L7.2.2 Deliverable - Final report on experimentations on industrial case studies

Release 1.0

Hi-Lite partners

May 31, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

The tools developed in the Hi-Lite project have been specified by all the partners of the project (academic, tool developers and end users). These requirements are presented in the deliverable L2.1.1 “Recommendations for the definition of the SPARK 2014 and E-ACSL languages” of the task 2.1 “Requirements”.

This document, written in the scope of the task 7.2 “Application of the tools to industrial case studies”, has the objective to validate the implementation of these tools in realistic contexts.

Three case studies, representative of various different domains of application, have been defined:

1. Application of the Hi-Lite tools on an Astrium case study (space domain)
2. Application of the Hi-Lite tools on various SPARK examples and Tokeneer (safety and security domains)
3. Application and integration of the tools in a middleware generation approach for component oriented critical software (Thales)

For each case study, this document:

- Describes the main features of the case study and
- Concludes on the interests of the Hi-Lite approach in the case study context.

This document is the final version of the preliminary report “Preliminary report on experimentations on industrial case studies” (deliverable L7.2.1).

CHAPTER

TWO

ANALYSIS OF SPACE SOFTWARE

This part of the document provides an assessment of the results of the Hi-Lite project thanks to a case study developed by Astrium Space Transportation. It allows concluding on the applicability of formal proof on space software and more generally on critical real-time embedded software.

The following sections describe:

1. The objectives of this part of the document and the approach followed to reach them.
2. The case studies developed by Astrium Space Transportation.
3. An assessment of the Hi-Lite project on these case studies.

2.1 Recalls on the objectives and on the followed approach

According to the applicable standards in the European space domain (ECSS / European Cooperation for Space Standardization), the developers of critical real time embedded software shall document the detailed design of the software and the corresponding test procedures. The objective of the Hi-Lite project is to support the detailed design and the unitary tests by providing:

- Means to formalize the detailed design
- Tools to improve the unitary tests.

The objectives of each software subprogram of the use case developed by Astrium Space Transportation are first described in an informal way by a textual description and then formalised by a contract defining:

- In which conditions the subprogram can be called (pre-condition)
- The expected behaviour of the subprogram (post-condition). In order to facilitate the following step (description of the test cases), the post-condition shall be as much as possible partitioned in cases.

In order to follow a classical test driven development, the test cases associated to the subprograms are informally described. The definition of the test cases shall be compatible with the definition of cases performed during the detailed design.

- The test cases are formalised in SPARK 2014
- The review of the detailed design and of the test cases (both formalised in SPARK 2014) shall be part of the Detailed Design Review (DDR) and of the Test Readiness Review (TRR)
- Each test case shall be either formally proved or tested.
- Test procedures are defined

The following coverages shall be ensured:

- Coverage between the software requirements and the detailed design
- Coverage between the detailed design and the test procedures

2.2 Astrium Space Transportation case studies

The case study of Astrium Space Transportation comes naturally from the space domain. A typical space flight program is roughly speaking made up of four parts:

- Low level software, operating system, drivers. This kind of software is out of scope of Hi-Lite.
- Data management: A typical example of such kind of software is the management of telemetry and telecommand. After analysis, it has appeared that Hi-Lite was not adapted to this kind of software, mainly concerned by encoding and decoding of data.
- Flight control or more generally numerical control / command algorithm
- Mission and Vehicle Management

The Astrium case study in Hi-Lite covers the last two aspects.

2.2.1 Numerical control / command algorithms

Description of the case study

Numerical control / command algorithms take as inputs floating point values, perform some numerical computations (with the classical basic mathematical operators such as additions, subtractions, multiplications, divisions, absolute values, trigonometry or operations on vectors and arrays, etc.) and return floating point results. Such algorithms have generally a retroaction loop, i.e. internal states.

It is generally not possible to define interesting functional contracts for such code. Indeed, the functional contract of the equation:

```
X := A * Y + Cos(Z)
```

is just itself (i.e. it is not possible to specify in a more abstract way this equation). Then, instead of defining functional contracts, this case study has the non functional objectives of proving the absence of run-time errors (such as division by zero) and the correctness of variable ranges (such as, for instance, a velocity shall always be between 0 and 25 km/s).

The Astrium case study implements as numerical control / command algorithms an example of solar wing management (for a spacecraft such as the ATV / Automated Transfer Vehicle). This piece of software is responsible:

- For the deployment of the solar wings (by cutting hardware links stowing the solar wings)
- For the orientation of the solar wings in order to optimize the received solar energy

The solar wing deployment has been modelled in SCADE and the SPARK compatible code has been automatically generated with the Qualified Code Generator of SCADE (KCG Ada). The objective of this demonstration is to assess the capability of the Hi-Lite toolset to analyze automatically generated code.

Example of generated code from a SCADE model:

```
if(Ctx.init_5) then
  SM_FSM_state_sel_1_1 := Kcg_Types.SSM_st_SURVIVAL;
  SM_AP_state_sel_1_1 := Kcg_Types.SSM_st_INIT_2;
  SC_t1_1_1 := P5_SGS_LIB_TYPES.IMPORTED_CONSTANTS.C_D_SC;
  last_RT_INHIBIT_TABLE_1_1 :=
    P5_SGS_LIB_TYPES.MAIN.C_D_SGS_RT_INHIBIT_TABLE;
  last_RT_INHIBIT_EQPT_1_1 := P5_SGS_LIB_TYPES.MAIN.C_D_RT_INHIBIT_EQPT;
  last_ALARM_ID_1_1 := Kcg_Types.NO_ALARM;
else
  SM_FSM_state_sel_1_1 := Ctx.SM_FSM_state_nxt_1_1;
  SM_AP_state_sel_1_1 := Ctx.SM_AP_state_nxt_1_1;
  SC_t1_1_1 := Ctx.rem_SC;
  last_RT_INHIBIT_TABLE_1_1 := Ctx.RT_INHIBIT_TABLE_1_1;
  last_RT_INHIBIT_EQPT_1_1 := Ctx.RT_INHIBIT_EQPT_1_1;
```

```
  last_ALARM_ID_1_1 := Ctx.ALARM_ID_1_1;
end if;
```

The solar wing rotation has been manually coded. The objective of this demonstration is to assess the capability of the Hi-Lite toolset to analyze manual algorithmic code. This piece of code uses a mathematical library which implementation is not fully in SPARK 2014, implying that it could not be formally proved, but only tested. However, the interface of this mathematical library is in SPARK 2014. The contracts defined on the mathematical library can then be used to prove the application code.

Example of contract in the mathematical library:

```
function Sin32 (X : T_Float32) return T_Float32
with
  Pre => ( X >= - C_2Pi32 ) and then
  ( X <= C_2Pi32 ),
  Post => ( Sin32'Result >= -1.0 ) and then
  ( Sin32'Result <= 1.0),
  Test_Case => (Name     => "-2Pi .. -3Pi/2",
                Mode     => Nominal,
                Requires => X >= -C_2Pi32 and X < -C_3Halfpi32,
                Ensures  => Sin32'Result >= 0.0 and Sin32'Result <= 1.0),
  Test_Case => (Name     => "-3Pi/2 .. -Pi",
                Mode     => Nominal,
                Requires => X >= -C_3Halfpi32 and X < -C_Pi,
                Ensures  => Sin32'Result >= 0.0 and Sin32'Result <= 1.0),
  ...
```

The contracts defined in the application code are then only related to the ranges of variables.

Example of contract in the application code:

```
subtype T_Angle_180 is T_Float32 range -180 .. 180;

function Normalise (X : in T_Float32) return T_Angle_180
with Pre =>
  ( X >= -720.0 ) and
     ( X <= 720.0 );
```

Verification and validation of this case study

All the contracts have been checked by dynamic testing. This phase is quite classical, except for the fact that the testing includes the preconditions and the postconditions defined in the case study. Then, gnatprove has been applied.

This case study contains two parts:

- The Mathematical_Library
- The Numerical_Algorithms

Results for "Mathematical_Library"

The analysis has been performed in 233 seconds (0 h 3 mn 53 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	15
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
division_check	3	3	100	0	0
overflow_check	9	9	100	0	0
postcondition	4	3	75	1	25
precondition	1	1	100	0	0
range_check	10	9	90	1	10
Total	27	25	92	2	7

Analysis of the non proved VCs:

Some algoritmic functions are not completely known by gnatprove.

Example:

```
function Arctan (X : T_Float32) return T_Float32
with
  Post => (Arctan'Result >= -C_Halfpi32)
          and then (Arctan'Result <= C_Halfpi32);

function Arctan (X : T_Float32) return T_Float32
is (Num32.Arctan (X));
```

The postcondition is not proved by gnatprove.

The exact behaviour of algorithmic functions depending of the implementation, this behaviour is acceptable. Algorithmic functions and their contracts are preferably validated by intensive testing.

Results for "Numerical_Algorithms"

The analysis has been performed in 653 seconds (0 h 10 mn 53 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	30
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	42	42	100	0	0
division_check	9	9	100	0	0
index_check	2	2	100	0	0
overflow_check	95	95	100	0	0
postcondition	2	2	100	0	0
precondition	34	34	100	0	0
range_check	81	78	96	3	3
Total	265	262	98	3	1

Analysis of the non proved VCs:

gnatprove has some difficulties to take into account rounding. In the following example, the second assertion is not proved.

Example:

```
function Round_Closest (X : T_Float32) return T_Float32
is (T_Float32'Rounding (X));

pragma Assert(( Y >= -11160.002 ) and (Y <= 11160.002 ));
Round_Y := Ml.Round_Closest (Y);
pragma Assert(( Round_Y >= -11160.0 ) and (Round_Y <= 11160.0 ));
```

gnatprove currently does not prove non linear equation. In the following example, the last assertion is not proved.

Example:

```
pragma Assert(X >= 0.0 and the x <= 180.0);
pragma Assert(Y >= -180.0 and then Y <= 0.0);
pragma Assert(Z >= 0.0 and then Z <= 1.0);
pragma Assert(X + Y >= 0.0);
Result := X + Y * Z;
pragma Assert (Result >= 0.0 and then Result <= 360.0);
```

2.2.2 Mission and Vehicle Management

Description of the case study

The Mission and Vehicle Management part of the Astrium case study consists in the implementation of a part of an ECSS (European Cooperation for Space Standardization) standard:

ECSS-E-ST-70-01C Space engineering - Spacecraft on-board control procedures

This standard defines the general principles of a Mission and Vehicle Management functionality. An on-board control procedure is in practice represented by a simplified programming language interpreted onboard the spacecraft. This interpreter is generally at the highest level of criticality of the spacecraft. The implementation of this interpreter in SPARK 2014 is table driven and relies greatly on Ada generic programming:

```
subtype R is Integer range 1 .. 100;
type T_Array is array (R range <>) of Boolean;

type T_Record (L : R) is record
   A : T_Array (1 .. L);
end record;

function G (X : T_Record) return Boolean is
  (for all I in X.A'Range => X.A (I));
```

The contracts defined on such code have (among others) the following objectives:

- Ensuring the permanent consistency of the software tables

- Ensuring the permanent consistency between the Mission and Vehicle Management function and the other functionalities (such as, for instance, the solar wing management)
- Ensuring the respect of some functional properties such as the mutual exclusion of execution of some automated procedures
- Ensuring the absence of run-time errors

The implementation of a reusable Mission and Vehicle Management relies greatly on

- Generic packages
- Discriminant

The generic packages allow an easy customization of the code:

```
generic
   -- the list of events
   type T_Event_Id is (<>);

package Mvm.Events is
```

The discriminants ensures a strict typing of the code, even in case of heterogeneous communication between components of the system:

```
type T_Type_Of_Monitoring is
  (No_Window,
   Time_Window,
   Protected_Window);

type T_Event_Status
  (Type_Of_Monitoring : T_Type_Of_Monitoring := No_Window)
is
   record
      Detection_Time : T_Float32;
      case Type_Of_Monitoring is
      when No_Window =>
         null;
      when Time_Window | Protected_Window =>
         Start_Window : T_Float32;
         End_Window : T_Float32;
      end case;
   end record;
```

Verification and validation of this case study

All the contracts have been checked by dynamic testing. This phase is quite classical, except for the fact that the testing includes the preconditions and the postconditions defined in the case study. Then, gnatprove has been applied.

This case study constains 10 parts:

- The Time_Management
- The Mathematical_Library
- The Single_Variable
- The List_Of_Variables
- The Events
- The Expressions
- The Parameters
- The Functional_Unit

- The Automated_Procedure
- The On_Board_Control_Procedure

Results for "Time_Management"

The analysis has been performed in 10 seconds (0 h 0 mn 10 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	3
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
postcondition	2	2	100	0	0
range_check	1	1	100	0	0
Total	3	3	100	0	0

Results for "Mathematical_Library"

The analysis has been performed in 475 seconds (0 h 7 mn 55 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	83
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
division_check	10	10	100	0	0
overflow_check	46	46	100	0	0
postcondition	26	24	92	2	7
precondition	6	6	100	0	0
range_check	49	47	95	2	4
Total	137	133	97	4	2

Analysis of the non proved VCs:

The reasons are similar to the mathematical library used for the SGS case study.

Results for "Single_Variable"

The analysis has been performed in 118 seconds (0 h 1 mn 58 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	85
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
discriminant_check	123	123	100	0	0
postcondition	30	30	100	0	0
precondition	115	115	100	0	0
Total	268	268	100	0	0

Results for "List_Of_Variables"

The analysis has been performed in 274 seconds (0 h 4 mn 34 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	140
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	85	85	100	0	0
loop_invariant_initialization	2	2	100	0	0
loop_invariant_preservation	2	2	100	0	0
postcondition	31	31	100	0	0
precondition	132	132	100	0	0
Total	252	252	100	0	0

Results for "Events"

The analysis has been performed in 371 seconds (0 h 6 mn 11 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	24
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	27	27	100	0	0
discriminant_check	104	104	100	0	0
loop_invariant_initialization	1	1	100	0	0
loop_invariant_preservation	1	1	100	0	0
overflow_check	5	5	100	0	0
postcondition	17	17	100	0	0
precondition	57	57	100	0	0
range_check	1	1	100	0	0
Total	213	213	100	0	0

Results for "Expressions"

The analysis has been performed in 1992 seconds (0 h 33 mn 12 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	331
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	385	385	100	0	0
discriminant_check	767	767	100	0	0
loop_invariant_initialization	2	2	100	0	0
loop_invariant_preservation	2	2	100	0	0
overflow_check	2	2	100	0	0
postcondition	97	97	100	0	0
precondition	413	413	100	0	0
range_check	2	2	100	0	0
Total	1670	1670	100	0	0

Results for "Parameters"

The analysis has been performed in 279 seconds (0 h 4 mn 39 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	62
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	0
Bodies not yet in alfa	0

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	2	2	100	0	0
discriminant_check	11	11	100	0	0
index_check	6	4	66	2	33
loop_invariant_initialization	1	1	100	0	0
loop_invariant_preservation	1	1	100	0	0
postcondition	2	2	100	0	0
precondition	1	1	100	0	0
range_check	7	7	100	0	0
Total	31	29	93	2	6

Analysis of the non proved VCs:

Gnatprove is not yet able to verify the index of an array which dimension is defined by a type discriminant, for example:

```
subtype R is Integer range 1 .. 100;
type T_Array is array (R range <>) of Boolean;

type T_Record (L : R) is record
   A : T_Array (1 .. L);
end record;

function G (X : T_Record) return Boolean is
  (for all I in X.A'Range => X.A (I));
```

In function "G", the index check "X.A (I)" is not proved even if "I" is defined in the range of "X.A"

An improvment of gnatprove is in progress in order to deal with such case (AdaCore ticket M416-015).

