An Analysis of Implementing PVS in SPARK Ada

Ashlie B. Hocking
Dependable Computing
Charlottesville, VA, USA
ben.hocking@dependablecomputing.com

Jonathan C. Rowanhill
Dependable Computing
Charlottesville, VA, USA
jonathan.rowanhill@dependablecomputing.com

Ben L. Di Vito
Dependable Computing
Williamsburg, VA, USA
ben.divito@dependablecomputing.com

***Abstract*— SPARK Ada's support for proofs of correctness make the programming language ideal for implementing a PVS specification. Algorithmically implementing a PVS specification in SPARK Ada allows users to maintain the rigor of PVS in executable code. The goal of such an implementation is to maintain the validity of the proofs showing the specification implements formal requirements specified in PVS as theorems. This then shows the implementation also satisfies those formal requirements.**

We synthesized portions of NASA's DAIDALUS (Detect and AvoID Alerting Logic for Unmanned Systems) PVS specification into SPARK Ada. To provide confidence in the correspondence between the PVS specification and the SPARK Ada implementation, we designed a formal synthesis process. This process, while currently manual, allows us to have increased confidence that the properties proven to hold for the specification will continue to hold for the implementation.

Keywords—formal specification, program synthesis, formal verification

I. Introduction

Formal specifications are useful for explicitly defining what programs should accomplish. By formalizing the specification, high-level design flaws can be caught early in the design process, which reduces the overall cost of development and increases confidence in the final product. Methods for detecting flaws in a specification include proving the specification satisfies requirements and showing that the specification is consistent — e.g., if a specification for a function is supposed to return a non-zero number, can one prove that the specification is guaranteed to do so?

This formal specification is then used as guidance for implementing the specification in a programming language that compiles to an executable program. Without a process to ensure that the implementation properly mirrors the specification, some of the gains provided by formal specification are lost. While the overall design benefits are retained, confidence in the correctness that requirements are satisfied is weakened unless one has a means to verify that the implementation matches the specification. Mirroring the specification in the implementation should be a method for preserving this confidence, but there can be an impedance mismatch caused by dissimilarities between the specification language and the implementation language. Another source of impedance mismatch is the difference between the purpose of a specification and the purpose of an implementation, specifically that a specification provides information about *what* a function needs to accomplish, and an implementation provides details about *how* a function accomplishes a task.

A. PVS Specification

PVS (Prototype Verification System) is a verification system [1] providing both a formal specification language and tools used to evaluate specifications written in the language. PVS *theories* are files primarily containing *types*, *constants*, *functional specifications*, and *theorems*. Below is an example theory demonstrating each of those features:

```
example: THEORY
BEGIN
 alert: TYPE = below(3);
 NO_ALERT: alert = 0;
 max_alert(a1, a2: alert): alert =
  max(a1, a2);
 max_ge_args: THEOREM
  FORALL(a1, a2: alert):
   max_alert(a1, a2)      >= a1 AND
   max_alert(a1, a2) >= a2;
END example
```

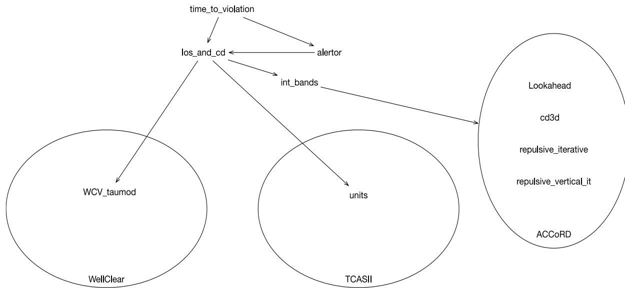
This example theory generates a type-correctness condition (TCC) for the `max_alert` specification that requires us to prove that the result will be of the correct type (i.e., that the value will be less than 3). One can verify both this TCC and the `max_ge_args` theorem by using the `(grind)` PVS proof strategy.

