opening the possibility of mathematically proving that all possible behaviors of the system are correct. To make an analogy, consider the Pythagorean Theorem. One could merely build confidence in its correctness by testing it against a set of randomly selected right triangles, or one could prove its correctness for all right triangles by applying geometric axioms.

In safety-critical domains where errors can lead to substantial damage or loss of life, there is a need to eliminate as many errors as possible. Certification standards for many safety-critical domains therefore promote the use of formal methods, e.g. ISO 26262 for the automotive domain [Bahig and El-Kadi, 2017], EN 50128 for the railway domain [Basile et al., 2018], and the DO-333 supplement to DO-178C for the aerospace domain [RTCA, 2011]. There is a perception that formal methods require significant expertise to use and may not provide a good return on investment [Davis et al., 2013, Bishop et al., 2011, Nemathaga and van der Poll, 2020], so historically the use of formal methods has been concentrated in safety-critical domains. However, the challenge of maintaining software correctness in the face of growing complexity has recently motivated the use of formal methods in other domains. For example, engineers at Amazon Web Services have been using formal methods since 2011 to help solve difficult design problems in systems that use distributed algorithms for data management [Newcombe et al., 2015], and Amazon Web Services also uses formal methods to address a variety of cyber security concerns [Chudnov et al., 2018, Backes et al., 2018, Backes et al., 2019]. Despite perceptions about formal methods being difficult and expensive to use, Newcombe et al (2015) found that "Formal methods find bugs in system designs that cannot be found through any other technique we know of," and "Formal methods are surprisingly feasible for mainstream software development and give good return on investment." In fact, they found that using formal

URI: https://hdl.handle.net/10125/103443
978-0-9981331-6-4

methods to write and check proofs of certain types of algorithms was actually faster and easier than doing so by hand. In general, while the use of formal methods does require some investment, the return on investment is eventually realized in terms of better reliability, security, and fewer bugs to fix after development [Nemathaga and van der Poll, 2020].

For formal program verification, one language and toolset that we have found relatively easy to use is SPARK [McCormick and Chapin, 2015]. This is both because the design philosophy of the language emphasizes safe and correct programming, it is freely available as part of GNAT Community Edition [AdaCore, 2022a], and there are a number of educational materials and examples readily available [AdaCore, 2021, Garion, 2019]. In this paper, we show how to use SPARK to implement and verify certain properties of Prim's algorithm for building minimum spanning trees (MSTs). We choose to focus on this algorithm because MSTs are used in problems involving network reliability, classification, and routing [Mariano et al., 2013, van Steen and Tanenbaum, 2017] and would benefit from formal verification given the need for reliability in these problem domains. The rest of the paper proceeds as follows. Section 2 discusses applications of SPARK in industry as well as efforts to prove Prim's algorithm using various methods, Section 3 gives some background on SPARK, Section 4 demonstrates how SPARK is used to develop and verify Prim's algorithm, Section 5 addresses the results of the analysis report generated by SPARK, and Section 6 concludes the paper.

2. Related Work

Formal methods have been used successfully in a variety of large projects and in proving Prim's algorithm. [Chapman and Schanda, 2014] and [Humphrey et al., 2022] provide overviews of how SPARK has been used in various projects, especially projects focused on cyber security and safety in the aerospace domain. These include the Ship Helicopter Operating Limits Information System (SHOLIS); the C130J "Hercules" core mission computer, which saw an 80% savings in the modified condition/decision coverage (MC/DC) testing budget due to the low number of faults discovered during testing; and the NSA-funded Tokeneer demonstrator, for which testing found zero defects for a period after delivery. In terms of cyber security, verification at the silver level in SPARK proves that code is free of many of the cyber vulnerabilities classified in the MITRE corporation Common Weakness Enumeration (CWE) database [MITRE, 2021]. This is discussed in [Chapman and Moy, 2018], which provides a mapping of how the language features of SPARK/Ada prevent certain classes of CWEs and how verification with SPARK prevents others.

We briefly note that other tools and frameworks perform analogous types of formal verification for different languages. For example, Frama-C is a framework for analysis of C code in which contracts and assertions are written in the ANSI/ISO C Specification Language (ACSL) and plugins for formal verification are available [Cuoq et al., 2012]. A case study comparing ACSL/Frama-C with SPARK can be found in [Brito and Pinto, 2010]. There is also Prusti for Rust [Finkbeiner et al., 2020] and Krakatoa [Marché et al., 2004] for Java, just to name a few.