Results for "Functional_Unit"

The analysis has been performed in 2921 seconds (0 h 48 mn 41 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	76
Bodies not in alfa	0
Specifications not in alfa	0
Bodies not yet in alfa	13
Bodies not yet in alfa	13

The origins of subprograms not yet in SPARK 2014 are the following:

- class wide types (5 subprograms)
- tagged type (17 subprograms)

These origins are related to Object Oriented Programming. The analysis of Object Oriented software is foreseen but has not yet been implemented.

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	2	2	100	0	0
discriminant_check	58	58	100	0	0
index_check	26	4	15	22	84
loop_invariant_initialization	1	1	100	0	0
loop_invariant_preservation	1	1	100	0	0
postcondition	3	2	66	1	33
precondition	1	1	100	0	0
range_check	14	10	71	4	28
Total	106	79	74	27	25

Analysis of the non proved VCs:

Most of the non proved VCs are due to index of an array which dimension is defined by a type descriminant (see "Parameters" section).

The proof of the postcondition is not possible before the other proofs.

Results for "Automated_Procedure"

The analysis has been performed in 7803 seconds (2 h 10 mn 3 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	192
Bodies not in alfa	28
Specifications not in alfa	15
Bodies not yet in alfa	3
Bodies not yet in alfa	3

The origins of subprograms not in SPARK 2014 are the following:

- access (26 subprograms)

Accesses are used in the case study to store objects in a table. This kind of design can not be proved by gnatprove.

The origins of subprograms not yet in SPARK 2014 are the following:

- class wide types (20 subprograms)
- tagged type (22 subprograms)

As before, this is due to Object Oriented programming.

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved

- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	6	3	50	3	50
discriminant_check	158	157	99	1	0
index_check	68	17	25	51	75
loop_invariant_initialization	2	1	50	1	50
loop_invariant_preservation	2	2	100	0	0
postcondition	7	4	57	3	42
precondition	13	12	92	1	7
range_check	28	15	53	13	46
Total	284	211	74	73	25

A part of the non proved VCs are due to index of an array which dimension is defined by a type descriminant (see "Parameters" section).

The remaining non proved VCs are due to a too complex subprogram. This subprogram shall be split in several smaller subprograms to be proved.

Results for "On_Board_Control_Procedure"

The analysis has been performed in 13705 seconds (3 h 48 mn 25 s)

The following table shows the number of subprograms in SPARK 2014, not in SPARK 2014 or partially in SPARK 2014.

Subprograms fully in alfa	547
Bodies not in alfa	447
Specifications not in alfa	30
Bodies not yet in alfa	13
Bodies not yet in alfa	13

The origins of subprograms not in SPARK 2014 are the following:

- access (61 subprograms)
- unchecked conversion (377 subprograms)

Access is used with Object (see "Automated_Procedure" section)

The unchecked conversion are used in a library allowing reading external inputs. All the concerned subprograms are very small and shall be validated by intensive testin because there are out of the perimeter of Hi-Lite and of SPARK.

The origins of subprograms not yet in SPARK 2014 are the following:

- attribute (10 subprograms)
- class wide types (23 subprograms)
- tagged type (25 subprograms)

As before, this is due to Object Oriented programming.

The following table shows for each kind of VC (Verification Condition):

- The number of VCs proved
- The number of VCs not proved
- The percents (the sum of percents can be different from 100 because of rounding)
- The totals

Features	Total VC	Number proved	Percent proved	Number not proved	Percent not proved
assertion	418	415	99	3	0
discriminant_check	1113	1107	99	6	0
index_check	82	19	23	63	76
loop_invariant_initialization	5	4	80	1	20
loop_invariant_preservation	5	5	100	0	0
overflow_check	7	7	100	0	0
postcondition	148	130	87	18	12
precondition	637	628	98	9	1
range_check	39	24	61	15	38
Total	2454	2339	95	115	4

A part of the non proved VCs are due to index of an array which dimension is defined by a type descriminant (see "Parameters" section).

The remaining non proved VCs are due to too complex subprograms. These subprograms shall be split in several smaller subprograms to be proved.

2.3 Hi-Lite assessment

SPARK 2014 shall fulfill the following requirements:

- Capability to formalize the test cases
- Verification that all test procedures cover all the test cases
- Support to the verification / validation of the absence of run-time errors
- Support to the verification / validation of functional properties
- Verification of the correctness of the access of all global variables
- Verification of the absence of out of range values
- Internal consistency of software unit. The Hi-Lite tools shall help ensuring that each aggregate satisfies the requirements of the software item. In other words, each caller of a subprogram shall ensures that the pre-conditions of the callee are respected and that the post-condition of the callee is compatible with the caller
- Capability to assess the commandability and observability. In other words, the Hi-Lite tools shall help detecting dead code
- Help proving that numerical protection mechanisms are correctly implemented
- Prove the correctness of a generic code in a specific context

Here is an assessment of these requirements:

Requirements	Assessment
Formalization of the test cases	Very good
Test cases coverage	Not implemented. This implies that the formalization of the test cases is today not really interesting
Proof of absence of run-time errors	Very good
Proof functional properties	Good. In order to be efficient, it requires some work (definition of contracts, definition of assertions, spliting of complex subprograms in smaller ones...)
Correct access to all global variables	Very good
Absence of out of range values	Very good. As for functional properties, it requires some manual work
Internal consistency of software unit	Very good
Dead code detection	No
Correct numerical protection	Very good
Correctness of a generic code in a specific context	Very good

CHAPTER

THREE

ANALYSIS OF SPARK EXAMPLES AND TOKENEER

3.1 Introduction

Hi-Lite is a project aimed at lowering the entry barrier for the use of formal methods for the development of high-integrity software. It targets ease of adoption through a loose integration of formal proofs with testing and static analysis, allowing a combination of techniques to be used based on a common expression of specifications. Its technical focus is on modularity, that allows a divide-and-conquer approach to large software systems, as well as adoption early in the software life cycle.

The Hi-Lite project initially defined a subset of the Ada 2012 language, known as SPARK 2014 and the GNATprove tool was developed to perform the formal analysis and proof of SPARK 2014 code. SPARK 2014 has subsequently evolved into the SPARK 2014 language, and the GNATprove tool is being developed with the goal of analysing and proving SPARK 2014 programs. This report will therefore use the term "SPARK 2014" when referring to the language being analysed by GNATprove.

A key part of the Hi-Lite project is GNATprove, a formal verification tool for Ada, based on the GNAT compiler. It can prove that subprograms respect their contracts, expressed as preconditions and postconditions in the syntax of SPARK 2014. The goal of this report is to assess the Hi-Lite utilities, and in particular GNATprove, by applying them to case studies. Most of the examples have been taken from existing SPARK 2005 training material and other SPARK 2005 case studies, and converted to use SPARK 2014 aspects as necessary. This allows a comparison to be made between the analysis results from the SPARK 2005 tools and GNATprove for the same examples. We can also compare and contrast the code and assertions in each language.

The following sections describe the results of applying the Hi-Lite (GNAT, GNATprove and GPS) and SPARK 2005 tools (Examiner, Simplifier and Victor) to a variety of examples, culminating in a larger industrial case study - Tokeneer. For method, discussion, and conclusions, see *Discussion and Conclusions*. Toolset versions and performance statistics are provided in the last part of this chapter.

The main SPARK 2005 analysis tools used for this study were:

- the Examiner, which performs flow analysis and generates Verification Conditions (VCs);
- the Simplifier, an automatic theorem prover for SPARK VCs;
- SPARKbridge, which uses the Victor tool to translate SPARK VCs to SMT-LIB format and attempts to prove them using SMT solvers (the default being Alt-Ergo).

The code examples presented in the following sections are written in SPARK 2014. In order to make comparisons, SPARK 2005 versions of the same examples have also been created but these are not generally shown in the body of this report. All the source code for the examples is available under docs/case_study in the SPARK 2014 repository (https://forge.open-do.org/anonscm/git/spark2014/spark2014.git).

The report sometimes makes references to tickets (TNs) in the form [ANNN-NNN]. These refer to the unique identifiers used to track issues in AdaCore's GNAT Tracker system.

A number of observations and recommendations are made throughout the report, and these are summarised at the end.

3.2 Exchange Procedure

The first example is taken from the Software Engineering with SPARK training course. The aim is to evaluate the tools on some straightforward examples (including correct implementations and versions with deliberate errors). Some of the examples have postconditions specified, so the tools can be used to verify the implementations against those postconditions. In other cases there are no postconditions but the tools can still be used to verify freedom from runtime exceptions.

3.2.1 Exchange_Examples

The package Exchange_Examples contains a number of variations on procedure Exchange. The procedure swaps the values of its two Integer parameters. There are variants with and without postconditions, with a correct implementation and with various deliberate errors. Some of these errors are detectable by flow analysis and some require proof.

3.2.2 Correct implementation of Exchange

The first version of Exchange is the correct implementation. Note that it is preceded by a separate declaration which is required in order to attach the postcondition aspect. (Postcondition aspects can only be attached to declarations, not bodies.)

Note: Observation 01: The restriction that Post aspects are only allowed on subprogram specifications will be removed, so GNATprove will permit postconditions to appear on bodies for which there is no separate specification [M227-046].

```
procedure Exchange (X, Y : in out Integer)
  with Post => (X = Y'Old and Y = X'Old);

procedure Exchange (X, Y : in out Integer) is
   T : Integer;
begin
   T := X;
   X := Y;
   Y := T;
end Exchange;
```

GNATprove successfully generates and discharges all the VCs for this version.

3.2.3 Exchange with no postcondition

This is identical to the correct version but it has no postcondition. Again, GNATprove successfully generates and discharges all VCs for this version. This is to be expected as the proof obligations are the same as for the previous version but without any correctness VCs for the postcondition.

```
procedure Exchange_No_Post (X, Y : in out Integer) is
   T : Integer;
begin
   T := X;
   X := Y;
   Y := T;
end Exchange_No_Post;
```

3.2.4 Exchange with unused variable

In this variant, the programmer has incorrectly assigned the value of X to Y in the final statement instead of assigning T to Y. This means that the value assigned to T is never used. (As a consequence, the procedure does not

exchange the values of its parameters, but as there is no postcondition we cannot expect the tools to detect that!) GNATprove generates and discharges all the VCs for this version.

```
procedure Exchange_No_Post_Unused (X, Y : in out Integer) is
   T : Integer;
begin
   T := X;
   X := Y;
   Y := X;
end Exchange_No_Post_Unused;
```

When analysed with –mode=flow GNATprove reports that X is an ineffective import and that the assignment of X to T is an ineffective statement. At present, flow analysis and proof have to be run separately in GNATprove but eventually the tool will perform both flow analysis and proof by default each time it is run.

Note: Observation 02: From a usability perspective it would be preferable if flow analysis and proof were not separate modes of operation. This is a known issue [M327-024] and will be addressed in a future version of GNATprove.

3.2.5 Exchange with uninitialized variable

In this version the programmer has forgotten to include the initial assignment from X to T. There is no postcondition so the tools cannot be expected to complain that X and Y are not swapped, but we would expect them to detect that the uninitialized value of T is assigned to Y.

```
procedure Exchange_No_Post_Uninitialised (X, Y : in out Integer) is
   T : Integer;
begin
   X := Y;
   Y := T;
end Exchange_No_Post_Uninitialised;
```

When analysed with –mode=flow GNATprove reports the use of uninitialized variable T in the assignment to Y. All VCs are discharged by GNATprove.

3.2.6 Exchange with unused variable and postcondition

This is the version seen earlier where the programmer has made a mistake in the final statement and typed "Y := X" instead of "Y := T". This time a postcondition has been added so the tools are able to detect that the implementation is not correct. A separate declaration has been provided so that the postcondition can be specified.

```
procedure Exchange_With_Post_Unused (X, Y : in out Integer)
  with Post => (X = Y'Old and Y = X'Old);

procedure Exchange_With_Post_Unused (X, Y : in out Integer) is
   T : Integer;
begin
   T := X;
   X := Y;
   Y := X;
end Exchange_With_Post_Unused;
```

GNATprove reports "postcondition not proved, requires Y = X'old". GPS highlights the line where the postcondition appears, and the error message explains what the problem is, in the context of the source code.

Note: Observation 03: The GNATprove/GPS integration provides the facility to display the path to an unproven check. To enable this feature the option –proof=then_split or –proof=path_wp must be used, and a small icon appears next to the line number where the unproved check occurs. Clicking this icon causes the path to the unproved check to be highlighted. This feature can be useful for debugging failed proofs when there are multiple paths leading to a check.

When analysed with –mode=flow GNATprove reports that X is an ineffective import and that the assignment of X to T is an ineffective statement.

3.2.7 Exchange with uninitialized variable and postcondition

This is identical to the "Exchange with uninitialized variable" example seen above but with the addition of a postcondition.

```
procedure Exchange_With_Post_Uninitialised (X, Y : in out Integer)
  with Post => (X = Y'Old and Y = X'Old);

procedure Exchange_With_Post_Uninitialised (X, Y : in out Integer) is
   T : Integer;
begin
   X := Y;
   Y := T;
end Exchange_With_Post_Uninitialised;
```

GNATprove is able to detect the discrepancy between the postcondition and the implementation and reports "post-condition not proved, requires Y = X'old". Again, GPS highlights the line where the postcondition appears, and the error message explains what the problem is, in the context of the source code.

When analysed with –mode=flow GNATprove reports the use of uninitialized variable T in the assignment to Y.

3.2.8 Exchange with runtime error

In this final version of the exchange procedure a potential runtime error has been introduced in the assignment to X. The intermediate subexpression "Y + 2 - 2" has the potential to overflow if Y is Integer'Last-1, although the final result will always be within the range of Integer.

```
procedure Exchange_RTE (X, Y : in out Integer)
  with Pre  => Y < Integer'Last,
       Post => (X = Y'Old and Y = X'Old);

procedure Exchange_RTE (X, Y : in out Integer) is
   T : Integer;
begin
   T := X + 0;
   X := Y + 2 - 2;
   Y := T;
end Exchange_RTE;
```

GNATprove reports that the overflow check is not proved for the line where Y + 2 - 2 is assigned to X. This is as expected because although the result is guaranteed to be within the range of Integer, the intermediate expression Y + 2 may not be. All other VCs are proved. Note that if the expression is changed to "Y + 1 - 1" then the overflow check is proved because the precondition guarantees that the initial value of Y is strictly less than Integer'Last.

3.2.9 Usability of GNATprove and GPS

GNATprove can be invoked directly from menus within GPS, as can the SPARK 2005 tools. In the examples above, when GNATprove is unable to discharge a VC, the error message is presented in source code terms ("post-condition not proved, requires Y = X'old") and the line of code where the postcondition appears is highlighted. The SPARK 2005 toolset, on the other hand, would present the user with an undischarged VC and the user would be responsible for mapping that back to the source code. For these simple examples it seems more user-friendly to work entirely at the source code level and to hide the underlying VCs from the user, as GNATprove does. Larger and more complex examples later in this study will attempt to determine how well the approach scales.

Both GNATprove and the SPARK 2005 tools require multiple steps to be performed in order to fully analyse the code. In GNATprove these steps are:

- Analyse with –mode=flow to perform flow analysis.
- Analyse with –mode=prove in order to generate and discharge VCs.