B. SPARK Ada

SPARK Ada is a subset of the Ada programming language containing those aspects of Ada that are amenable to formal verification [2]. When discussing SPARK Ada in this paper, we are referring to SPARK 2014, which is the latest version of the language. SPARK Ada provides the ability to prove contracts covering data dependencies, information flow, and functional specifications. The two features we leverage in this research are the ability to show freedom from exceptions (e.g., absence from overflow errors) and the ability to show that a postcondition of a function will be satisfied when the function's preconditions are satisfied.

Original work supporting AFRL RAST Program via IST Corporation

Fig. 1. PVS dependencies for time_to_violation functionality.



The SPARK Ada language was chosen due to a lower impedance mismatch between the purpose of writing an implementation in the language and the purpose writing a formal specification. Specifically, SPARK Ada provides the ability to specify *what* a function needs to accomplish as well as *how* the function will accomplish that task and then prove that the *how* guarantees the *what*.

Proving program correctness in SPARK Ada, both freedom from exception and adherence to formal contracts, requires discharging *verification conditions* (VCs). VCs are proof requirements that developers discharge to ensure correctness of the implementation. Most VCs can be discharged automatically by the SPARK Pro toolset. When VCs are not discharged automatically, the user has the option to either add annotations (typically via pragma Asserts) or to use a manual prover.

C. DAIDALUS

DAIDALUS (Detect and AvoID Alerting Logic for Unmanned Systems) contains a NASA PVS library specifying algorithms for detect-and-avoid system for UAS (unmanned aircraft systems) that need to operate in the NAS (national airspace system), based on the definition of a well-clear boundary defined at NASA [3][4]. The primary functionality specified by DAIDALUS are detection logic, maneuver guidance logic, and alerting logic.

This research focused on the functionality required to implement the PVS specification for time_to_violation, the PVS theory responsible for building a list of when an ownship UAS will violate the airspace of other aircraft in the NAS, with one element in the list for each aircraft that will have its airspace violated. The PVS dependencies for this functionality are shown in Fig. 1.

D. Retrenchment

In formal verification a *refinement* of function *A* to function *A'* occurs when the precondition of *A'* is no stronger than the precondition of *A* (i.e., *A'* can be used wherever *A* can be used) and the postcondition of *A'* is no weaker than the postcondition of *A* (i.e., the result of *A'* can be used wherever the result of *A* can be used).

In *retrenchment*, however, preconditions can be strengthened and postconditions can be weakened, although not in an arbitrary manner [5]. Retrenchment is necessary when synthesizing a SPARK Ada implementation from the DAIDALUS specification, because the PVS specification relies on ideal (or *divine*) reals and unbounded integers, which is typical of PVS specifications, rather than on IEEE floating-point numbers or 32-bit integers. In the work here, we take the retrenchment a step further, to tighten the bounds of acceptable inputs and outputs to real-world values. So, for example, altitudes are never allowed to exceed 100 km. This further retrenchment was used on the rationale that no additional reasoning is required to retrench from unbounded integers to 32-bit integers than to retrench from unbounded integers to more tightly bound integers, and the additional retrenchment provides additional assurance of correctness beyond even what is provided by the formal specification.

E. Goal Structuring Notation

Goal Structuring Notation (GSN) is a graphical notation for representing structured arguments [6]. GSN arguments typically begin with a top-level assurance claim node (e.g., “The SPARK Ada implementation of DAIDALUS has been correctly synthesized from the PVS specification.”) Claim nodes can have context (e.g., a definition of “correct synthesis”), justification, or assumption nodes attached to them. Claim nodes can be supported by strategy nodes which document how lower-level claim nodes or evidence nodes support the higher-level claim node. Evidence nodes are terminal nodes that describe evidence used to support higher-level claims. Examples of GSN arguments are provided later in this paper.