In regard to Prim's algorithm, some efforts have already been made to apply formal methods. Abrial, Cansell, and Méry give an approach to proving Prim's algorithm using the formal modeling tool Atelier B [Abrial et al., 2003]. Atelier B is an environment for generating and proving proof obligations for formal models, e.g. of algorithms. Such models can be automatically translated to C, C++, Ada, or HIA code [ClearSy, 2022], but since errors could be introduced during this translation, additional program verification tools such as SPARK should be used.

Another effort has succeeded in a proof of full functional correctness of an executable implementation of Prim's algorithm written in verifiable C using Coq: CompCert and the Verified Software Toolchain (VST) separation logic deductive verifier [Mohan et al., 2021]. Mohan demonstrates that Prim's algorithm works on disconnected graphs (thus finding a minimal spanning forest (MSF) rather than a MST) and predicts that more-automated tools such as Why3 would not be able to prove full functional correctness as easily as their work with VST.

Based on the denotational semantics of the language, SPARK translates programs along with checks and contracts to be verified to the Why3 deductive verification platform [Filliâtre and Paskevich, 2013]. 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 checks and contracts. Why3 then uses multiple theorem provers/satisfiability modulo theory (SMT) solvers to discharge the VCs, including CVC4 [Barrett et al., 2011], Alt-Ego [Conchon et al., 2018], and Z3 [de Moura and Bjørner, 2008]. While the tools attempt to automate this process, sometimes additional assertions in the code must be provided by the user to guide the underlying provers. We demonstrate what

SPARK is able to prove automatically using Prim's algorithm as an example. This example also serves as a simple tutorial on how to begin using SPARK to formally verify an executable implementation of a common algorithm.

3. SPARK

This section provides an overview of SPARK, much of which is summarized from [AdaCore, 2021]. SPARK is both a programming language *and* a formal verification toolset. SPARK as a programming language is based on the Ada programming language. Ada has a number of features that help support the development of safe and correct programs, which SPARK builds upon. However, SPARK both adds some features that support formal verification and removes some features that make formal verification difficult. To summarize, SPARK leverages features from Ada such as

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

and removes features such as

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

For users that rely on Ada features that are restricted in the SPARK subset, note that while SPARK can be used to prove an entire program, it can also applied to only specific parts of a program, including designated lines, subprograms, or packages. Combined with the fact that SPARK is compiled using an Ada compiler, this makes it possible to mix unproven Ada code with restricted features into the program if necessary. The SPARK User's Guide goes into more detail about these restrictions in [AdaCore, 2022d]. The relationship between SPARK and Ada is depicted in Figure 1.

As a static verification toolset, SPARK verifies code without compiling or executing it. SPARK performs several different types of static analysis. One is *flow analysis*, which checks the initialization of variables, unused assignments, unmodified variables, and data dependencies between inputs and outputs of subprograms. The other is *proof*, which checks for the absence of run-time errors as well as conformance of the program with its specifications.

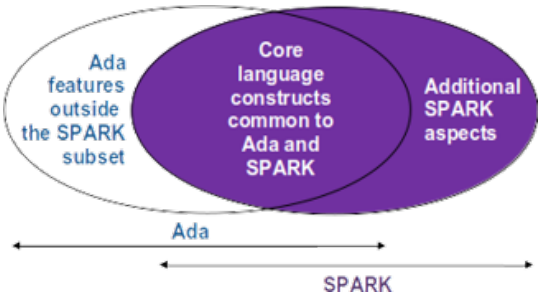

Figure 1. The relationship between SPARK and Ada [AdaCore, 2021].

Given the different types of analysis SPARK can perform and the fact that the level of detail in program specifications can vary from partial to complete, some colloquial terms have been adopted by SPARK users to define the level of assurance that has been attained for the code. In order from lowest to highest these levels are colloquially referred to as "stone," "bronze" "silver," "gold," and "platinum" [AdaCore, 2022b]. A simple example of a gold level proof can be found in [Baity et al., 2021], which verifies that a merge sort algorithm satisfies a partial specification. Table 1 gives more detail on the levels of SPARK verification and the use cases at each level.