In SPARK 2005 the steps are:

- Analyse with the Examiner to detect semantic and flow errors, and to generate VCs.
- Run SPARKsimp to discharge the VCs (with the Simplifier and, optionally other tools).
- Run POGS to summarise the analysis results.

Both GNATprove and the SPARK 2005 tools might benefit from combining these steps into a single command which invokes flow analysis and proof, as the user may not wish to be troubled by the distinction between the analysis phases. It is understood that this will be done for GNATprove - the current separation between the two modes is only present for implementation and development purposes as observed earlier.

3.3 Stacks, Queues and QueueOperations

3.3.1 Stacks and Queues package specifications

Here we are provided with the specification for a package which provides a (private) Stack type and some operations on it. A second package provides a (private) Queue type and operations. The bodies of the packages are not provided. The aim of this example is to investigate how GNATprove deals with the analysis of calls to subprograms for which no body has been provided.

```
package Stacks is

   type Stack is limited private;

   function EmptyStack(S : in Stack) return Boolean;

   function FullStack(S : in Stack) return Boolean;

   procedure ClearStack(S :    out Stack);

   procedure Push(S : in out Stack; X : in Integer);

   procedure Pop(S : in out Stack; X : out Integer);

   procedure Top(S : in     Stack; X : out Integer);

private
   type Stack is new Integer;
end Stacks;

package Queues is

   type Queue is limited private;

   function EmptyQueue(Q : in Queue) return Boolean;

   function FullQueue(Q : in Queue) return Boolean;

   procedure ClearQueue(Q : out Queue);

   procedure EnQueue(Q : in out Queue; X : in Integer);

   procedure DeQueue(Q : in out Queue; X : out Integer);

private
```

```
   type Queue is new Integer;
end Queues;
```

3.3.2 QueueOperations package

QueueOperations is a client of Stacks and Queues. It reverses the elements in a Queue object by pushing them onto a Stack, then popping them back off and into the Queue again. The body of Queues is provided, but it makes use of Stacks and Queues for which no bodies are available for analysis.

```
with Stacks, Queues;
--# inherit Stacks, Queues;
package QueueOperations is

   procedure ReverseQueue(Q : in out Queues.Queue);

end QueueOperations;

package body QueueOperations is

   procedure ReverseQueue(Q : in out Queues.Queue) is
      S: Stacks.Stack;
      X: Integer;
   begin
      Stacks.ClearStack(S);
      while not Queues.EmptyQueue(Q) loop
         Queues.DeQueue(Q, X);
         Stacks.Push(S, X);
      end loop;
      while not Stacks.EmptyStack(S) loop
         Stacks.Pop(S, X);
         Queues.EnQueue(Q, X);
      end loop;
   end ReverseQueue;

end QueueOperations;
```

Note: Observation 04: At present, attempting to analyse QueueOperations with –mode=flow results in the error "raised WHY.NOT_IMPLEMENTED : flow-control_flow_graph.adb:513". This will be addressed in a future version of GNATprove.

When analysed in proof mode GNATprove is able to discharge all VCs for QueueOperations.

This example demonstrates that GNATprove is able to analyse partial programs, for cases where package specifications are provided but there are no bodies. If the body was provided but only some of the subprograms were implemented then stubs or null bodies would need to be provided for the remainder.

Note: Observation 05: Using stubs for which no completion is provided is not currently possible with GNATprove as it results in an internal error. This issue [M320-027] will be addressed in a future version of the tool. Another option would be to use the facilities of SPARK 2014 to mark code in or out of SPARK. The rules for this language feature are currently under development.

3.4 Stacks, Queues and QueueOperations with Proof

Now we revisit the Stacks, Queues and QueueOperations packages but this time the bodies of Stacks and Queues have been completed and proof contracts have been added to their specifications. For example in the procedure Push shown below the stack pointer is incremented with the possibility of going out of the range of its type, Count_T. This is guarded against by placing a precondition on the specification to ensure that we don't try to push when the stack is full.

```
procedure Push(S : in out Stack; X : in Integer)
  with pre => not FullStack (S);

procedure Push(S : in out Stack; X : in Integer) is
begin
   S.The_Top := S.The_Top + 1;
   S.The_Stack (S.The_Top) := X;
end Push;
```

The function FullStack, referenced in precondition to Push, currently has no postcondition. We will come back to this shortly.

```
function FullStack(S : in Stack) return Boolean;
```

GNATprove is able to discharge all the VCs with the exception of those for the calls to Stacks.Push and Queues.Enqueue in the code below, because it cannot be shown that their preconditions are met. (In the SPARK 2005 version of the code the same two precondition checks are also not proved.)

```
procedure ReverseQueue(Q : in out Queues.Queue) is
   S: Stacks.Stack;
   X: Integer;
begin
   Stacks.ClearStack(S);
   while not Queues.EmptyQueue(Q) loop
      Queues.DeQueue(Q, X);
      Stacks.Push(S, X);
   end loop;
   while not Stacks.EmptyStack(S) loop
      Stacks.Pop(S, X);
      Queues.EnQueue(Q, X);
   end loop;
end ReverseQueue;
```

For Stacks.Push it is necessary to show that the Stack is not full. We know this is true because the stack is cleared before entering the loop and because the maximum number of elements in a queue is not greater than the maximum number of elements on a stack. (The maximum number of elements happens to be the same for queues and stacks but this requires knowledge of the private types in their respective package specifications.)

As well as adding loop invariants it was necessary to change the code and the contracts in both the SPARK 2014 and the SPARK 2005 versions of the example in order to complete the proof. The modified code for the Stacks package is shown below.

```
package Stacks is

   type Stack is private; -- had to remove 'limited' to allow use of 'Old
   Max_Count : constant Integer := 100;
   subtype Stack_Size is Natural range 0 .. Max_Count;

   -- Proof function that should not be called in code.
   function Size (S : in Stack) return Stack_Size;

   ...

   procedure Push (S : in out Stack; X : in Integer)
     with pre  => Size (S) in 0 .. Max_Count - 1,
          post => Size (S) = Size(S'Old) + 1;
```

More of the implementation detail has become visible in the specification, breaking the abstraction to some extent. We can now see that the stack has up to 100 elements, and we have provided a function to get the current size. As the Size function makes use of the 'Old attribute on type Stack we have had to change Stack from limited private to private (because 'Old is not permitted for limited private types). The function Size is only needed in contracts and should not be called in general code. The loss of abstraction seems to be inevitable in order make the necessary information available to complete the proof.

Note: Recommendation 01: It was sometimes found to be necessary to make implementation detail public for proof purposes when it would otherwise have been private (see Stacks and Tokeneer for examples). This loss of abstraction is undesirable and it is recommended that features are added to the SPARK 2014 language and tools to address this issue. (In fact such features are currently being designed.)

The SPARK 2005 version shown below is very similar, although it allows Stack to remain as limited private and Size is a proof function so there is no possibility of it being called in general code.

```
package Stacks is

   type Stack is limited private;
   Max_Count : constant Integer := 100;
   subtype Stack_Size is Natural range 0 .. Max_Count;

   --# function Size (S : in Stack) return Stack_Size;

   ...

   procedure Push(S : in out Stack; X : in Integer);
   --# pre  (Size (S) in 0 .. Max_Count-1);
   --# post (Size(S) = Size(S~) + 1);
```

The definition of the proof function is provided in the body of the package where the details of type Stack are visible.

```
--# function Size (S : in Stack) return Stack_Size;
--# return S.The_Top;
```

Note: Observation 06: Note that modelling SPARK 2005 proof functions as executable functions in SPARK 2014 is not ideal as there is nothing to prevent them from being called in general code. The solution to this is to label them as ghost functions via "convention => ghost" which means they may only be called from within proof expressions or from other ghost functions. This approach would have been taken for this example but it was not yet implemented when the example was originally developed.

The modifications to the Queues package are very similar to those made to the Stacks package so it is not reproduced here. The updated ReverseQueue procedure looks like this.

```
-- spec
procedure ReverseQueue(Q : in out Queues.Queue)
  with Pre => Queues.Size(Q) in Queues.Queue_Size;

-- body
procedure ReverseQueue(Q : in out Queues.Queue) is
   S: Stacks.Stack;
   X: Integer;
   Count : Natural := 0;
begin
   Stacks.ClearStack(S);
   while not Queues.EmptyQueue(Q) loop
      pragma Loop_Invariant ((Queues.Size(Q) in 1 .. Queues.Max_Count) and
                             (Stacks.Size(S) = Count) and
                             (Count = Queues.Size(Q'Loop_Entry) - Queues.Size(Q)));
      Queues.DeQueue(Q, X);
      Stacks.Push(S, X);
      Count := Count + 1;
   end loop;
   while Count > 0 loop
      Stacks.Pop(S, X);
      Queues.EnQueue(Q, X);
      Count := Count - 1;
      pragma Loop_Invariant ((Stacks.Size(S) = Count) and
                             (Queues.Size(Q) >= 1) and
                             (Queues.Size(Q) = Count'Loop_Entry - Count));
   end loop;
```

```
end ReverseQueue;
```

With these loop invariants GNATprove is able to prove that the code is free from runtime exceptions. The SPARK 2005 version is almost identical and is similarly proved free from exceptions by the SPARK 2005 tools. (Note that there is no postcondition on ReverseQueue so there is no proof that ReverseQueue actually reverses the elements of the queue.)

Note that the local variable Count was introduced for proof purposes and does not actually need to be used in the code itself. (The test for "Count > 0" in the second loop could be replaced by "not EmptyStack(S)" and indeed it was written like that in an earlier attempt at the SPARK 2005 implementation, but making use of Count in the code avoids a flow error from the Examiner.)

Note: Recommendation 02: Sometimes variables are introduced for proof purposes only and they are not actually needed in general executable code. It is recommended (in both SPARK 2005 and SPARK 2014) that a mechanism be introduced for declaring "ghost variables" for use in proof only. The design of this feature is already underway in SPARK 2014.

3.5 Central Heating Controller

3.5.1 Proof of Absence of Runtime Errors

Description

This example is based on a larger tutorial exercise from the Software Engineering with SPARK course. The code implements a Central Heating Controller which reads from various physical input devices and writes to various output devices. The input and output devices themselves are managed via packages which are provided as specifications with no bodies. The main program implements the functionality of the controller itself, and is derived from a formal Z specification.

This section considers proof of absence of runtime errors in the heating controller. The next section adds postconditions and investigates their proof.

Note that the original SPARK 2005 interface packages use external own variables to represent the input and output devices. These are not (always) modelled in the SPARK 2014 aspects. For example, in the AdvanceButton.Raw package shown below the physical input from the button is represented by the moded own variable Inputs, but there is no analogous entity in the SPARK 2014 model.

```
-- Raw Advance Button Boundary Package
-- boundary package providing raw access to the advance switch
private package AdvanceButton.Raw
--# own in Inputs;
is
   procedure Read (Pressed : out Boolean);
   --# global  in Inputs;
   --# derives Pressed from Inputs;
   --
   -- Pressed is True if the advance button is down.

end AdvanceButton.Raw;
```

But in the Clock package the external own variable Times is modelled in the SPARK 2014 aspects, at least in that the Read procedure has a postcondition that makes use of the proof function PF_Read.

```
-- Clock
package Clock
--# own in Ticks : Times;
is
  subtype Times is Integer range 0 .. 86399;

  function PF_Read return Times;
```

```
   procedure Read (Time : out Times)
     with Post => (Time = PF_Read);
   --# global  in Ticks;
   --# derives Time from Ticks;
   --# post (Time = Ticks~);
   --  Once again "and (Ticks = Ticks'Tail (Ticks~));" has been omitted for simplicity.
   --
   -- Time contains the number of seconds since the controller was powered
   -- up and resets to zero every 24 hours.

end Clock;
```

As observed earlier, it would be useful to be able to identify such "proof functions" in SPARK 2014.

Note: Recommendation 03: SPARK 2005 uses external own variables to model inputs and outputs at the interface with the outside world, and the SPARK 2005 tools treat these variables as 'special' for flow analysis and proof. SPARK 2014 currently lacks support for modelling such variables, and it is recommended that such support be added. This will be dealt with using state abstractions (Abstract_State aspect).

Results

Note that the code for this example has not been included in its entirety here. There are 479 lines including whitespace and comments. Approx 50% of these lines are executable. The code includes the original SPARK annotations. Proof aspects have also been added to support GNATprove.

Note: Observation 07: Flow analysis of the Central Heating Controller example with GNATprove is not currently possible due to use of features for which flow analysis has not yet been implemented.

Both GNATprove and the SPARK 2005 tools are able to discharge all VCs for this example.

3.5.2 Partial Correctness Proof

There is also a version of the central heating controller code in which postconditions have been added, so proof of partial correctness can be attempted. The loop invariants have been modified accordingly to support the proof of the postconditions.