F. Iterative Assurance

Iterative assurance is a term we use to define an approach to assurance where a system progresses to the next stage of development (including possible release) prior to completing all possible assurance techniques at the current stage of development. An argument against iterative assurance is that catching design flaws early in the design process is less expensive than catching the design flaws later in the design process. To combat this expense, iterative assurance should be used sparingly and only when the benefits appear to clearly outweigh the costs. Additionally, the decision to postpone more rigorous assurance techniques should be supported by explicit, structured arguments (e.g., GSN) showing why the more rigorous assurance techniques are very unlikely to find the type of design flaws those techniques focus on. When resources become available to perform the more rigorous assurance, if flaws are discovered, the arguments supporting the claim that flaws were very unlikely to find design flaws should be analyzed for weaknesses.

II. Implementation Approach

A. Overview

The key features in the PVS specification to implement in SPARK Ada are theories, types, constants, and functions. The other primary considerations are the theorems and lemmas in PVS. In addition to implementing the desired components of DAIDALUS in SPARK Ada, we also implemented required components of the PVS prelude and the NASA PVS libraries.

B. Process

The process for synthesizing the SPARK Ada implementation follows the following steps:

1. Identify all dependencies.
   a. Examine the IMPORTING statements to determine which other PVS theories are referenced by the theory being implemented.
   b. Determine which types, constants, and functions from the imported theories are required. Typically, not all functionality will be required. Some imported theories will not be required at all by the functionality being implemented.
   c. Repeat this step for each IMPORTING statement in the imported theories.
2. Create a SPARK Ada package (.ads file and optional .adb file) for each PVS theory to synthesize. In the header of the package, document which PVS theory the package corresponds to.
3. Identify whether the theory uses generics, and if so, document name and type information for each generic parameter.
4. In the SPARK Ada packages, import the packages associated with the corresponding PVS imports using with and use statements. Instantiate generic packages with appropriate parameters where possible. Potential issues with importing generics are discussed later.
5. Identify types for retrenchment.
   a. Reals will typically translate to either single or double precision floating point numbers. Document the choice, including if the choice varies depending on context.
   b. Integers can translate to 32 or 64-bit signed integers. Document the choice.
   c. Naturals can translate to 32 or 64-bit unsigned integers or to signed integers with a range beginning at 0. Document the choice.
   d. Reals and integers in PVS have positive, negative, non-zero, non-positive, and non-negative variants. Non-positive and non-negative variants are created by limiting the range. Non-zero variants are created by using SPARK Ada type predicates, e.g., subtype nz_float is Float with Predicate => (nz_float /= 0.0); Positive and negative variants can be represented with type predicates or by limiting the range using the smallest possible float (Float'Small). For this work, we used the latter approach as that approach resulted in VCs that were easier to discharge.
   e. Determine whether to add additional constraints, e.g., limiting altitudes to 100 km.
6. Implement types, using either retrenchment where necessary as discussed in the previous steps.
   a. Enumerations can be copied almost verbatim, with minor syntactical changes . E.g., the definition Region: TYPE = {NONE, FAR, MID, NEAR, RECOVERY, UNKNOWN} in PVS can be written in SPARK Ada as type Region is (NONE, FAR, MID, NEAR, RECOVERY, UNKNOWN); .
   b. For records with named fields, the synthesis of SPARK Ada is very similar to the PVS, so that Spread: TYPE = [# lo:real, hi:real #] in PVS can be written in SPARK Ada as type Spread is record lo: Float; hi: Float; end record; .
   c. For records with unnamed fields in PVS (or tuples), field names must be generated for the SPARK Ada analog, as tuples are not defined for SPARK Ada. So, the PVS type definition Spread: TYPE = [real, real] is synthesized with generated field names in SPARK Ada as type Spread is record f1: Float; f2: Float; end record;.
   d. Predicated type declarations in PVS become predicated type declarations in SPARK Ada, so the predicated type definition in PVS nzreal: TYPE = {r: real | r /= 0} becomes in SPARK Ada the type definition subtype nz_float is Float with Predicate => (nz_float /= 0.0);.
7. Implement constants, using the type definitions previously discussed.
8. Implement the functions. For most cases, this synthesis is fairly straightforward, for example, the function sq(a): nonneg_real = a*a in PVS becomes function sq(a: Float) return nn_float is (a*a) in SPARK Ada. Some non-straightforward cases are discussed later.