Table 1. Levels of SPARK verification. From [AdaCore, 2022b]

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

4. Example

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

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

4.1. Prim's Algorithm

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

A min-priority queue is commonly used in Prim's algorithm to quickly extract the next minimum edge. For each vertex $v \in G.V$, let $G.Adj[v]$ be a list of adjacent vertices. For each vertex in the min-priority queue Q, let $v.key$ store the minimum weight of any edge connecting v to a vertex in the tree A (with $v.key = \infty$ if there is no edge), so that function $Extract\text{-}Min(Q)$ returns the vertex associated with the smallest weight. Let $v.\pi$ store the corresponding parent of v in the tree. Then Algorithm 1 outlines the steps in Prim's algorithm as given in [Cormen et al., 2022], with r being root of the generated tree A and the structure of A described by the values of $v.\pi$ extracted from extracted from from queue Q.

Algorithm 1 Prim's Algorithm [Cormen et al., 2022]

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

4.2. Prim's Algorithm in SPARK

The Ada language includes two types of subprograms: functions and procedures. A function is a subprogram that returns a value, while a procedure is a subprogram that does not. The implementation of Prim's algorithm presented in this paper uses two functions: one called $Extract\_Min$ and another called Mst_Prim. Rather than using a min-priority queue as shown in Algorithm 1, this implementation tracks the visited vertices with a type called $Visited\_Set$, which is an array of Booleans, and uses $Extract\_Min$ to find the next minimum edge that has not yet been visited. The declaration of these functions is given in Listing 1, and the implementations are given in Listing 2 and revised in Listing 4. This example restricts the graph size to only 5 vertices for simplicity.

Listing 1. Package specification mst_prim.ads

```
package MST_Prim with SPARK_Mode is
  Subtype Weight is Integer
    range 0 .. Integer'Last;
  Subtype Extended_Vertex is Integer
    range 0 .. 5;
  Subtype Vertex is Extended_Vertex
    range 1 .. 5;
  type Destinations is
    array(Vertex) of Vertex;
  type Weights_List is
    array (Vertex) of Weight;
  type MST is record
    Weights: Weights_List;
    Edges: Destinations;
  end record;
  type Visited_Set is
    array(Vertex) of Boolean;
```

```
  type Adj_List is
    array (Vertex) of Weight;
  type Graph is
    array (Vertex) of Adj_List;
  function Extract_Min
    (Weights: Weights_List;
     Visited: Visited_Set)
    return Vertex;
  function Mst_Prim
    (G: Graph;
     r: Vertex)
    return MST;
end MST_Prim;
```

Since SPARK is a subset of Ada, it follows the same packaging structure. The code is structured into two files: a specification file with an ".ads" file extension which contains function and parameter declarations for the package, and a package body file with an ".adb," file extension which contains the function implementations for the package. The following code is an implementation of Prim's algorithm in SPARK at the stone level, i.e. the code compiles but the results show that it does not pass flow analysis as shown in Listing 3.

Listing 2. Package body mst_prim.adb

```
package body MST_Prim
  with SPARK_Mode
is

  function Extract_Min
    (Weights: Weights_List;
     Visited: Visited_Set)
  return Vertex is
    min: Weight := Weight'Last;
    min_Index: Integer;
  begin
    min_Index := 1;
    for I in Vertex loop
      if Weights(I) < min
        and Visited(I) = False
      then
        min := Weights(I);
        min_Index := I;
      end if;
    end loop;
    return min_Index;
  end Extract_Min;

  function Mst_Prim
    (G: Graph; r: Vertex)
  return MST is
    M : MST;
    Visited: Visited_Set :=
             (others => False);
    u : Vertex;
  begin
    M.Weights(r) := 0;
    M.Edges(r) :=
      Extended_Vertex'First;
    for I in Vertex loop
      u := Extract_Min
           (M.Edges, Visited);
      Visited(u) := True;
      for V in Vertex loop
        if G(u)(V) > 0
          and Visited(V) = False
          and G(u)(V) < M.Weights(V)
        then
          M.Weights(V) := G(u)(V);
          M.Edges(V) := u;
        end if;
      end loop;
    end loop;
  return M;
  end MST_Prim;
end MST_Prim;
```

Listing 1 contains the package specification. In the specification file, we set up types to describe a graph and an MST. We represent a graph as an adjacency matrix that specifies weights for each edge, and we represent an MST as a record that contains a list of parents and a list of minimum edge weights, each list having one element per vertex. The list of minimum edges weights is stored using a type called $Weights\_List$, which is analogous to the set of $v.key$ values in Algorithm 1, and the list of parents is stored using a type called $Destinations$, which is analogous to the set of $v.\pi$ values. Each list is an array indexed by $Vertex$, so that each index into the array corresponds to a vertex in the graph. The $Vertex$ type in Listing 1 cannot hold any value less than 1 or greater than 5. This SPARK feature, which is inherited from Ada, helps with type safety. Strong types in SPARK help clarify the intent of the code and ensure that values are not corrupted by incompatible types during run-time. Type names are also case insensitive, which adds additional clarity to the code by enforcing that all type names must be unique. If the names are not unique, the code will not compile. For instance, one cannot name a variable "vertex" when there is already a type called "Vertex." Bounding a type ensures that out-of-bounds values cannot be assigned to variables of that type without a run-time error during execution. The SPARK verification tools can automatically prove

that variables of bounded types are never assigned out-of-bounds values as part of the *flow analysis* check.