The postconditions on the higher level subprograms are quite large. For example, here is the postcondition on the procedure CheckModeSwitch in both SPARK 2014 and SPARK 2005 with the derivation from the Z specification shown in the form of comments.

```
procedure CheckModeSwitch
  with Pre => (HeatingIsOn = Actuator.IsOn),  -- invariant condition
  Post => ((HeatingIsOn = Actuator.IsOn) and  -- invariant condition

  ---------- Mode switch in off position ----------------------Z Schema: +--ModeOff--(page 6)--
  (if (ModeSwitch.PF_Read = ModeSwitch.off) then                    --| mode? = off
  (not Indicator.IsOn and                                           --| Indicator' = isOff
     (not HeatingIsOn) and                                          --| Heating' = isOff
     (not Actuator.IsOn))) and                                      --+-------------------------

---------- Mode switch in continuous position ----------------Z schema: ModeContinuous -(page 7,
                                                                  --+--ModeContinuousOp-------
  (if (ModeSwitch.PF_Read = ModeSwitch.cont) then                 --| mode? = continuous
  (Indicator.IsOn)) and                                           --| Indicator' = isOn
                                                                  --+-------------------------
                                                                  --+-ModeContinuousOff-------
  (if ((ModeSwitch.PF_Read = ModeSwitch.cont) and                 --| ModeContinuousOp
        Thermostat.RoomTooWarm) then                              --+------------------
  ((not HeatingIsOn) and                                          --| thermostat? = aboveTemp
     (not Actuator.IsOn))) and                                    --| Heating' = IsOff
                                                                  --+-------------------------
```

```
                                                                    --+-ModeCOntinuousOn--------
   (if ((ModeSwitch.PF_Read = ModeSwitch.cont) and                  --| ModeContinuousOp
         not Thermostat.RoomTooWarm) then                           --+------------------
   (HeatingIsOn and                                                 --| thermostat? = belowTemp
      Actuator.IsOn)) and                                           --|  Heating' = IsOn
                                                                    --+-------------------------

----------- Mode switch in timed position ---------------------Z schema: ModeTimed -(page 8, comp
                                                                    --+-ModeTimedPossiblyOn-----
   (if ((ModeSwitch.PF_Read = ModeSwitch.timed) and                 --| mode? = timed
         IsInOperatingPeriod (Clock.PF_Read,                        --| ... in operating period
                              ClockOffset,                          --| -- Indicator' = isOn
                              OnOffTime)) then                      --+-------------------------
   (Indicator.IsOn)) and
                                                                    --+-ModeTimedOff------------
   (if ((ModeSwitch.PF_Read = ModeSwitch.timed) and                 --| mode? = timed
         not IsInOperatingPeriod (Clock.PF_Read,                    --| ... not in operating peri
                                  ClockOffset,                      --| Indicator' = isOff
                                  OnOffTime)) then                  --| Heating' = isOff
   ((not Indicator.IsOn) and                                        --+-------------------------
      (not HeatingIsOn) and
      (not Actuator.IsOn))) and

                                                                    --+-ModeTimedAboveTemp------
   (if ((ModeSwitch.PF_Read = ModeSwitch.timed) and                 --| ModeTimedPossiblyOn
         IsInOperatingPeriod (Clock.PF_Read,                        --+--------------------
                              ClockOffset,                          --| thermostat? = aboveTemp
                              OnOffTime) and                        --| Heating' = isOff
         Thermostat.RoomTooWarm) then                               --+-------------------------
   ((not HeatingIsOn) and
      (not Actuator.IsOn))) and

                                                                    --+-ModeTimedBelowTemp------
   (if ((ModeSwitch.PF_Read = ModeSwitch.timed) and                 --| ModeTimedPossiblyOn
         IsInOperatingPeriod (Clock.PF_Read,                        --+--------------------
                              ClockOffset,                          --| thermostat? = belowTemp
                              OnOffTime) and                        --| Heating' = isOn
         not Thermostat.RoomTooWarm) then                           --+-------------------------
   (HeatingIsOn and
      Actuator.IsOn)));

procedure CheckModeSwitch
--# global  in     Thermostat.Inputs,
--#                Clock.Ticks,
--#                ModeSwitch.Inputs,
--#                OnOffTime,
--#                ClockOffset;
--#            out Indicator.Outputs,
--#                Actuator.Outputs;
--#         in out HeatingIsOn;
--# derives Actuator.Outputs,
--#         HeatingIsOn          from Thermostat.Inputs,
--#                                   Clock.Ticks,
--#                                   ModeSwitch.Inputs,
--#                                   OnOffTime,
--#                                   ClockOffset,
--#                                   HeatingIsOn &
--#         Indicator.Outputs    from Clock.Ticks,
--#                                   ModeSwitch.Inputs,
--#                                   OnOffTime,
--#                                   ClockOffset;

--# pre   HeatingIsOn <-> Actuator.IsOn (Actuator.Outputs);  -- invariant condition

--# post (HeatingIsOn <-> Actuator.IsOn (Actuator.Outputs))  -- invariant condition
```

```
--#
--#      and
--#
--# ----------- Mode switch in off position ----------------------Z Schema: +--ModeOff--(page 6)
--#      ((ModeSwitch.Inputs~ = ModeSwitch.off) ->                          --| mode? = off
--#         (not Indicator.IsOn (Indicator.Outputs) and                     --| Indicator' = isOff
--#            (not HeatingIsOn) and                                        --| Heating' = isOff
--#               (not Actuator.IsOn (Actuator.Outputs))))                  --+--------------------
--#
--#      and
--#
--# ----------- Mode switch in continuous position ----------------Z schema: ModeContinuous -(page
--#
--#                                                                         --+--ModeContinuousOp--
--#         ((ModeSwitch.Inputs~ = ModeSwitch.cont) ->                      --| mode? = continuous
--#            (Indicator.IsOn (Indicator.Outputs)))                        --| Indicator' = isOn
--#                                                                         --+--------------------
--#      and
--#                                                                         --+-ModeContinuousOff--
--#         (((ModeSwitch.Inputs~ = ModeSwitch.cont) and                    --| ModeContinuousOp
--#           Thermostat.RoomTooWarm (Thermostat.Inputs~)) ->               --+------------------
--#              ((not HeatingIsOn) and                                     --| thermostat? = above
--#                 (not Actuator.IsOn (Actuator.Outputs))))                --| Heating' = IsOff
--#                                                                         --+--------------------
--#      and
--#                                                                         --+-ModeContinuousOn---
--#         (((ModeSwitch.Inputs~ = ModeSwitch.cont) and                    --| ModeContinuousOp
--#           not Thermostat.RoomTooWarm (Thermostat.Inputs~)) ->           --+------------------
--#              (HeatingIsOn and                                           --| thermostat? = belo
--#                 Actuator.IsOn (Actuator.Outputs)))                      --|  Heating' = IsOn
--#                                                                         --+--------------------
--#      and
--#
--# ----------- Mode switch in timed position ---------------------Z schema: ModeTimed -(page 8,
--#
--#                                                                         --+-ModeTimedPossiblyO
--#      (((ModeSwitch.Inputs~ = ModeSwitch.timed) and                      --| mode? = timed
--#          IsInOperatingPeriod (Clock.Ticks~,                             --| ... in operating p
--#                               ClockOffset,                              --| -- Indicator' = is
--#                               OnOffTime)) ->                            --+--------------------
--#            (Indicator.IsOn (Indicator.Outputs)))
--#
--#      and
--#                                                                         --+-ModeTimedOff-------
--#      (((ModeSwitch.Inputs~ = ModeSwitch.timed) and                      --| mode? = timed
--#         not IsInOperatingPeriod (Clock.Ticks~,                          --| ... not in operati
--#                                  ClockOffset,                           --| Indicator' = isOff
--#                                  OnOffTime)) ->                         --| Heating' = isOff
--#            ((not Indicator.IsOn (Indicator.Outputs)) and                --+--------------------
--#               (not HeatingIsOn) and
--#                  (not Actuator.IsOn (Actuator.Outputs))))
--#
--#      and
--#                                                                         --+-ModeTimedAboveTemp
--#      (((ModeSwitch.Inputs~ = ModeSwitch.timed) and                      --| ModeTimedPossiblyO
--#          IsInOperatingPeriod (Clock.Ticks~,                             --+--------------------
--#                               ClockOffset,                              --| thermostat? = abov
--#                               OnOffTime) and                            --| Heating' = isOff
--#             Thermostat.RoomTooWarm (Thermostat.Inputs~)) ->             --+--------------------
--#               ((not HeatingIsOn) and
--#                  (not Actuator.IsOn (Actuator.Outputs))))
--#
--#      and
```

```
--#                                                                    --+-ModeTimedBelowTemp
--#      (((ModeSwitch.Inputs~ = ModeSwitch.timed) and                 --| ModeTimedPossiblyO
--#          IsInOperatingPeriod (Clock.Ticks~,                        --+--------------------
--#                               ClockOffset,                         --| thermostat? = belo
--#                               OnOffTime) and                       --| Heating' = isOn
--#           not Thermostat.RoomTooWarm (Thermostat.Inputs~)) ->      --+--------------------
--#              (HeatingIsOn and
--#                 Actuator.IsOn (Actuator.Outputs)))
--#      ;
```

All VCs are discharged by GNATprove and by the SPARK 2005 tools for the SPARK 2014 and 2005 versions of the code respectively. Although large (up to around 50 lines) the postconditions consist of fairly straightforward Boolean expressions using mainly "and" and "or" operators.

Note: Observation 08: The partial correctness proof of the Central Heating Controller demonstrates that GNATprove is able to discharge quite large postconditions in a reasonable time. (The default timeout of 1s was sufficient. Full statistics are in the appendix.)

Note: Observation 09: It is interesting to note that GNATprove generates and proves 37 VCs, compared to 97 VCs for the SPARK 2005 tools. This highlights the different VC generation schemes used. GNATprove generates one VC for each check, whilst SPARK 2014 generates one VC for each path to each check. This results in SPARK 2005 having a larger number of VCs compared to GNATprove, but they tend to be smaller.

3.6 Examples from the Advanced SPARK 2005 Training Course

This section considers some code examples from the tutorials on the Advanced SPARK Program Design and Verification training course. These are aimed at teaching users techniques for writing SPARK 2005 code in a proof-friendly way. As before, the original SPARK 2005 code has been translated into SPARK 2014 for comparison.

3.6.1 Increment

In this example a procedure takes an Integer, X, of mode 'in out' and adds one to it. A suitable precondition must be provided in order to guard against overflow.

```
package T1Q1 is

  procedure Increment (X: in out Integer)
    with Pre => (X < Integer'Last),
         Post => (X = X'Old + 1);
  --# derives X from X;
  --# pre  X < Integer'Last;
  --# post X = X~ + 1;

end T1Q1;
```

```
package body T1Q1 is

  procedure Increment (X: in out Integer) is
  begin
    X := X + 1;
  end Increment;

end T1Q1;
```

The code shows the original SPARK 2005 annotations and the SPARK 2014 aspects, which are very similar. GNATprove discharges all the VCs for this example.

3.6.2 Increment2

This procedure takes two Integer parameters and increments both of them. A suitable precondition guards against overflow. In the original SPARK 2005 example an assertion is inserted between the two assignment statements. The point of this is to illustrate to SPARK 2005 users that the assertion is "forgetful" so users must add any information to it that they wish to be carried forward. The equivalent in SPARK 2014 is the pragma Assert_And_Cut. As the 'Old attribute may only be referred to in postconditions, local constants have been declared to represent the initial values of X and Y.

Note: Recommendation 04: The current inability to reference the *Old* and *Loop_Entry* attributes in assertions and loop invariants needs to be resolved. If the language rules cannot be relaxed then ghost variables may offer an acceptable solution to this issue.

```
package T1Q2 is

  procedure Increment2 (X, Y: in out Integer)
    with Pre => ((X /= Integer'Last) and (Y /= Integer'Last)),
    Post => ((X = X'Old + 1) and (Y = Y'Old + 1));
  --# derives X from X & Y from Y;
  --# pre X /= Integer'Last and Y /= Integer'Last;
  --# post X = X~ + 1 and Y = Y~ + 1;

end T1Q2;

package body T1Q2 is

  procedure Increment2 (X,Y: in out Integer) is
      X_Old : constant Integer := X;
      Y_Old : constant Integer := Y;
  begin
    X := X + 1;
    pragma Assert_And_Cut ((X = X_Old + 1) and (Y = Y_Old));
    --# assert X = X~ + 1 and Y = Y~;
    Y := Y + 1;
  end Increment2;

end T1Q2;
```

All GNATprove VCs are discharged automatically.

3.6.3 Swap, NandGate and NextDay

In these tutorials the specifications of several subprograms are provided and the students are tasked with adding suitable bodies which implement the specifications and are proved by the tools.

The postcondition on NandGate illustrates the equivalence between the use of implication "A -> B" in SPARK 2005 and "if A then B" in SPARK 2014 aspects.

For the NextDay example the intention is to see whether the implementation using the 'Succ attribute or the implementation with a case statement is easier to prove.

```
package T1Q3 is

  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  procedure Swap(X,Y: in out Integer)
    with Post => ((X = Y'Old) and (Y = X'Old));
  --# derives X from Y & Y from X;
  --# post X = Y~ and Y = X~;

  procedure NandGate(P,Q: in Boolean; R: out Boolean)
    with Post => ((if ((not P) and (not Q)) then R) and
```

```
    (if ((not P) and Q) then R) and
    (if (P and (not Q)) then R) and
    (if (P and Q) then (not R)));
  --# derives R from P, Q;
  --# post ( ((not P) and (not Q)) -> R    ) and
  --#      ( ((not P) and    Q   ) -> R    ) and
  --#      ( (   P    and (not Q)) -> R    ) and
  --#      ( (   P    and    Q   ) -> not R);
  --  The above is the truth table for "A nand B"

  procedure NextDay_A(Today: in Day; Tomorrow: out Day)
    with Post => ((Today=Mon and Tomorrow=Tue) or
                  (Today=Tue and Tomorrow=Wed) or
                  (Today=Wed and Tomorrow=Thu) or
                  (Today=Thu and Tomorrow=Fri) or
                  (Today=Fri and Tomorrow=Sat) or
                  (Today=Sat and Tomorrow=Sun) or
                  (Today=Sun and Tomorrow=Mon));
  --# derives Tomorrow from Today;
  --# post (Today=Mon and Tomorrow=Tue) or
  --#      (Today=Tue and Tomorrow=Wed) or
  --#      (Today=Wed and Tomorrow=Thu) or
  --#      (Today=Thu and Tomorrow=Fri) or
  --#      (Today=Fri and Tomorrow=Sat) or
  --#      (Today=Sat and Tomorrow=Sun) or
  --#      (Today=Sun and Tomorrow=Mon);

  procedure NextDay_B(Today: in Day; Tomorrow: out Day)
    with Post => ((Today=Mon and Tomorrow=Tue) or
                  (Today=Tue and Tomorrow=Wed) or
                  (Today=Wed and Tomorrow=Thu) or
                  (Today=Thu and Tomorrow=Fri) or
                  (Today=Fri and Tomorrow=Sat) or
                  (Today=Sat and Tomorrow=Sun) or
                  (Today=Sun and Tomorrow=Mon));
  --# derives Tomorrow from Today;
  --# post (Today=Mon and Tomorrow=Tue) or
  --#      (Today=Tue and Tomorrow=Wed) or
  --#      (Today=Wed and Tomorrow=Thu) or
  --#      (Today=Thu and Tomorrow=Fri) or
  --#      (Today=Fri and Tomorrow=Sat) or
  --#      (Today=Sat and Tomorrow=Sun) or
  --#      (Today=Sun and Tomorrow=Mon);

end T1Q3;

package body T1Q3 is

  procedure Swap(X,Y: in out Integer) is
    Temp: Integer;
  begin
    Temp := X;
    X := Y;
    Y := Temp;
  end Swap;

  procedure NandGate(P,Q: in Boolean; R: out Boolean) is
  begin
    R := not (P and Q);  -- simplest implementation
  end NandGate;

  procedure NextDay_A(Today: in Day; Tomorrow: out Day) is
  -- This is implementation (a) of NextDay, in which Day'Succ is used
```

```
  begin
    if Today = Sun then
      Tomorrow := Mon;
    else
      Tomorrow := Day'Succ(Today);
    end if;
  end NextDay_A;

  procedure NextDay_B(Today: in Day; Tomorrow: out Day) is
  -- This is implementation (b) of NextDay, in which a case-statement is used
  begin
    case Today is
      when Mon => Tomorrow := Tue;
      when Tue => Tomorrow := Wed;
      when Wed => Tomorrow := Thu;
      when Thu => Tomorrow := Fri;
      when Fri => Tomorrow := Sat;
      when Sat => Tomorrow := Sun;
      when Sun => Tomorrow := Mon;
    end case;
  end NextDay_B;

end T1Q3;
```

GNATprove is able to discharge all VCs for this example.

If VCs are generated for the equivalent SPARK 2005 code then the SPARK Simplifier discharges all VCs except one (for NextDay_A). The remaining VC is discharged by SPARKbridge (Victor + Alt-Ergo).

An alternative way of specifying the postconditions is used in the package T1Q3_Alt. Instead of implication (SPARK 2005) or "if then" (SPARK 2014 aspects) the postconditions are expressed in an equivalent form using only "and" and "or". For example, here is the specification of NextDay_B.

```
procedure NextDay_B(Today: in Day; Tomorrow: out Day)
 with Post => ((if (Today=Mon) then Tomorrow=Tue) and
  (if (Today=Tue) then Tomorrow=Wed) and
  (if (Today=Wed) then Tomorrow=Thu) and
  (if (Today=Thu) then Tomorrow=Fri) and
  (if (Today=Fri) then Tomorrow=Sat) and
  (if (Today=Sat) then Tomorrow=Sun) and
  (if (Today=Sun) then Tomorrow=Mon));
--# derives Tomorrow from Today;
--# post (Today=Mon -> Tomorrow=Tue) and
--#      (Today=Tue -> Tomorrow=Wed) and
--#      (Today=Wed -> Tomorrow=Thu) and
--#      (Today=Thu -> Tomorrow=Fri) and
--#      (Today=Fri -> Tomorrow=Sat) and
--#      (Today=Sat -> Tomorrow=Sun) and
--#      (Today=Sun -> Tomorrow=Mon);
```

As with the previous version, all VCs are discharged by GNATprove. This time, the Simplifier is able to discharge all SPARK 2005 VCs without the aid of SPARKbridge.