C. Theorems and Postconditions

In general, the theorems in PVS do not need to be recreated in SPARK Ada, under the argument that if the theorems have been proven in PVS over the functions specified in PVS, there is no need to reprove the theorems in SPARK Ada assuming those functions have been correctly implemented. The emphasis is on showing that the PVS functions have been successfully implemented in SPARK Ada rather than on reproving the PVS theorems. Note, however, that due to the retrenchment of some types, this argument is not without flaws. An approach to iterative assurance could therefore later attempt to reproduce the PVS theorems in SPARK Ada in order to further increase confidence, though this approach will not be discussed further here.

Postconditions in PVS are indistinguishable from type definitions in that any postcondition must be part of the description of the return type. For example, in the PVS prelude, the function max is defined by:

```
max(m, n): {p: real | p >= m AND p >= n} =
IF m < n THEN n ELSE m ENDIF
```

Original work supporting AFRL RAST Program via IST Corporation

In this case, the return type is a predicated type that is a real number greater than both *m* and *n*. In SPARK Ada we can write this as:

```
function max(m, n: float) return float is
(if (m < n) then n else m)
  with Post => max'Result >= m and
       max'Result >= n;
```

In this implementation, we have represented the return type as simply float, and used the post-condition to predicate the return type in a manner matching the PVS predicate. One could argue that this post-condition is unnecessary using the same argument used for arguing against reproducing theorems. Here we elected to reproduce the implicit postconditions explicitly on the rationale that doing so increases confidence in the implementation, is typically easier to prove in SPARK Ada than the theorems we are not reproducing, and can help us to discharge VCs elsewhere in the implementation.

D. Issues

We have identified a total of thirteen issues that need to be addressed when synthesizing SPARK Ada from PVS:

1. Issue: PVS is case-sensitive, but SPARK Ada is not, so reproducing names exactly can lead to collisions.
    a. Related sub-issue: PVS allows duplicate names in contexts that SPARK Ada does not. (Note that the fourth issue in this list is specifically for theory/type collisions.)
    b. Resolution: rename variables/functions/types that have a naming collision and track the renaming in a table or database. For example, see Table 1, where we have eliminated a usage column for conciseness.

Table 1. Table tracking renaming due to name collisions.

1st case	Usage	2nd case	Package	Renamed to
TCOA	generic	tcoa	timevars	tcoa_fn
Sign	type	sign	signs	sign_fn
region	field	Region	alertor	RegionEnum
WCV_taumod	theory	WCV_taumod	WCV_taumod	WCV_taumod_fn
WCVTable	type	wcvtable	alertor	wcvtbl

2. Issue: Some PVS functions pass functions as arguments, but SPARK Ada does not support this.
    a. Resolution: track instantiations where we would pass functions and invoke the passed-in function directly where it would otherwise be called. For example, see Table 2.

Table 2. Table tracking passed functions.

Calling theory	Calling function	Called theory	Called fn	Passed function
horizontal_wcv _taumod	horizontal_wc v	horizontal_wcv	horizontal _WCV	taumod
wcv_taumod	WCV_taumod	wcv	WCV	taumod
wcv	WCV	horizontal_wcv	horizontal _WCV	taumod (from wcv_taumod)
wcv_taumod	WCV_taumod _interval	wcv	WCV_inte rval	horizontal_WCV_t aumod_interval

3. Issue: SPARK Ada will not let us pass functions as arguments (same as prior issue) which can lead to circular references to the functions when we use them explicitly.
    a. Resolution: put functions that would cause circular package references in packages that do not have this problem and track the movement in a table. For example, see Table 3.

Table 3. Table tracking function movement and renaming.