Listing 2 contains the implementations of the functions for the algorithm. For a given initial vertex r, the MST_Prim function initializes the corresponding weight in MST M to zero and the edge to its parent as $Extended\_Vertex'First$ (which is zero) to represent NIL. At each iteration, the vertex that is reachable with minimum weight from the current tree stored in MST M is selected as the current vertex and is marked as visited in $Visit\_Set$. From here, the function follows the process as described in Algorithm 1 and updates the minimum weights to reach each vertex that the current vertex is connected to. Using $Extract\_Min$ the next reachable vertex with minimum edge weight is selected, and the process of updating the edge weights in the solution MST continues. As a new edge is added, a new partial solution MST is created containing all the vertices that have been visited so far.

Although the code in the Listing 2 is correct SPARK code (making this a stone-level verification), it does not pass SPARK flow analysis. Flow analysis verifies that the code satisfies checks on variables that model how the data flows through them at run-time [AdaCore, 2021]. In this case, M is not properly initialized, so analysis with SPARK provides the output in Listing 3, identifying the fields of the record M that are not properly initialized. Uninitialized variables introduce non-determinism which is evident when the program specified in Listing 2 is executed. Without initializing the variable M, the execution of the code results in an incorrect MST. To resolve this issue, we simply initialize M as shown in Listing 4, with minimum edge weights all set to the maximum possible value for type $Integer$ and all edges to parents set to 0 (representing NIL).

Listing 3. Console output after SPARK analysis of the code in Listing 1 and 2

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

Listing 4. Package body mst_prim.adb

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

Once M is initialized, flow analysis passes and the code constructs a correct MST. This raises the verification level to bronze, and with no run-time errors, it is even considered silver level. The automatic achievement of silver level verification in this case is mainly due to the simplicity of the code and the straightforward type definitions. The analysis report in the next section explains what checks were proved to achieve these levels of verification.

5. Results

SPARK generates an analysis report that summarizes the checks performed on the code during analysis, including whether or not the checks were successful and which tools or provers were used to discharge them. These details are presented as a table included in the analysis report as shown in Listing 5 and Listing 6. Each row in the table represents the categories of checks that SPARK performs and the columns represent which tool was used to discharge each check. When a prover is used to discharge a check, the name of the prover is cited in the provers column. The numbers in the table represent the total number checks verified by the associated tool. The number of steps needed to prove the checks and a breakdown of the checks by subprogram are also given in the analysis report. A thorough description of the analysis report including descriptions of the table columns and rows can be found in [AdaCore, 2022c] but the results for this example are presented here.

Listing 5 shows the analysis report for our program when it had an uninitialized variable. When a variable is uninitialized, flow analysis fails. The code analyzed in this case is contained in Listing 1 and Listing 2, where the MST variable in the MST_Prim subprogram is uninitialized. After running SPARK with the "prove all" option on this code, 4 checks (50% of all of the checks) are unproved as shown in the analysis report. These four checks are directly related to the warnings given in Listing 3. This means there is no guarantee that valid values will be passed through the variable M in the subprogram Mst_Prim. The "Unproved" column in Listing 5 demonstrates how flow analysis is

able to catch initialization issues that are not caught by the compiler. Initializing M as shown in Listing 4 resolves the error and the new analysis report, presented in Listing 6, shows that all initialization checks pass. Additionally, the code now passes flow analysis and is guaranteed to have valid information flow because flow analysis is sound, which means that if the errors it is supposed to catch are not caught, then there are no such errors [AdaCore, 2021].

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

Listing 5. SPARK analysis report with MST uninitialized

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

max steps used for successful proof: 1

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

Listing 6. SPARK analysis report with MST uninitialized

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

max steps used for successful proof: 1

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

Proofs performed by the provers are only guaranteed if flow analysis is passing. Therefore, the results in the "Provers" column should be considered only after flow analysis is complete. For more complex code or code with gold or platinum level functional specifications, it is likely that more effort would be required from the developer, e.g., additional annotations in the code needed to guide the provers (see [Baity et al., 2021] for an example).