3.6.4 Integer Square Root

The procedure ISQRT calculates the Integer Square Root of a Natural number. Sometimes postconditions can mirror the code quite closely but this is a nice example of where the postcondition looks very different from the implementation.

Note the use of pragma Loop_Invariant which is the equivalent of the loop invariant assertion in SPARK 2005.

```
package T1Q4 is

  procedure ISQRT(N: in Natural; Root: out Natural)
    with Post => (Root*Root <= N and (Root+1)*(Root+1) > N);
  --# derives Root from N;
  --# post Root*Root <= N and
  --#      (Root+1)*(Root+1) > N;

end T1Q4;

package body T1Q4 is

   procedure ISQRT(N: in Natural; Root: out Natural) is
      -- Introduce a new subtype to use to avoid possible overflow
      -- of expression in loop exit statement.
      subtype Big_Natural is long_long_integer range 0..Long_Long_Integer'Last;

      Local_Root : Big_Natural;

   begin
      Local_Root := 0;

      loop
         exit when (Local_Root + 1) * (Local_Root + 1) > Big_Natural (N);

         Local_Root := Local_Root + 1;

       -- Loop Invariant is in terms of the incremented value of Local_Root.
       pragma Loop_Invariant
         (Local_Root * Local_Root <= Big_Natural(N)
          and then Local_Root <= Big_Natural(Natural'Last));
       --# assert Local_Root * Local_Root <= Big_Natural(N) and
       --#        Local_Root <= Big_Natural(N);

      end loop;

      Root := Natural(Local_Root);
   end ISQRT;
end T1Q4;
```

All the VCs are discharged automatically by the SPARK Simplifier, so long as a configuration file is provided to inform the Examiner of the size of Long_Long_Integer when generating VCs.

Note: Observation 10: The SPARK 2005 tools use a configuration file to specify properties of the target such as size of integer. By default GNATprove assumes that the configuration of the target is the same as the host, but this can be overridden and a specific target configuration can be provided with the switch -gnateT.

GNATprove initially discharged all VCs with the exception of those for the Post aspect on the subprogram specification. As aspects are executable there is a potential for overflow in the expressions "Root*Root" and "(Root+1)*(Root+1)". To avoid this it is necessary to instruct the compiler to eliminate intermediate overflows in assertions by using arbitrary precision arithmetic as required. There are two methods of doing this, via the compiler switch "-gnato" or via the configuration pragma "Overflow_Checks". These are described in the GNAT Pro User's Guide. For the examples in this study the option "-gnato13" was specified via GPS: Project > Edit Project Properties > Switches > Ada. The option "-gnato13" specifies strict checks in general (the default) but eliminates the possibility of overflow in assertions by using arbitrary precision. With this option all VCs are discharged by GNATprove.

3.6.5 Bounded_Addition

This subprogram adds two Integers and truncates the result if it goes out of range. The postconditions in SPARK 2005 and SPARK 2014 are very similar. As in earlier examples, SPARK 2005 uses implication "->" where the

SPARK 2014 aspect uses "if then".

```
package T1Q5 is

  procedure Bounded_Add(X,Y: in Integer; Z: out Integer)
    with Post => ((if (Integer'First <= X+Y and X+Y <= Integer'Last) then Z=X+Y)
    and (if (Integer'First > X+Y) then Z=Integer'First)
    and (if (X+Y > Integer'Last) then Z=Integer'Last));
  --# derives Z from X, Y;
  --# post ((Integer'First <= X+Y and X+Y <= Integer'Last) -> Z=X+Y)
  --#  and ((Integer'First >  X+Y)                          -> Z=Integer'First)
  --#  and (                          (X+Y >  Integer'Last) -> Z=Integer'Last);

end T1Q5;

package body T1Q5 is

  procedure Bounded_Add(X,Y: in Integer; Z: out Integer) is
  begin
    if X < 0 and Y < 0 then -- both negative

      if X < Integer'First - Y then
        Z := Integer'First;
      else
        Z := X + Y;
      end if;

    elsif X > 0 and Y > 0 then -- both positive

      if X > Integer'Last - Y then
        Z := Integer'Last;
      else
        Z := X + Y;
      end if;

    else -- one +ve, one -ve: must be safe to add them

      Z := X + Y;

    end if;
  end Bounded_Add;

end T1Q5;
```

GNATprove discharges all the VCs for this example.

The SPARK Simplifier discharges all but one of the VCs generated by the Examiner for the SPARK 2005 version of the code. If SPARKsimp is run with the -victor option to invoke SPARKbridge then the remaining VC is discharged by Victor + Alt-Ergo.

3.6.6 Raise to Power

This procedure raises an Integer to a specified power, returning the result as an Integer. The precondition ensures that the result will be within the bounds of the Integer type. The implementation uses a loop, for which the invariant is specified in SPARK 2005 (as an assertion) and SPARK 2014 (as a pragma Loop_Invariant).

```
package T1Q6 is

  procedure Raise_To_Power(X: in Integer; Y: in Natural; Z: out Integer)
    with Pre => (X ** Y in Integer),
    Post => (Z = X ** Y);
  --# derives Z from X, Y;
```

```
   --# pre  X ** Y in Integer;
   --# post Z = X ** Y;

end T1Q6;

package body T1Q6 is

   procedure Raise_To_Power(X: in Integer; Y: in Natural; Z: out Integer) is
      A, C: Integer;
      B: Natural;
   begin
      A := X;
      B := Y;
      C := 1;
      while B > 0
      --# assert C*(A**B) = X**Y and
      --#   X**Y in Integer;
      loop
         pragma Loop_Invariant ((C*(A**B) = X**Y) and
                                (X**Y in Integer));
         if B mod 2 = 0 then
            B := B / 2;
            A := A * A;
         else
            B := B - 1;
            C := C * A;
         end if;
      end loop;
      Z := C;
   end Raise_To_Power;

end T1Q6;
```

GNATprove is unable to discharge the VCs for the loop invariant and for the overflow checks on the assignments "A := A * A;" and "C := C * A;".

The SPARK 2005 tools are similarly unable to discharge the SPARK VCs for the loop invariant and for the same two assignments, even when SPARKbridge is employed.

Note: Observation 11: The Raise_To_Power example illustrates that VCs involving non-linear arithmetic are typically hard to prove, both for the SPARK 2005 and the SPARK 2014 toolsets. The Riposte counter-example finding tool was applied to the SPARK 2005 VCs in an attempt to clarify whether they are actually non-provable or just hard, but it reported COMPLEXITY_EXPLOSION and could not reach a verdict.

Note: Recommendation 05: The Raise_To_Power example illustrates a difference between the SPARK 2005 and SPARK 2014 toolsets. GNATprove helps the user by highlighting the lines that are not proved but does not provide the user with full details of the VC. With SPARK 2005 the user must look at the VCs in order to see exactly what cannot be proved. This is one level of indirection away from the code so in that sense it is less user-friendly. However, the VCs also show precisely what hypotheses are available to the prover which can be very useful when debugging proof attempts, especially for advanced users. It is recommended that some way is found to make the the SPARK 2014 VCs more accessible to users.

3.7 Array Examples

3.7.1 Swap two elements of an array

This procedure swaps the values of two elements in an array. The postcondition makes use of quantifiers to specify that all the other elements of the array are preserved.

```
package T2Q1a is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

  procedure Swap (A: in out ArrayType; I, J: in IndexType)
    with Post => (A(I) = A'Old(J) and A(J) = A'Old(I) and
                    (for all N in IndexType => (if (N/=I and N/=J) then A(N) = A'Old(N))));
  --# derives A from A, I, J;
  --# post A(I) = A~(J) and A(J) = A~(I) and
  --#      (for all N in IndexType => ((N/=I and N/=J) -> A(N) = A~(N)));

end T2Q1a;

package body T2Q1a is

  procedure Swap (A: in out ArrayType; I, J: in IndexType) is
    T: ElementType;
  begin
    T    := A(I);
    A(I) := A(J);
    A(J) := T;
  end Swap;

end T2Q1a;
```

GNATprove discharges all VCs for this subprogram. The SPARK Simplifier discharges all the VCs for the SPARK 2005 version of this example.

It is observed that SPARK 2005 provides a shorthand notation for array updates where specified elements are updated and the other elements are preserved, so an alternative method for specifying the postcondition in SPARK 2005 is:

```
procedure Swap (A: in out ArrayType; I, J: in IndexType);
--# derives A from A, I, J;
--# post A = A~[I => A~(J); J => A~(I)];
```

There is also such a notation in SPARK 2014, the 'Update attribute, but it is not yet supported by GNATprove.

3.7.2 Clear Array

This subprogram loops over the elements of an array, setting each one to zero. There is no postcondition - the aim is simply to prove freedom from runtime exceptions.

```
package T2Q2 is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

  procedure Clear (A: in out ArrayType; L,U: in IndexType);
  --# derives A from A, L, U;

end T2Q2;

package body T2Q2 is

  procedure Clear (A: in out ArrayType; L,U: in IndexType) is
  begin
    for I in IndexType range L..U loop
      A(I) := 0;
```

```
    end loop;
  end Clear;

end T2Q2;
```

Both GNATprove and the SPARK 2005 tools generate and discharge all VCs successfully for this example.

3.7.3 Clear array with postcondition

Now we return to the Clear subprogram seen earlier but this time a postcondition has been added, so the proof tools can check this against the implementation. Note that the Loop_Invariant pragma makes use of the attribute A'Loop_Entry to refer to the value of A on entry to the loop.

```
package T2Q4 is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

  procedure Clear (A: in out ArrayType; L,U: in IndexType)
    with Post => ((for all N in IndexType range L..U => (A(N) = 0)) and
                    (for all N in IndexType => (if (N<L or N>U) then A(N) = A'Old(N))));
  --# derives A from A, L, U;
  --# post (for all N in IndexType range L..U => (A(N) = 0)) and
  --#       (for all N in IndexType => ((N<L or N>U) -> A(N) = A~(N)));

end T2Q4;

package body T2Q4 is

  procedure Clear (A: in out ArrayType; L,U: in IndexType) is
  begin
    for I in IndexType range L..U loop
      pragma Loop_Invariant ((for all N in IndexType range L..(I-1) => (A(N) = 0)) and
                      (for all N in IndexType => (if (N<L or N>=I) then (A(N) = A'Loop_Entry(N))
      A(I) := 0;
      --# assert (for all N in IndexType range L..I => (A(N) = 0)) and
      --#         (for all N in IndexType => ((N<L or N>I) -> A(N) = A~(N))) and
      --#         U = U% and L <= I;
    end loop;
  end Clear;

end T2Q4;
```

Both GNATprove and the SPARK 2005 tools generate and discharge all VCs successfully for this example.

3.7.4 Find largest element

In this example, students on the SPARK 2005 course are asked to write and prove a subprogram which finds the largest element in an array. Seven variations on the solution are provided, some using quantifiers and some using proof functions.

In the SPARK 2005 version of the code a proof function The_Max is declared thus:

```
--# function The_Max(A: ArrayType;
--#                  L, U: IndexType) return ElementType;
--# return Max => (for all N in IndexType range L..U => (A(N) <= Max))
--#   and  (for some N in IndexType range L..U => (A(N) = Max));
```

This can be used in proof annotations to represent "a function that returns the largest element in the array" which is a useful shorthand as an alternative to writing the definition in terms of quantifiers whenever it is needed.

There is no SPARK 2014 equivalent of the proof function, so an executable function The_Max has been defined, with an equivalent postcondition aspect. This function can be used in assertions as required and is not intended to be called in general code. All VCs for The_Max are discharged by GNATprove.

The first solution is named MaxElement_P1B1. This does not make use of The_Max, but instead uses quantifiers to explicitly specify its behaviour in the postcondition and the loop invariant. All VCs for the SPARK 2014 version of the code are discharged by GNATprove, and all VCs for the SPARK 2005 version are discharged by the SPARK 2005 tools (Simplifier plus SPARKbridge).

The procedure named MaxElement_P2B1 makes use of the function The_Max in its postcondition. The postcondition is clearly much more concise, because the detail is factored out into The_Max. Similarly, The_Max is used in the Loop_Invariant. GNATprove is currently unable to prove this loop invariant.

Note: Observation 12: GNATprove is unable to prove the loop invariant because The_Max is not an expression function and so its postcondition is not propagated to VCs for expressions where it is used. This will be addressed in a future version of GNATprove [M322-027].

MaxElement_P3B1 has no loop invariant specified and there is no postcondition on its specification, so there is nothing to prove other than absence of runtime exceptions. GNATprove and the SPARK 2005 tools prove all VCs for their respective versions of this subprogram.

MaxElement_P1B2 is similar to MaxElement_P1B1 but the loop invariant has been relocated to the beginning of the loop and modified accordingly. All VCs are discharged automatically by GNATprove and by the SPARK 2005 tools.

MaxElement_P2B2 has the loop invariant in the same place as P1B2 but it uses the call to The_Max instead of the explicit expression using quantifiers. GNATprove is unable to prove this invariant as noted above.

MexElement_P3B2 uses a slightly more efficient implementation of the algorithm which reduces the number of loop iterations by one. There is no loop invariant or postcondition and all VCs are discharged by GNATprove and the SPARK 2005 tools for their respective implementations.

3.7.5 Sum elements of array

SumArray sums the elements of an array. SumArray_Shift is similar, and has the same postcondition, but a 'Shift' parameter has been added to make the implementation and proof more complex. Rather than initialising the sum to zero it is initialised to Shift, but the value of Shift is then subtracted again in the body of the loop so Shift has no impact on the end result (provided it doesn't cause or prevent an overflow).

The SPARK 2005 solution makes use of a proof function Sum_Between and this has been modelled as an executable function in the SPARK 2014 version. All VCs are discharged by the SPARK 2005 tools.

Note: Observation 13: With the default timeout of 1s GNATprove is unable to prove the two loop invariants. It also leaves the range check undischarged on line 14 of the package specification (in the expression function Sum_Between). Increasing the timeout from 1s to 15s enables GNATprove to successfully discharge the VCs for the loop invariants - the total time for this is 37s. The range check is actually unprovable as it stands. To discharge it requires strengthening the postcondition on Sum_Between, but that would prevent it from being used in the proof of the loop invariants so this will be deferred until completion of L525-024.