Original pkg	Function	New package	New fn
horizontal_WCV_ taumod	taumod	taumod	taumod
horizontal_WCV_ taumod	horizontal_WCV_ taumod_interval	horizontal_WCV_ taumod_interval	interval

4. Issue: SPARK Ada does not (easily) let package names and types be the same. See also the first issue.
    a. Resolution: pluralize package names or otherwise rename package names to avoid this issue and track the renaming in a table. For example, see Table 4.

Table 4. Table tracking package renaming.

PVS Theory	SPARK Ada Package
lookahead	lookaheads
sign	signs

5. Issue: PVS generics can have predicates attached to them, but SPARK Ada generics do not allow predicates.
    a. Resolution: use pragma Assert in package specification and track the asserts in a table for later review. For example, see Table 5.

Table 5. Table tracking generic predicate to Assert conversion.

SPARK Ada Package	Generic name	Predicate/Assert
tcasra2d	T	T >= B
WCV_interval	T	B < T
EntryExit_interval	T	T > B

6. Issue: PVS has automatic conversions, but SPARK Ada does not support automatic conversions.
    a. Resolution: create and track use of explicit conversions.

Table 6. Table tracking implicit to explicit conversion changes.

Implicit conversion	Explicit conversion	Where used
Vect2 (from Vect3)	vect3_to_2	wcv.WCV

7. Issue: SPARK Ada and PVS do not have the same reserved words.
    a. Resolution: track how the PVS variables that are reserved words in SPARK Ada are modified. For example, see Table 7.

Table 7. Table tracking keyword collisions.

Keyword	Location	Modification
entry	EntryExit record	entry_l
exit	EntryExit record	exit_l
delta	horizontal function name	delta_fn

8. Issue: Return types in SPARK Ada cannot use generic packages instantiated with arguments passed to the function, but PVS allows this.

    a. Related sub-issue: due to this issue, some other functions will have to be moved to the created Ada package to break circular references

    b. Resolution: create a generic package devoted to the function and track the packages in a table, e.g., Table 8.

Table 8. Table tracking package creation to handle passed generics. Asterisks (*) indicate a SPARK Ada function representing 2 different PVS functions.

PVS Theory	Function	Ada package	Fn
horizontal_WCV_taumod	horizontal_WCV_taumod_interval	horizontal_WCV_taumod_interval	interval
wcv	WCV_interval	WCV_interval	Interval*
wcv_taumod	WCV_taumod_interval	WCV_interval	Interval*
vertical_WCV	vertical_WCV_interval	vertical_WCV_interval	interval

9. Issue: PVS has something like anonymous records (i.e., tuples) that allow for access via `[#] where [#] is a number, but SPARK Ada records require fields to begin with a letter.

    a. Resolution: create named records in SPARK Ada that begin with `f`. For nested anonymous records, repeat the `f` when nesting. Track these named records in a table, for example Table 9.

Table 9. Table tracking record field name creations.

PVS Definition	SPARK Ada Definition
TTVL: TYPE = list[[Aircraft, [real,real], bool]]	type TTV is record f1: Aircraft; f2f1: Float; f2f2: Float; f3: Boolean; end record; package TTV_Container is new Formal_Doubly_Linked_Lists(TTV); subtype TTVL is TTV_Container.List;
Spread : TYPE = [real, real]	type Spread is record f1: Float; f2: Float; end record;

Alertor : TYPE = {al: [nat, list[AlertTable]] | (null?[AlertTable](al`2) AND al`1=0) OR (al`1>0 AND al`1 <=length(al`2))}	package AlertTable_Container is new Formal_Doubly_Linked_Lists(AlertTable);
	subtype AlertTableList is AlertTable_Container.List;
	type Alertor(capacity: Count_Type) is record
	f1: nat; f2: AlertTableList(capacity);
	end record with
	Predicate => (f1 <= nat(AlertTable_Container.Length(f2)));

10. Issue: PVS supports curried functions and more broadly functions that return other functions that can be evaluated later. SPARK Ada does not support the ability to return a function.

    a. Resolution: rather than return a function, add the arguments that the function would take to this function. Track the changes in a table, e.g., Table 10.