Since all subprograms in this example had 0 unproved checks in all rows above "Functional contracts," the subprograms are verified to the silver level. We consider this an automatic proof because the code did not need to be annotated with pre- and post-conditions, loop invariants, assertions, etc. for all checks to prove. Raising this code to the gold or platinum level would require first writing functional specifications for each subprogram in the form of pre- and postconditions, then using SPARK to attempt to prove that they are satisfied. After that, if SPARK is not able to prove that the code satisfies the specifications automatically, then the developer would need to add additional annotations in the code to guide the provers. At the silver level, the code is currently guaranteed to have correct variable initialization and data flow and to be free of run-time errors, which is a key step in demonstrating that the code is highly reliable.

6. Conclusion

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

In the future, we would like to re-implement the algorithm using a formally verified implementation

of a min-priority queue such as the one presented in [Baity, 2021] to match Algorithm 1 more closely. Next, we would like to formally verify that our implementation satisfies functional specifications of the algorithm. This would require writing appropriate specifications for our subprograms in the form of pre- and postconditions that describe their desired behavior. While SPARK attempts to prove such properties automatically, fully automated proof is generally not feasible, so it is likely that annotations in the form of assertions and loop invariants will be needed to guide the provers. As a starting point, we can leverage work by Möller and Höfner who prove Prim's algorithm by hand (i.e. not with formal methods), but using a proof strategy explicitly designed to facilitate formal program verification of an implementation of the algorithm [Möller and Höfner, 2019]. Most other proof approaches rely on (1) existence of a minimal spanning tree of the overall graph and (2) properties that rely on reasoning about graph cycles, both of which are hard to reason about in program verification tools. Möller and Höfner's proof strategy establishes invariants at each step of the algorithm, which is far easier to reason about in program verification. In a SPARK implementation, these invariants likely provide the assertions and loop invariants needed to prove the code satisfies functional specifications. If we are able to complete a proof of full functional correctness using the approaches in [Möller and Höfner, 2019] and also provide enough additional functionality to form a library, we plan to make our code available as a crate through the new Alire (Ada LIbrary REpository) distribution system. Completing this proof will show that a full functional correctness proof of Prim's algorithm is possible with more-automated tools such as Why3, contrary to Mohan et al.'s prediction that such tools would not be able to prove full functional correctness as easily as their work with VST [Mohan et al., 2021].

Additional research can be done in comparing a variety of formal verification tools using other languages such as Rust with Prusti. The Rust language was built with many of the same security concerns in mind that resulted in the creation of Ada/SPARK. Recently, SPARK released support for a restricted form of pointers inspired by Rust [Dross and Kanig, 2020]. This new feature allows for the verification of recursive data structures, demonstrating that as the SPARK language grows, more challenging structures and algorithms can be formally verified using SPARK. Since both the Rust and SPARK languages were developed with similar safety and security concerns in mind, they will probably continue to influence each other. An interesting exercise with these languages would be to use the same specification to generate and verify code in each language to see how the tools, reliability, safety, and security compare. This exercise may give some insight to what the future of formal software verification may look like.

7. Acknowledgements

This paper was partially supported by AFOSR grant #20RQCOR096.

References

[Abrial et al., 2003] Abrial, J.-R., Cansell, D., and Méry, D. (2003). Formal derivation of spanning trees algorithms. In *International Conference of B and Z Users*, pages 457–476.

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

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

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

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

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

[Backes et al., 2019] Backes, J., Bayless, S., Cook, B., Dodge, C., Gacek, A., Hu, A. J., Kahsai, T., Kocik, B., Kotelnikov, E., Kukovec, J., et al. (2019). Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification (CAV)*, pages 231–241. Springer.

[Backes et al., 2018] Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., and Varming, C. (2018). Semantic-based automated reasoning for AWS access policies using SMT. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE.

[Bahig and El-Kadi, 2017] Bahig, G. and El-Kadi, A. (2017). Formal verification of automotive design in compliance with ISO 26262 design verification guidelines. *IEEE Access*, 5:4505–4516.

[Baity et al., 2021] Baity, R., Humphrey, L. R., and Hopkinson, K. (2021). Formal verification of a merge sort algorithm in SPARK. In *AIAA Scitech 2021 Forum*.

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

[Barrett et al., 2011] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and

Tinelli, C. (2011). CVC4. In *International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer.