3.7.6 Find

In this example the subprogram Find searches through an array looking for a particular value and returning two exports: a Boolean Found which is true if (and only if) the required value is present within the array, and an Index which gives the location of such a value if it is present. The postcondition specifies the desired behaviour.

```
package T2Q7 is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

  procedure Find (A: in ArrayType; Value: in ElementType;
```

```
                  Found: out Boolean; Index: out IndexType)
    with Post => ((Found = (for some N in IndexType => (A(N) = Value))) and
                    (if Found then (A(Index) = Value and
                                    (for all N in IndexType range 1..Index-1 => (A(N) /= Value)))
                    (if (not Found) then Index = IndexType'Last));
  --# derives Found, Index from A, Value;
  --# post (Found <-> (for some N in IndexType => (A(N) = Value))) and
  --#       (Found -> (A(Index) = Value and
  --#                  (for all N in IndexType range 1..Index-1 => (A(N) /= Value)))) and
  --#       (not Found -> Index = IndexType'Last);

end T2Q7;

package body T2Q7 is

  procedure Find (A: in ArrayType; Value: in ElementType;
                  Found: out Boolean; Index: out IndexType) is
  begin
    Index := IndexType'First;
    Found := False;
    loop
      pragma Loop_Invariant (not Found and
                      Index in IndexType'First..IndexType'Last and
                      (for all N in IndexType range 1..(Index-1) => (A(N) /= Value)));
      Found := A(Index) = Value;
      exit when Found or Index = IndexType'Last;
      --# assert not Found and
      --#        Index in IndexType'First..IndexType'Last-1 and
      --#        (for all N in IndexType range 1..Index => (A(N) /= Value));
      Index := Index + 1;
    end loop;
  end Find;

end T2Q7;
```

All VCs are proved by GNATprove and the SPARK 2005 tools for their respective implementations.

3.7.7 Array of Fibonacci numbers

The subprogram CreateFibArray populates a 32-element array of Positives with the first 32 numbers in the Fibonacci sequence. This is done by initializing the first two elements of the array to 1, then looping over the remaining elements, setting each element to the sum of the two preceding elements. (Note that the value of element 32 in the sequence is 2,178,309 which is well within the range of Positive on a 32-bit machine.)

Two versions of the subprogram are provided. The first, CreateFibArray, has a postcondition stating the required properties of the array. The second, CreateFibArray_RTConly has no postcondition so only needs to be proved free from runtime exceptions.

In the SPARK 2005 solution the proof function fib has been defined recursively and is used to specify the postcondition and the loop invariant for CreateFibArray. In the SPARK 2014 solution fib is an expression function.

```
package T2Q8 is

  subtype IndexType is Positive range 1..32;
  type FibArrayType is array (IndexType) of Positive;

  function fib (I: IndexType) return Positive is
     (if (I <= 2) then 1
     else (Fib(I-1) + Fib(I-2)));
  --# function fib (I: IndexType) return Positive;
  --# return Result => ((I <= 2 -> Result = 1) and
  --#                    (I > 2 -> Result = Fib(I-1) + Fib(I-2)));
```

```
  procedure CreateFibArray (A: out FibArrayType)
    with Post => (for all N in IndexType => (A(N) = fib(N)));
  --# derives A from;
  --# post for all N in IndexType => (A(N) = fib(N));

  procedure CreateFibArray_RTConly (A: out FibArrayType);
  --# derives A from;

end T2Q8;

package body T2Q8 is

  procedure CreateFibArray (A: out FibArrayType) is
  begin
    A := FibArrayType'(others => 1);
    pragma Assert (A(1) = fib(1) and A(2) = fib(2));
    --# assert A(1) = fib(1) and A(2) = fib(2);
    for I in IndexType range 3..32 loop
      pragma Loop_Invariant (A(1) = 1 and A(2) = 1 and I >= 3 and
                             fib(1) = 1 and fib(2) = 1 and
                             (for all N in IndexType range 3..(I-1) =>
                                (A(N) = fib(N) and A(N) >= 1 and
                                   A(N) <= 2**(N-2))));
      A(I) := A(I-1) + A(I-2);
      --# assert A(1) = 1 and A(2) = 1 and I >= 3 and
      --#   fib(1) = 1 and fib(2) = 1 and
      --#   (for all N in IndexType range 3..I =>
      --#      (A(N) = fib(N) and A(N) >= 1 and
      --#         A(N) <= 2**(N-2)));
    end loop;
  end CreateFibArray;

  procedure CreateFibArray_RTConly (A: out FibArrayType) is
  begin
     A := FibArrayType'(others => 1);
    pragma Assert (A(1) = fib(1) and A(2) = fib(2));
     --# assert A(1) = fib(1) and A(2) = fib(2);
    for I in IndexType range 3..32 loop
      pragma Loop_Invariant (A(1) = 1 and A(2) = 1 and I >= 3 and
                             fib(1) = 1 and fib(2) = 1 and
                             (for all N in IndexType range 3..(I-1) =>
                                (A(N) >= 1 and A(N) <= 2**(N-2))));
      A(I) := A(I-1) + A(I-2);
      --# assert A(1) = 1 and A(2) = 1 and I >= 3 and
      --#   fib(1) = 1 and fib(2) = 1 and
      --#   (for all N in IndexType range 3..I =>
      --#      (1 <= A(N) and A(N) <= 2**(N-2)));
    end loop;
  end CreateFibArray_RTConly;

end T2Q8;
```

GNATprove is unable to prove the range and overflow checks for the expression function fib. It might be possible to address this with a suitable postcondition aspect but that would currently prevent the expression from appearing in the VCs for uses of fib (see L525-024 and M326-033).

GNATprove is also unable to prove the loop invariants (for both versions of the subprogram) and the range checks for the assignment statements in the loops.

For the SPARK 2005 version of the code the SPARK 2005 tools are similarly unable to prove the VCs for the loop invariants and the assignments, although in the course material from which this example is taken a manual argument is provided for those undischarged VCs.

In the SPARK 2005 version, fib is a proof function. As proof functions are not executable there are no runtime check VCs to be discharged for it.

The conclusions to be drawn from this example are similar to those already observed for the Raise_To_Power subprogram, namely that non-linear arithmetic is problematic for both the SPARK 2005 and SPARK 2014 proof tools (and indeed this is not a problem specific to SPARK but applies to provers in general!). As with Raise_To_Power, the SPARK 2014 interface is more user-friendly with respect to relating the proofs back to the source code, although the SPARK 2005 VCs have advantages when it comes to inspecting them to determine precisely what could not be proved and what hypotheses were given. This is beneficial when manual justifications have to be provided for provable VCs which defy the current prover technologies.

3.8 Further Advanced SPARK course examples

This section looks at further examples from the Advanced SPARK Program Design and Verification course.

3.8.1 Rotate3

This subprogram takes an array and three indices as parameters and rotates the array elements specified by the indices. The postcondition specifies the rotation which must be performed. The implementation of Rotate3 is done by making successive calls to a Swap routine. The aim is to explore the capabilities of the proof system for cases where one subprogram's implementation meets its postcondition by making calls to other, suitably annotated, subprograms.

```
package T3Q1 is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

  procedure Rotate3(A: in out ArrayType; X, Y, Z: in IndexType)
    with Pre => (X /= Y and
                 Y /= Z and
                 X /= Z),
    Post => (A(X) = A'Old(Y) and A(Y) = A'Old(Z) and A(Z) = A'Old(X) and
               (for all N in IndexType => (if (N/=X and N/=Y and N/=Z) then A(N) = A'Old(N))));
  --# derives A from A, X, Y, Z;
  --# pre X /= Y and
  --#     Y /= Z and
  --#     X /= Z;
  --# post A(X) = A~(Y) and A(Y) = A~(Z) and A(Z) = A~(X) and
  --#      (for all N in IndexType => ((N/=X and N/=Y and N/=Z) -> A(N) = A~(N)));

end T3Q1;

package body T3Q1 is

  procedure Swap (A: in out ArrayType; I, J: in IndexType)
    with Post => (A(I) = A'Old(J) and A(J) = A'Old(I) and
                    (for all N in IndexType => (if (N/=I and N/=J) then A(N) = A'Old(N))));

  procedure Swap (A: in out ArrayType; I, J: in IndexType)
  --# derives A from A, I, J;
  --# pre  I /= J;
  --# post A(I) = A~(J) and A(J) = A~(I) and
  --#      (for all N in IndexType => ((N/=I and N/=J) -> A(N) = A~(N)));
  is
    T: ElementType;
  begin
    T    := A(I);
```

```
    A(I) := A(J);
    A(J) := T;
  end Swap;

  procedure Rotate3(A: in out ArrayType; X, Y, Z: in IndexType) is
  begin
    Swap(A, X, Y);
    Swap(A, Y, Z);
  end Rotate3;

end T3Q1;
```

The postconditions in SPARK 2005 and SPARK 2014 illustrate how array updates are expressed in each notation.

All VCs for Swap and Rotate3 are proved automatically by GNATprove and the SPARK 2005 toolset (using SPARKbridge) for their respective versions of the code.

3.8.2 Quadruple

The function Quadruple meets its specified postcondition by making successive calls to function Double. It is another case where the implementation of one subprogram depends on calls to another.

```
package T3Q2 is

  function Quadruple (X: in Natural) return Natural
    with Pre => (X <= Natural'Last/4),
    Post => (Quadruple'Result = 4 * X);
  --# pre    X <= Natural'Last/4;
  --# return 4 * X;

end T3Q2;

package body T3Q2 is

   function Double (X: in Natural) return Natural
     with Pre => (X <= Natural'Last/2),
     Post => (Double'Result = 2 * X);

   function Double (X: in Natural) return Natural
   --# pre    X <= Natural'Last/2;
   --# return 2 * X;
   is
   begin
      return 2 * X;
   end Double;

   function Quadruple (X: in Natural) return Natural is
      T : Natural;
   begin
      T := Double (Double (X));
      pragma Assert (T = 2 * (2 * X));
      return T;
   end Quadruple;

end T3Q2;
```

All VCs for Double and Quadruple are proved automatically by GNATprove and the SPARK 2005 toolset for their respective versions of the code.

3.8.3 DoNothing

This is a more complex example where the proof of the postcondition of the top level procedure, DoNothing, relies on the postconditions of its called subprograms. As its name suggests, DoNothing preserves the values of its parameters, but it does so via a series of intermediate transformations. The challenge for the proof system is to determine that the combination of these operations results in the parameters being set back to their original values.

```
package T3Q3 is

  procedure DoNothing (X, Y: in out Natural)
    with Pre => (Y > 0 and X >= Y),
    Post => (X = X'Old and Y = Y'Old);
  --# derives X from X & Y from Y;
  --# pre  Y > 0 and X >= Y;
  --# post X = X~ and Y = Y~;

end T3Q3;

package body T3Q3 is

  procedure A (I, J: in Natural; K: out Natural)
    with Pre => (I + J <= Natural'Last),
    Post => (K = I + J);

  procedure A (I, J: in Natural; K: out Natural)
  --# derives K from I, J;
  --# pre  I + J <= Natural'Last;
  --# post K = I + J;
  is
  begin
    K := I + J;
  end A;

  procedure M (I, J: in Natural; K: out Natural)
    with Pre => (I * J <= Natural'Last),
    Post => (K = I * J);

  procedure M (I, J: in Natural; K: out Natural)
  --# derives K from I, J;
  --# pre  I * J <= Natural'Last;
  --# post K = I * J;
  is
  begin
    K := I * J;
  end M;

  procedure D (I, J: in Natural; K, L: out Natural)
    with Pre => (J /= 0),
    Post => (K = I / J and L = I - K * J);

  procedure D (I, J: in Natural; K, L: out Natural)
  --# derives K, L from I, J;
  --# pre  J /= 0;
  --# post K = I / J and L = I - K * J;
  is
  begin
    K := I/J;
    L := I - K * J;
  end D;

  procedure DoNothing (X, Y: in out Natural) is
    P, Q, R: Natural;
  begin
```

```
    D(X, Y, Q, R);
    M(Q, Y, P);
    A(P, R, X);
    --# assert X = X~ and Y = Y~ and Q = X / Y and P = (X / Y) * Y;
    --# accept Flow, 10, R, "Value of R not required";
    D(P, Q, Y, R);
    --# end accept;
    --# check Y~ * (X~ / Y~) / (X~ / Y~) = Y~;

  --# accept Flow, 601, X, Y, "Overall result is that X is unchanged";
  --# accept Flow, 601, Y, X, "Overall result is that Y is unchanged";
  end DoNothing;

end T3Q3;
```

Neither GNATprove nor the SPARK 2005 tools are able to prove the postcondition for this example. It appears to fall into the same class as the Raise_To_Power and Fib examples seen earlier (VCs are probably provable but cannot be discharged with the current tools) and similar observations could be made. The model answers from the SPARK training course state that the SPARK 2005 VCs can be discharged with the SPARK Proof Checker (interactive prover) but this was not done for this case study.

3.8.4 SumArray

SumArray calculates the sum of the elements of an array. In the original SPARK 2005 version of this example a proof function, Summed_Between, is used in the return annotation in order to express the desired functionality. In the SPARK 2014 version this is modelled using the expression function Summed_Between.

```
package T3Q4 is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;
  subtype SumType is Natural range 0..100000;

  function Summed_Between(A: in ArrayType;
                          L,U: in IndexType) return SumType
   with Pre  => (L <= U),
        Post => (Summed_Between'Result <= (U - L+1) * 1000);

  function Summed_Between(A: in ArrayType;
                          L,U: in IndexType) return SumType is
     (if (L = U) then A(L)
     elsif (L < U) then Summed_Between(A, L, U-1) + A(U)
     else 0);
  --# function Summed_Between(A: in ArrayType;
  --#                         L,U: in IndexType) return SumType;
  --# pre L <= U;
  --# return Sum => (((L = U) -> Sum = A(L)) and
  --#                  ((L < U) -> Sum = Summed_Between(A, L, U-1) + A(U)));

  function SumArray (A: in ArrayType) return SumType;
  --# return Summed_Between(A, IndexType'First, IndexType'Last);

end T3Q4;

package body T3Q4 is

  function SumArray (A: in ArrayType) return SumType is
    Sum: SumType := 0;
  begin
    for I in IndexType loop
```

```
      pragma Loop_Invariant ((if I /= IndexType'First then Sum = Summed_Between(A, IndexType'First
        Sum <= 1000 * (I - IndexType'First));
      Sum := Sum + A(I);
      --# assert Sum = Summed_Between(A, IndexType'First, I) and
      --#        Sum <= 1000 * (I - IndexType'First + 1);
    end loop;
    return Sum;
  end SumArray;

end T3Q4;
```

All the VCs for this example are discharged by both GNATprove and the SPARK 2005 tools.

Note: Observation 14: It is interesting to note the differences in the function Summed_Between in the SPARK 2005 and SPARK 2014 versions of this example. In SPARK 2005 it appears as a proof function and in SPARK 2014 it is an expression function. Initially the expression function was given without a separate declaration but GNATprove was unable to prove the range check for the recursive call on the second line of the expression. This was addressed by adding a separate declaration with the necessary postcondition aspect, allowing all checks to be proved.