Table 10. Table tracking curried functions.

Function	Primary variables	Curried variables	Theory/ Package
table_to_ConfDet	wcvtable	B, T, aco, aci	alertor

11. Issue: Packages have to be defined at the top of a function, as opposed to in the middle of a function, so VCs that are generated by generics while defining the package do not have sufficient context to be discharged. For example, imagine a generic in a PVS theory with the precondition that it must be greater than zero, and this generic is defined in an IF condition checking that the variable is greater than zero, then in PVS it is trivial to prove the precondition is met. However, in trying to implement this in SPARK Ada, the Ada package must be defined at the top of the function where the variable could be less than zero, resulting in a VC that cannot be discharged.

    a. Resolution: create a helper function that can then define the package in the proper context. Track helper functions in a table, for example Table 11.

Table 11. Table tracking alternate variables used for satisfying generic preconditions.

Ada Package	Function	Referenced Package	VC	Use Location	Alternate Definition
los_and_cd	wcv_taumod_cd	WCV_interval	B < T	else for "if B >= T"	if B>=T then B + 0.1 else T
wcv	WCV_detection	WCV_interval	B < T	else for "if B = T" and "elsif B >T"	if B>=T then B + 0.1 else T

12. Issue: Functions for generic packages cannot be called until packages are instantiated. When generic subpackages are instantiated in the body of a package, constraint functions are not available for pre-conditions.

    a. Resolution: recreate constraint functions, with added parameters for generic variables of generic packages. Track constraint functions in a table, e.g., Table 12.

Table 12. Table tracking recreated constraint functions.

Ada Package	Generic Subpackage	Constraint	Note
vertical_wcv_interval	vertical	Theta_H_pre	
los_and_cd	wcv	WCV_pre	From wcv_taumod
los_and_cd	horizontal_wcv	horizontal_WCV_pre	From wcv (via wcv_taumod)
los_and_cd	taumod	taumod_pre	From horizontal_wcv (via wcv)
los_and_cd	tcas_tau	tau_mod_def_pre	From taumod (via horizontal_wcv)
los_and_cd	vertical_wcv	vertical_WCV_pre	From wcv (via wcv_taumod)

13. Issue: SPARK Ada uses IEEE floating point numbers and PVS uses unbounded, infinite-precision reals.

    a. Resolution: create arguments about retrenchment being valid.

A theme running through the resolutions for these issues is the need to track the changes required to resolve the issues.

Tracking is required so that the correctness of the synthesis can be more easily analyzed by reviewers.

III. Verification Conditions

As discussed previously, *verification conditions* (VCs) are proof requirements that developers discharge to ensure correctness of the implementation. Discharging VCs guarantees both freedom from exception (e.g., division by zero) and adherence to formal contracts. Most VCs can be discharged automatically by the SPARK Pro toolset. When VCs are not discharged automatically, the two reasons are that the code is either not correct or the prover is not able to find a proof that the code is correct. In the former case, additional guards are usually sufficient to render the code provably correct, mainly by strengthening the pre-conditions. In the latter case, the user has the option to either add annotations (typically via `pragma Asserts`) or to use a manual prover. Adding annotations is more robust to later changes in the implementation and is the approach taken here.

A total of 1,257 VCs were generated for the synthesized SPARK Ada code. We used a combination of automatic provers to discharge the VCs, specifically CVC4, Z3, and altergo. These provers and the built-in flow and interval provers were able to discharge more than 90% of the VCs being discharged automatically without the addition of annotations. Most of the remaining VCs were due to code that could lead to exceptions (floating-point overflows) and were discharged by strengthening preconditions. All but six of the remaining VCs were discharged by the addition of annotations. Rather than spending extra resources discharging the final six VCs we created structured arguments that the code corresponding to the VCs was correct despite the inability to rigorously prove the code correct by discharging the VCs.