[Basile et al., 2018] Basile, D., ter Beek, M. H., Fantechi, A., Gnesi, S., Mazzanti, F., Piattino, A., Trentini, D., and Ferrari, A. (2018). On the industrial uptake of formal methods in the railway domain. In *International Conference on Integrated Formal Methods (iFM)*, pages 20–29. Springer.

[Bishop et al., 2011] Bishop, M., Hay, B., and Nance, K. (2011). Applying formal methods informally. In *44th Hawaii International Conference on System Sciences (HICSS)*, pages 1–8. IEEE.

[Brito and Pinto, 2010] Brito, E. and Pinto, J. S. (2010). Program verification in SPARK and ACSL: A comparative case study. In *International Conference on Reliable Software Technologies*, pages 97–110.

[Chapman and Moy, 2018] Chapman, R. and Moy, Y. (2018). Cyber security. `https://www.adacore.com/uploads/books/pdf/AdaCore-Tech-Cyber-Security-web.pdf`.

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

[Chudnov et al., 2018] Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., et al. (2018). Continuous formal verification of Amazon s2n. In *International Conference on Computer Aided Verification (CAV)*, pages 430–446. Springer.

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

[Conchon et al., 2018] Conchon, S., Coquereau, A., Iguernlala, M., and Mebsout, A. (2018). Alt-Ergo 2.2. In *International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–11.

[Cormen et al., 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

[Cuoq et al., 2012] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-C. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 233–247. Springer.

[Davis et al., 2013] Davis, J. A., Clark, M., Cofer, D., Fifarek, A., Hinchman, J., Hoffman, J., Hulbert, B., Miller, S. P., and Wagner, L. (2013). Study on the barriers to the industrial adoption of formal methods. In *International Workshop Formal Methods for Industrial Critical Systems*, pages 63–77. Springer.

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

[Dross and Kanig, 2020] Dross, C. and Kanig, J. (2020). Recursive data structures in SPARK. In *International Conference on Computer Aided Verification (CAV)*, pages 178–189. Springer.

[Filliâtre and Paskevich, 2013] Filliâtre, J.-C. and Paskevich, A. (2013). Why3 – where programs meet provers. In *European Symposium on Programming (ESOP)*, pages 125–128. Springer.

[Finkbeiner et al., 2020] Finkbeiner, B., Oswald, S., Passing, N., and Schwenger, M. (2020). Verified Rust monitors for Lola specifications. In *International Conference on Runtime Verification*, pages 431–450. Springer.

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

[Humphrey et al., 2022] Humphrey, L., Baity, R., and Hopkinson, K. (2022). Formal verification of safety-critical software using SPARK. In *Aviation Cybersecurity: Foundations, Principles, and Applications*, pages 31–48. IET.

[Marché et al., 2004] Marché, C., Paulin-Mohring, C., and Urbain, X. (2004). The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1-2):89–106.

[Mariano et al., 2013] Mariano, A., Lee, D., Gerstlauer, A., and Chiou, D. (2013). Hardware and software implementations of Prim's algorithm for efficient minimum spanning tree computation. In *International Embedded Systems Symposium*, pages 151–158.

[McCormick and Chapin, 2015] McCormick, J. W. and Chapin, P. C. (2015). *Building high integrity applications with SPARK*. Cambridge University Press.

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

[Mohan et al., 2021] Mohan, A., Leow, W. X., and Hobor, A. (2021). Functional correctness of c implementations of dijkstra's, kruskal's, and prim's algorithms. In *International Conference on Computer Aided Verification*, pages 801–826. Springer.

[Möller and Höfner, 2019] Möller, B. and Höfner, P. (2019). A new correctness proof for Prim's algorithm. `http://www.Informatik.Uni-Augsburg.de`.

[Nemathaga and van der Poll, 2020] Nemathaga, A. and van der Poll, J. A. (2020). *Formal Methods Adoption in the Commercial World*. PhD thesis, University of South Africa.

[Newcombe et al., 2015] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73.

[O'Regan, 2017] O'Regan, G. (2017). *Concise guide to formal methods*. Springer.

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

[Rushby, 1997] Rushby, J. (1997). Formal methods and their role in the certification of critical systems. In *Safety and reliability of software based systems*, pages 1–42. Springer.

[van Steen and Tanenbaum, 2017] van Steen, M. and Tanenbaum, A. (2017). Distributed systems. `https://www.distributed-systems.net/index.php/books/ds3/`.