3.8.5 Sorting Algorithm

This example appears in *SPARK - the proven approach to high integrity software* by John Barnes. It consists of a procedure Sort declared together with a few related types in a package Array_Utilities. Two proof functions are used in the SPARK 2005 version - the function Ordered is true if the part of the array from L to U is in ascending order, and the function Perm is true if the set of values of the two array parameters are the same set with identical duplications if any. The example is discussed and refined within the book - the version shown here is based on the version shown in section 15.6 of the book.

```
package T3Q5 is

   Max_Table_Size : constant := 100;

   type Base_Index_Type is range 0 .. Max_Table_Size;
   subtype Index_Type is Base_Index_Type range 1 .. Max_Table_Size;

   type Contents_Type is range -1000 .. 1000;

   type Array_Type is array (Index_Type) of Contents_Type;

   --# function Ordered (A : in Array_Type) return Boolean;
   --# return for all I in Index_Type range Index_Type'First .. Index_Type'Pred (Index_Type'Last)
   --#   (A (I) <= A (I + 1));

   function Perm(A, B : Array_Type) return Boolean is
      ((for some I in Index_Type => (for some J in Index_Type => (B(I) = A(J) and B(J) = A(I) and
       (for all N in Index_Type => (if (N /= I and N /= J) then A(N) = B(N)))))));

   --# function Perm(A, B : Array_Type) return Boolean;
   --  return ((for some I in Index_Type => (for some J in Index_Type => (B = A[I => A(J); J => A
   --            (for some C in Array_Type => (Perm (A, C) and Perm (B, C))));

   procedure Sort (Table : in out Array_Type);
   --# derives Table from Table;
   --# post Ordered (Table) and Permutation (Table, Table~);

end T3Q5;

package body T3Q5 is

   procedure Sort(Table : in out Array_Type) is
```

```
Key : Index_Type;
Table_Old : constant Array_Type := Table;

function Find_Smallest
  (Arr  : in Array_Type;
   L, U : in Index_Type) return Index_Type
with Pre  => (L <= U),
     Post => (L <= Find_Smallest'Result and Find_Smallest'Result <= U and
                 (for all N in Index_Type range L .. U => (Arr(Find_Smallest'Result) <= Arr(
                 (for some N in Index_Type range L .. U => (Arr(Find_Smallest'Result) = Arr(

function Find_Smallest
  (Arr  : in Array_Type;
   L, U : in Index_Type) return Index_Type
--# pre L <= U;
--# return Smallest_Index =>
--#   (for all X in Index_Type range L .. U =>
--#      (Arr (Smallest_Index) <= Arr (X))) and
--#     Smallest_Index in L .. U;
is
   K : Index_Type;
begin
   K := L;
   --# assert K = L
   --#    and (for all X in Index_Type range L .. L =>
   --#           (Arr (K) <= Arr (X)));
   if L < U then
      for I in Index_Type range L + 1 .. U loop
         if Arr (I) < Arr (K) then
            K := I;
         end if;
         pragma Loop_Invariant (I >= L + 1 and I  <= U
                                and L < U
                                and (for all X in Index_Type range L .. I =>
                                       (Arr (K) <= Arr (X)))
                                and K in L .. U);
         --# assert I >= L + 1 and I <= U
         --#    and L = L%     and U = U%
         --#    and L < U
         --#    and (for all X in Index_Type range L .. I =>
         --#           (Arr (K) <= Arr (X)))
         --#    and K in L .. U;
      end loop;
   end if;
   return K;
end Find_Smallest;

procedure Swap_Elements(T : in out Array_Type;
                        I, J : in Index_Type)
  with Post => (T(I) = T'Old(J) and T(J) = T'Old(I) and
     (for all N in Index_Type => (if (N /= I and N /= J) then T(N) = T'Old(N)))
      and Perm(T,T'Old));

procedure Swap_Elements (T    : in out Array_Type;
                         I, J : in     Index_Type)
--# derives T from T,I,J;
--# post T = T~[I => T~(J);
--#             J => T~(I)]
--#  and Permutation (T, T~);
is
   Temp : Contents_Type;
begin
   Temp := T(I);
```

```
         T(I) := T(J);
         T(J) := Temp;
         --# accept W, 444, "This assumption uses the definition of a permutation:",
         --#                 "If we swap any two elements, the array is a permutation",
         --#                 "of itself.";
         --# assume Permutation (T~[I => T~(J); J => T~(I)], T~); -- Note use of assume anno!
         --# end accept;
      end Swap_Elements;

   begin
      --# accept W, 444, "If two arrays are exactly the same, then they are also",
      --#                "(trivial) permutations of each other.";
      --# assume (Table = Table~) -> Permutation (Table, Table~);
      --# end accept;
      for Low in Index_Type range Index_Type'First .. Max_Table_Size - 1 loop

         Key := Find_Smallest (Table, Low, Max_Table_Size);
         pragma Assert
           (for all I in Index_Type range Low .. Max_Table_Size =>
              (Table (Key) <= Table (I)));
         if Key /= Low then
            Swap_Elements (Table, Low, Key);
            pragma Assert
              (for all I in Index_Type range Low .. Max_Table_Size =>
                 (Table (Low) <= Table (I)));
         end if;
         pragma Assert
           (for all I in Index_Type range Low .. Max_Table_Size =>
              (Table (Low) <= Table (I)));
         pragma Loop_Invariant
           ((for all I in Index_Type range 1 .. Low =>
              (Table (I) <= Table (I + 1))) and
              (for all I in Index_Type range Low .. Max_Table_Size =>
                 (Table (Low) <= Table (I))));
         --# assert (for all I in Index_Type range Index_Type'First .. Low =>
         --#            (Table (I) <= Table (I + 1)))
         --#    and (for all I in Index_Type range Low .. Index_Type'Last =>
         --#            (Table (Low) <= Table (I)))
         --#    and Permutation (Table, Table~);
      end loop;
   end Sort;

end T3Q5;
```

Note: Observation 15: In order to complete the SPARK 2005 version of this proof it was necessary to supply the Simplifier with the following user-defined proof rule.

```
permutation_is_transitive(1): permutation(A, C) may_be_deduced_from
  [ permutation(A, B), permutation(B, C),
    goal(checktype(A, array_type)),
    goal(checktype(B, array_type)),
    goal(checktype(C, array_type)) ].
```

This user rule provides a definition for the proof function Perm. Such definitions would normally be provided directly in the source code via –# return annotations as the user has originally attempted to do in this case.

```
--# function Perm(A, B : Array_Type) return Boolean;
--  return ((for some I in Index_Type => (for some J in Index_Type => (B = A[I => A(J); J => A(I)
--             (for some C in Array_Type => (Perm (A, C) and Perm (B, C))));
```

If this definition is converted from a comment to an annotation it is rejected by the Examiner because the existential quantifier "for some C in Array_Type" requires Array_Type to be scalar. It should also be noted that two instances of the –# assume annotation were used in the SPARK 2005 version of the proof. The expression given by the

assume annotation is given to be true and does not need to be proved. Therefore it must be used with care and causes the Examiner to generate a warning, which should be justified with an –# accept annotation as has been done here.

```
--# accept W, 444, "This assumption uses the definition of a permutation:",
--#                "If we swap any two elements, the array is a permutation",
--#                "of itself.";
--# assume Permutation (T~[I => T~(J); J => T~(I)], T~);
--# end accept;

--# accept W, 444, "If two arrays are exactly the same, then they are also",
--#                "(trivial) permutations of each other.";
--# assume (Table = Table~) -> Permutation (Table, Table~);
--# end accept;
```

With these two assumptions and the user-rule all VCs are discharged by the SPARK 2005 tools. For the code as shown, all VCs are discharged by GNATprove. Note however that there is currently no postcondition on the top-level procedure Sort so its correctness is not being shown by GNATprove. Further work is needed to specify and prove a suitable postcondition in SPARK 2014.

Note: Observation 16: In order to get GNATprove to prove all the VCs for T3Q5.Sort the option -proof=path_wp was specified. This causes GNATprove to generate a VC for each path to each check. It can be useful to try this option when GNATprove is unable to prove all checks, although it generally increases the time taken so it is not recommended as the default.

3.9 Tokeneer

The final example in this case study is Tokeneer, a highly secure biometric software system that was originally developed by Altran in SPARK 2005. The Tokeneer project was commissioned by the US National Security Agency (NSA) to demonstrate the feasibility of developing systems to the level of rigour required by the higher assurance levels of the Common Criteria. The development artefacts, including all source code, are now publicly available. For more details see www.adacore.com/sparkpro/tokeneer.

Tokeneer is significantly larger than the programs seen so far in this report, and provides a representative example that is closer to a real industrial development. The core system consists of approximately 10,000 lines of SPARK 2005 code (declarations and executable lines, excluding blank lines, comments and SPARK annotations). There are also approximately 3,700 lines of supporting code written in Ada which mimicked the drivers to peripherals connected to the core system. Performing a full conversion of the entire code base to SPARK 2014 and making the necessary changes to discharge all the VCs with GNATprove is outside the scope of this study. Two approaches to the translation to SPARK 2014 were considered:

- The first option was to use an automated translation script, which converts the majority of SPARK 2005 constructs to their SPARK 2014 equivalents and leaves the remainder to be converted manually. This was the approach that was originally planned for the case-study.
- Option two is to perform the conversion manually. This was initially not favoured due to the effort involved. However, such a translation had already been performed by AdaCore, making this approach more favourable.

Given the presence of an existing translation, option two was chosen. The translation had been performed some time ago and was not up-to-date with the latest developments in SPARK 2014 and GNATprove, so additional work was required to push it through the tools. On a system of this size it was not feasible to investigate the proof of all the code in detail. In addition, Because some language features are not yet implemented in SPARK 2014 (most notably abstract state and refined contracts), to analyse the whole of Tokeneer would require a significant rewriting and deviation from the desired design. Therefore, we adopted the following approach:

1. Analysis of the entire code base after minimal almost automatic translation of the SPARK 2005 code, leaving non-trivially translated language features as comments. This involved making any modifications necessary to allow as much of the code as possible to be analysed (i.e. not rejected) by GNATprove, and looking at the statistics from that analysis on the understanding that more work could be done to improve the provability of the code (in terms of both speed and success rate) given sufficient time.

2. A number of smaller studies, on selected smaller parts of the Tokeneer code, dedicated to evaluating specific features and goals of Hi-Lite that are additional to, or complementary to, SPARK 2005. These were:
    - A more complete manual translation of one package (*auditlog*) for evaluation of completeness (number of proved VCs), timings, and usability of GPS/GNATprove.
    - Evaluation of *executable assertions* semantics and the following requirement in Hi-Lite to verify that these are free of run-time exceptions, and asking the questions: How many extra VCs do we get using executable semantics (SPARK 2014) compared to mathematical semantics (SPARK 2005)? Do these extra VCs contribute to specification validation or do we get more false alarms?
    - Evaluation of the different semantics for overflows in annotations, counting generated and proved VCs for the different options.

The results of this process are presented in the following sections. As with the other case study examples there are more detailed statistics in the appendix of this report.

3.9.1 Stage 1 - analysis of all core code

As explained above, the first step was to analyse all of the core Tokeneer code with both the SPARK 2005 tools and GNATprove. Analysis with the SPARK 2005 tools was straightforward as the code was originally developed in SPARK 2005 and continually analysed throughout the development process (in line with Altran's development approach of "Correctness by Construction"). On the other hand the conversion to SPARK 2014 and analysis with GNATprove was, by necessity, retrospective. The results of this process and some observations that were raised during the translation are discussed below.

- As with previous examples in this report, SPARK 2005 proof functions were converted into executable functions in SPARK 2014 - indeed this is the main difference compared to the option of translating the code automatically with a script that was discussed earlier.
- Tagged record types are used in Tokeneer but are not currently supported by SPARK 2014. To maximise the amount of Tokeneer code that could be analysed by GNATprove, a manual conversion from tagged record types to ordinary records was performed. This was fairly straightforward - wherever a record extended a tagged record it was replaced with an ordinary record containing all the fields from the parent type, plus the extra fields from the extension. Where inherited functions for parent types were used these were replaced by normal functions with equivalent behaviour. This translation was performed solely to get as much of the Tokeneer code as possible into the current SPARK 2014 subset - in future it is expected that tagged types will be supported in SPARK 2014 so this translation would not be required.
- When the original translation was done the pragmas "Assert_And_Cut" and "Loop_Invariant" did not exist so SPARK 2005 assertions and loop invariants had all been modelled as pragma "Assert". For this study they were changed to "Assert_And_Cut" and "Loop_Invariant" as appropriate.
- The original translation used pragmas to specify preconditions and postconditions rather than using aspects. These are equivalent, and although aspects are generally preferred over pragmas they have been left in the form of pragmas for this study as they illustrate this possibility in contrast to the earlier case study examples.

Note: Observation 17: For some subprograms there is a pragma providing a precondition or postcondition on the subprogram specification, and a second pragma giving a refined version of the precondition or postcondition on the subprogram body. These refined preconditions and postconditions are currently ignored by GNATprove but will be analysed in future. For an example see the procedure AuditLog.ClearLogEntries.

Another difference between Tokeneer and the earlier examples in this case study is that not all of the core Tokeneer code is written in the SPARK (2005 or 2014) subset. In SPARK 2005 this is handled by providing a SPARK specification to the non-SPARK code, analysing that specification (thus enabling it to be used for the modular verification of dependent units) but not analysing the body containing the offending non-SPARK code. In SPARK 2005, the analysis of non-SPARK code can be suppressed by using the –# hide annotation to exclude the private part of a package specification, a package body, a subprogram body or an exception handler. Alternatively, analysis of an entire compilation unit such as a package body or a separate subprogram body can be suppressed by simply not presenting it to the SPARK 2005 tools for analysis. This method was used for some of the original Tokeneer code, such as the body of package Clock.Interface_P. For SPARK 2014 GNATprove takes a different approach in that non-SPARK code does not need to be explicitly "hidden" from analysis, but the tools detect whether the Ada

code supplied is in the SPARK subset or not. At present this behaviour is controlled by the –mode switch which can be set to detect, force, flow or prove. By default, non-SPARK code is ignored but if mode=force then non-SPARK code is rejected with an error. It is understood that this will change in future and non-SPARK code will be rejected unless it is explicitly labelled as not SPARK via a pragma. The details of this are still being finalised.

Note: Observation 18: We were concerned that the presence of a non-SPARK construct (access type) in file.ads might prevent the analysis of other units that depended on this unit or this type. This concern arose because in SPARK 2005 the access type would have represented a syntax error which, if not hidden, would have prevented the analysis of any other unit which depended on file.ads. Some experimentation with GNATprove determined that the presence of the offending construct does not prevent the analysis of any dependent units.

Note: Recommendation 06: When initially analysing Tokeneer from within GPS we noticed that GPS was launching editor windows for files as the analysis proceeded, but that it only did this for approximately half of the files being analysed. We initially wondered if this was because it had failed to analyse the other units, but we then realised that it was only launching windows for files where there was something of interest to report. It is recommended that the tools provide more positive confirmation for the user showing that all files had been successfully analysed [M415-023].

The SPARK 2005 version of Tokeneer had a total of 16 .adb files that could not be Examined. Most of these files called non-Ada code that interfaced with external devices. After the conversion, 13 of these files had to be placed in a separate directory so as for GNATprove to proceed with the analysis. When the bodies were present, the tools tried to analyse them and after failing (due to missing dependencies), the analysis stopped. After these files were removed, the tools used the specification files, which were in SPARK, in order to proceed with the analysis. It is worth mentioning that 3 of the 16 non-SPARK files did not have to be removed in order for the analysis to proceed. These 3 packages had no missing file dependencies.

Note: Observation 19: When GNATprove encounters a package body containing compilation errors it prevents the analysis of other units that depend on the corresponding package specification. On the other hand, if the package body is missing altogether then dependent units can be analysed.

Another idea we considered was to attempt the GNATprove analysis of the 'support' packages which are not in SPARK and which were excluded from the SPARK 2005 analysis. However this analysis was not possible due to missing dependencies on other units and we did not devote effort to trying to resolve these issues.