IV. Argumentation

While ideally all VCs would be discharged prior to using a SPARK Ada program, there are situations that arise where limited resources might be better spent elsewhere, or time constraints might require releasing the software before all VCs have been discharged. An approach to iterative assurance is to postpone the discharge of VCs that are difficult to discharge, if a convincing argument can be written that the code responsible for generating the VC is correct. When combining SPARK Ada with Ada libraries that do not use SPARK contracts (e.g., `sqrt` from `Ada.Numerics.Elementary_Functions`), such an approach might be necessary to allow research into how to best discharge such VCs.

Note that one should always be able to discharge a VC by using a `pragma Assume`, but that approach is essentially the same as telling the prover to ignore the VC. Any such approach should also be supported by a structured argument explaining why the `pragma Assume` is valid. Another approach to discharging VCs that appear to be otherwise intractable is to define axioms in SPARK Ada, either externally (through the use of `External_Axiomization`), or by creating dummy functions where the postcondition of the function is the axiom one wishes to introduce, and is assumed in the body of the function with a `pragma Assume`. Introducing axioms should also be accompanied by arguments about why the axioms are valid.

An example argument, consider the following VC generated by SPARK Pro:

```
los_and_cd.adb:37:64: medium: precondition might fail, in instantiation at
alertor.ads:28 (e.g. when B = 0.0 and T = 0.0 and entry_exit = (entry_l =>
0.0, exit_l => 0.0))
```

The argument associated with this VC is shown in Fig. 2 and Fig. 3. In Fig. 2, the argument is made that the precondition is satisfied by showing that:

1. the precondition for the function at line 37 (`wcv_taumod_cd_B_lt_T`) has no surprises,
2. the precondition for `wcv_taumod_cd_B_lt_T` satisfies the precondition for the function being invoked (details of this claim are shown in Fig. 3), and
3. the precondition for `wcv_taumod_cd_B_lt_T` invalidates the counterexample reported by SPARK Pro.

While the second claim implies the third claim, a reported counterexample (as provided in the VC with the statement `when B = 0.0 and T = 0.0 and entry_exit = (entry_l => 0.0, exit_l => 0.0)`) is treated as a potential defeater to the argument and argue why the claims would be invalid if the counterexample were true. The evidence supporting the third claim is that the precondition that `B < T` prevents `B` and `T` from both being equal to `0.0`.

In Fig. 3, we argue that the precondition for `wcv_taumod_cd_B_lt_T` implies satisfaction of the precondition requirement at line 37, specifically that the precondition `wcv_interval_pre`, which is defined in `wcv_interval.ads`, as instantiated by line 31 in `los_and_cd.adb`, and invoked with values `aci.s-aco.s` for `s`, and `aci.v-aco.v` for `v` is satisfied. We argue that we have identified the precondition that needs to be satisfied correctly, and that the precondition `wcv_interval_pre` is explicitly satisfied by the precondition of `wcv_taumod_cd_B_lt_T`.

Fig. 2 Top half of the argument that precondition on line 37 in los_and_cd.adb is satisfied.

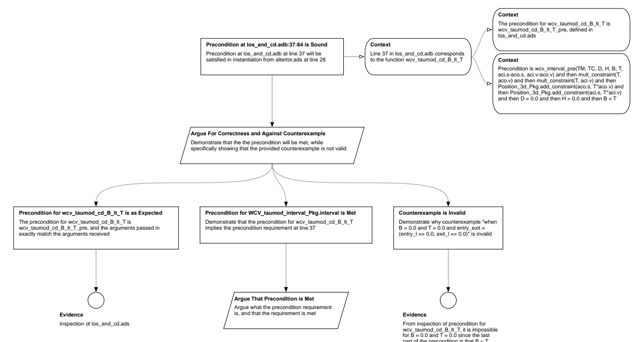
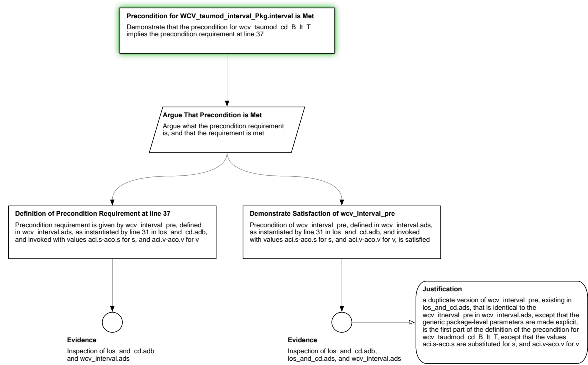


Fig. 3. Remaining argument that precondition on line 37 in los_and_cd.adb is satisfied.



While this argument seems simple enough that one would expect SPARK Pro to be able to discharge the VC the argument refers to, the line in question is instantiating a generic SPARK Ada package and we have found that VCs involving generics seem harder for the proof tool to discharge.

V. Related Work

The “mirrored implementation phase” of MINERVA [7] is similar to the approach we take here, except that they focus the mirroring on “interfaces to the algorithm specifications” rather than on the details of the specification. Our approach mirrors both the interface and the specification, but also either requires specifications that are implementable or that have implementable alternative formulations that have been proven to be identical. Instead of relying on the implementation details matching the specification details, MINERVA uses numerical evaluation to compare the results of the mirrored implementation to the PVS specification. While our work focuses on matching implementation details to the specification details as closely as possible, numerical evaluation should also be applied for additional verification.

VI. Conclusion and Future Work

A structured approach to synthesizing SPARK Ada code from a PVS specification allows for confidence in the correctness of the SPARK Ada implementation relative to the specification. This confidence is increased by using the SPARK Pro toolset to discharge VCs, demonstrating freedom from exception and adherence to formal contracts. Where the VCs cannot be readily discharged, we supplement the approach with structured arguments to maintain confidence in the correctness of the implementation.

In attempting to translate from PVS to SPARK Ada as faithfully as possible, one thing we encountered was the difficulty in discharging VCs associated with generics. In the PVS specification for DAIDALUS, generics were heavily used which impeded our ability to discharge VCs and were responsible for 4 of our 13 identified issues in the synthesis of SPARK Ada code from PVS. Internally, when we create PVS specifications we now tend to avoid generics, in part due to the difficulty they can create in synthesizing SPARK Ada code from them. However, when relying on external PVS specifications (such as for DAIDALUS), we do not want to alter the PVS specification, so alternative approaches to dealing with generics will be sought.

Acknowledgment

This work was funded by USAF AFRL/RQQA contract FA8650-17-F-2220. Approved for Public Release: Distribution Unlimited (Case Number: 88ABW-2017-0496).

References

[1] S. Owre, S. Rajan, J. M., Rushby, N. Shankar, and M. Srivas, “PVS: Combining specification, proof checking, and model checking,” Proc. 8th CAV, pp. 411-414, July 1996.

[2] R. Chapman and F. Schanda. "Are we there yet? 20 years of industrial theorem proving with SPARK," ITP: LNCS, pp. 17-26, 2014.

[3] C. Muñoz, A. Narkawicz, G. Hagen, J. Upchurch, A. Dutle, M. Consiglio, and J. Chamberlain, “DAIDALUS: detect and avoid alerting logic for unmanned systems,” DASC 2015, September 2015.

[4] S. Balachandran, C. A. Muñoz, M. C. Consiglio, M. A. Feliú, and A. V. Patel, “Independent configurable architecture for reliable operation of unmanned systems with distributed onboard services,” DASC 2018, September 2018.

[5] R. Banach and M. Poppleton, “Retrenchment: An engineering variation on refinement,” International Conf. of B Users, pp. 129-147, April 1998.

[6] T. Kelly and R. Weaver, “The goal structuring notation–a safety argument notation,” Proc. of DSN 2004, July 2004.

[7] A. Narkawicz, C. Muñoz, and A. Dutle, “The MINERVA software development process,” 6th Workshop on Automated Formal Methods, AFM 2017, May 2017.