On a related topic, there is a GPS command "Prove > Show Unprovable Code" which invokes GNATprove with the options "–mode=detect -f". This analyses the files in the project and produces a report detailing what could not be analysed and why. A snippet of this report is copied below:

```
...
tcpip.adb:56:09: warning: component type is not in SPARK
tcpip.adb:57:13: warning: type is not in SPARK
tcpip.adb:57:30: warning: type used is not in SPARK
tcpip.adb:57:30: warning: type is not in SPARK
tcpip.adb:58:34: warning: type used is not in SPARK
...
```

This is followed by some summary statistics which give a useful high-level view of how much of the code could be analysed:

```
...
tokenapi.ads:142:04: info: attribute definition clause is not yet supported
**********************************
Subprograms in SPARK      :  78% (465/600)
  ... already supported   :  73% (436/600)
  ... not yet supported   :   5% ( 29/600)
Subprograms not in SPARK  :  23% (135/600)

Subprograms not in SPARK due to (possibly more than one reason):
 exception                :  17% (102/600)
 access                   :   6% ( 33/600)
 impure function          :   4% ( 24/600)
 dynamic allocation       :   1% (  6/600)
 tasking                  :   1% (  4/600)
 unchecked conversion     :   0% (  1/600)
```

```
Subprograms not yet supported due to (possibly more than one reason):
 concatenation            :   7% ( 43/600)
 attribute                :   1% (  7/600)
 arithmetic operation     :   1% (  6/600)
 operation on arrays      :   0% (  2/600)

Units with the largest number of subprograms in SPARK:
 enclave                  : 100% (37/37)
 configdata               :  94% (33/35)
 tokenreader              : 100% (23/23)
 crypto                   :  59% (23/39)
 auditlog                 :  81% (22/27)
 screen                   : 100% (19/19)
(...)

Units with the largest number of subprograms not in SPARK:
 spark_io                 :  65% (28/43)
 file                     :  79% (22/28)
 crypto                   :  41% (16/39)
 certproc                 :  63% (15/24)
 tokenapi                 :  92% (11/12)
 tcpip                    :  54% (7/13)
(...)
**********************************
Statistics above are logged in gnatprove.out
```

Note: Observation 20: The failure to analyse a package body (which does exist) appears to render the corresponding package specification unavailable for analysis. But if the package body is removed altogether then the corresponding specification can be used in the analysis of units which depend on it. This is because GNATprove will not analyse a unit unless each unit on which it depends has a specification and either a body which can be successfully compiled or else the body is absent. This is a significant difference from SPARK 2005, in which analysis of package bodies is totally independent from analysis of units that depend on their specifications. ([M422-020] covers the internal discussion of this issue.)

Note: Recommendation 07: In some cases there are VCs which cannot be proved because they are false. There are no known instances of such VCs in the final code presented in this report but instances of false VCs did arise when mistakes were made whilst working on the examples. In SPARK 2005 the Simplifier will sometimes detect false VCs and report them as such, although it is more likely that they will be reported as unproven. The Riposte counter-example finding tool can be used to check unproven SPARK 2005 VCs and, if it determines that they are false, generate counter-examples to help the user understand why they are false. Our understanding is that, due to the technology used, GNATprove will always report false SPARK 2014 VCs as not proven rather than false. The ability to detect false VCs and generate counter-examples would be a useful extension to the capabilities of the SPARK 2014 tools and it is recommended that this be included in future. It is understood that such a feature is already planned.

Having performed the conversions to SPARK 2014 mentioned above, all the Tokeneer files were analysed from within GPS using the option "Prove > Prove All" which generated the following command line:

```
gnatprove -Ptest.gpr --ide-progress-bar --show-tag -U -j4 --report=all --timeout=5
```

The output from this command appears in the locations pane in GPS. We copied this to a file and used 'grep' to count the total number of VCs (count the occurrences of "proved"), unproved VCs (count the occurrences of "not proved") and used similar text filters to count subcategories of VCs as shown in the table below.

Note: Recommendation 08: The SPARK 2014 tools currently lack automated support for providing an overall proof status summary. This is important when analysing larger projects and it is recommended that such a feature be developed.

VCs associated with:	Proved	Not Proved	Total
Pre/Post-conditions	39	106	145
Assertions/Checks	21	5	26
Runtime Checks	192	44	236
Total	252	155	407

These figures are not representative of the level of proof that could eventually be achieved with the SPARK 2014 version of Tokeneer for (at least) two reasons.

1. They are from a 'first cut' translation from SPARK 2005 to SPARK 2014. If each unproven check was investigated in detail and modifications were made as necessary in an attempt to enable it to be proved then it is likely that many more checks would be proved. In particular, the SPARK 2005 proof functions have not generally been converted into SPARK 2014 equivalents.
2. A timeout of 5 seconds was specified. It is likely that more VCs would be proved with a higher timeout value. (Conversely, lowering the timeout from 5s to 1s reduces the analysis time from 6m49s to 3m42s but only reduces the number of VCs discharged by 2.)

To get a more representative example, the proof of the AuditLog package was examined in detail as described in the following section.

3.9.2 Stage 2 - proof of AuditLog

The AuditLog package was selected for a more detailed investigation as this is the example used in the Tokeneer "Discovery" online tutorial (http://www.adacore.com/sparkpro/tokeneer/discovery/). The aim was to try and maximise the amount of this package that could be analysed and proved, making modifications as necessary in order to achieve this.

The following steps were performed:

- A number of subprograms were private to the body of auditlog and did not have a separate declaration in the package specification. Separate declarations were created for these subprograms so that precondition and postcondition aspects (or pragmas) could be added to them.
- Public subprograms already had separate declarations in the package specification, but in many cases these had preconditions and postconditions on their bodies but not on their specifications. In such cases the preconditions and postconditions were moved from the bodies to the specifications at the same time as translating them from SPARK 2005 to SPARK 2014. However, these preconditions and postconditions often referred to types and variables which were only visible in the body. The workaround for this was to move those type and variable declarations to the body which, unfortunately, makes them visible to other packages. For example, in the specification of procedure ClearLogEntries shown below the precondition refers to UsedLogFiles which was declared in the package body. Therefore when the precondition was moved from the package body to the package specification the declaration of UsedLogFiles was also moved from the package body to the package specification, as was the declaration of its type, LogFilesListT.

```
procedure ClearLogEntries (User    : in     AuditTypes.UserTextT);
--# global in     ConfigData.State;
--#        in     Clock.Now;
--#        in out FileState;
--#        in out State;
--# derives FileState,
--#          State     from FileState,
--#                         State,
--#                         ConfigData.State,
--#                         User,
--#                         Clock.Now;
pragma Precondition
   (UsedLogFiles.Length >= 1 and then
    NumberLogEntries =
      LogEntryCountT(UsedLogFiles.Length -1)*MaxLogFileEntries +
      LogFileEntries(CurrentLogFile));
pragma Postcondition
   (UsedLogFiles.Length >= 1 and then
    NumberLogEntries =
      LogEntryCountT(UsedLogFiles.Length -1)*MaxLogFileEntries +
      LogFileEntries(CurrentLogFile));
```

Having made these modifications GNATprove gave the following analysis results. (Note that this table was generated manually from the GNATprove output.)

VCs associated with:	Proved	Not Proved	Total
Pre/Post-conditions	10	6	16
Assertions/Checks	12	3	15
Runtime Checks	52	9	61
Total	74	18	92

There were 9 RTEF (runtime exception freedom) VCs left unproven from the previous stage. All of them were related to SPARK 2005 proof annotations that, after having been converted to their SPARK 2014 equivalents, had run-time checks associated with them (which is not the case for SPARK 2005 annotations).

All of these unproven RTEF VCs were related to showing that the expression "UsedLogFiles.Length -1" cannot underflow, which was part of an invariant about the logfiles and logentries. This underflow range check in specification terms correspond to unintentionally including a negative count of file entries (LogEntryCountT). After consulting the original design documents of Tokeneer and the original Tokeneer project leader it was concluded that another invariant indeed is that the UsedLogFiles always is non-empty. So in this case, the new semantics had allowed us to find a small but real issue with the specification that could be improved, which we did. Strengthening the preconditions to state that "UsedLogFile.Length >= 1" enables GNATprove to discharge all of these VCs but pushes the obligation onto the callers of the subprograms concerned, increasing the number of precondition and postcondition checks which cannot be proved.

VCs associated with:	Proved	Not Proved	Total
Pre/Post-conditions	16	5	21
Assertions/Checks	12	3	15
Runtime Checks	67	0	67
Total	95	8	103

These results were generated using the (default) option -gnato11 (discussed earlier in this report) which specifies that proof expressions obey normal Ada semantics and may cause intermediate overflows. We thought it would be interesting to repeat this proof using the -gnato13 option which specifies that arbitrary precision arithmetic should be used in proof expressions to eliminate the possibility of overflow in proofs, but uses the normal Ada semantics in general code.

Note: Observation 21: When analysing AuditLog, 38 additional VCs were generated when -gnato11 was specified, compared to when -gnato13 was specified. These represent 38 additional checks for overflow in proof expressions. However GNATprove was able to discharge all of the additional VCs, resulting in no overall difference to the number of undischarged VCs. This was somewhat unexpected and may be an indication of the high quality of the original proof annotations.

Note: Observation 22: As a piece of further work it would be interesting to investigate the remaining undischarged VCs for AuditLog and determine how best to tackle them.

3.10 Summary of Recommendations and Observations

This section collects the recommendations and observations from the body of the report.

3.10.1 Recommendations

Recommendation 01: It was sometimes found to be necessary to make implementation detail public for proof purposes when it would otherwise have been private (see Stacks and Tokeneer for examples). This loss of abstraction is undesirable and it is recommended that features are added to the SPARK 2014 language and tools to address this issue. (In fact such features are currently being designed.)

Recommendation 02: Sometimes variables are introduced for proof purposes only and they are not actually needed in general executable code. It is recommended (in both SPARK 2005 and SPARK 2014) that a mechanism be introduced for declaring "ghost variables" for use in proof only. The design of this feature is already underway in SPARK 2014.

Recommendation 03: SPARK 2005 uses external own variables to model inputs and outputs at the interface with the outside world, and the SPARK 2005 tools treat these variables as 'special' for flow analysis and proof. SPARK

2014 currently lacks support for modelling such variables, and it is recommended that such support be added. This will be dealt with using state abstractions (Abstract_State aspect).

Recommendation 04: The current inability to reference the *Old* and *Loop_Entry* attributes in assertions and loop invariants needs to be resolved. If the language rules cannot be relaxed then ghost variables may offer an acceptable solution to this issue.

Recommendation 05: The Raise_To_Power example illustrates a difference between the SPARK 2005 and SPARK 2014 toolsets. GNATprove helps the user by highlighting the lines that are not proved but does not provide the user with full details of the VC. With SPARK 2005 the user must look at the VCs in order to see exactly what cannot be proved. This is one level of indirection away from the code so in that sense it is less user-friendly. However, the VCs also show precisely what hypotheses are available to the prover which can be very useful when debugging proof attempts, especially for advanced users. It is recommended that some way is found to make the the SPARK 2014 VCs more accessible to users.

Recommendation 06: When initially analysing Tokeneer from within GPS we noticed that GPS was launching editor windows for files as the analysis proceeded, but that it only did this for approximately half of the files being analysed. We initially wondered if this was because it had failed to analyse the other units, but we then realised that it was only launching windows for files where there was something of interest to report. It is recommended that the tools provide more positive confirmation for the user showing that all files had been successfully analysed [M415-023].

Recommendation 07: In some cases there are VCs which cannot be proved because they are false. There are no known instances of such VCs in the final code presented in this report but instances of false VCs did arise when mistakes were made whilst working on the examples. In SPARK 2005 the Simplifier will sometimes detect false VCs and report them as such, although it is more likely that they will be reported as unproven. The Riposte counter-example finding tool can be used to check unproven SPARK 2005 VCs and, if it determines that they are false, generate counter-examples to help the user understand why they are false. Our understanding is that, due to the technology used, GNATprove will always report false SPARK 2014 VCs as not proven rather than false. The ability to detect false VCs and generate counter-examples would be a useful extension to the capabilities of the SPARK 2014 tools and it is recommended that this be included in future. It is understood that such a feature is already planned.

Recommendation 08: The SPARK 2014 tools currently lack automated support for providing an overall proof status summary. This is important when analysing larger projects and it is recommended that such a feature be developed.

Recommendation 09: There are many options configuring the behaviour of the tools. Whilst these provide flexibility to the user they can also be confusing. When a proof attempt fails there are various options that can be tried such as increasing the timeout, changing the proof mode to –proof=path_wp, or changing the expression semantics via -gnato13. Then these changes are remembered by GPS and are applied to the command line the next time the 'Prove ...' command is invoked. Some thought needs to be given to what the most sensible default options are and how best to deal with the user interaction for managing these options.

3.10.2 Observations

Observation 01: The restriction that Post aspects are only allowed on subprogram specifications will be removed, so GNATprove will permit postconditions to appear on bodies for which there is no separate specification [M227-046].

Observation 02: From a usability perspective it would be preferable if flow analysis and proof were not separate modes of operation. This is a known issue [M327-024] and will be addressed in a future version of GNATprove.

Observation 03: The GNATprove/GPS integration provides the facility to display the path to an unproven check. To enable this feature the option –proof=then_split or –proof=path_wp must be used, and a small icon appears next to the line number where the unproved check occurs. Clicking this icon causes the path to the unproved check to be highlighted. This feature can be useful for debugging failed proofs when there are multiple paths leading to a check.

Observation 04: At present, attempting to analyse QueueOperations with –mode=flow results in the error "raised WHY.NOT_IMPLEMENTED : flow-control_flow_graph.adb:513". This will be addressed in a future version of GNATprove.

Observation 05: Using stubs for which no completion is provided is not currently possible with GNATprove as it results in an internal error. This issue [M320-027] will be addressed in a future version of the tool. Another option would be to use the facilities of SPARK 2014 to mark code in or out of SPARK. The rules for this language feature are currently under development.

Observation 06: Note that modelling SPARK 2005 proof functions as executable functions in SPARK 2014 is not ideal as there is nothing to prevent them from being called in general code. The solution to this is to label them as ghost functions via "convention => ghost" which means they may only be called from within proof expressions or from other ghost functions. This approach would have been taken for this example but it was not yet implemented when the example was originally developed.

Observation 07: Flow analysis of the Central Heating Controller example with GNATprove is not currently possible due to use of features for which flow analysis has not yet been implemented.

Observation 08: The partial correctness proof of the Central Heating Controller demonstrates that GNATprove is able to discharge quite large postconditions in a reasonable time. (The default timeout of 1s was sufficient. Full statistics are in the appendix.)

Observation 09: It is interesting to note that GNATprove generates and proves 37 VCs, compared to 97 VCs for the SPARK 2005 tools. This highlights the different VC generation schemes used. GNATprove generates one VC for each check, whilst SPARK 2014 generates one VC for each path to each check. This results in SPARK 2005 having a larger number of VCs compared to GNATprove, but they tend to be smaller.

Observation 10: The SPARK 2005 tools use a configuration file to specify properties of the target such as size of integer. By default GNATprove assumes that the configuration of the target is the same as the host, but this can be overridden and a specific target configuration can be provided with the switch -gnateT.

Observation 11: The Raise_To_Power example illustrates that VCs involving non-linear arithmetic are typically hard to prove, both for the SPARK 2005 and the SPARK 2014 toolsets. The Riposte counter-example finding tool was applied to the SPARK 2005 VCs in an attempt to clarify whether they are actually non-provable or just hard, but it reported COMPLEXITY_EXPLOSION and could not reach a verdict.

Observation 12: GNATprove is unable to prove the loop invariant because The_Max is not an expression function and so its postcondition is not propagated to VCs for expressions where it is used. This will be addressed in a future version of GNATprove [M322-027].

Observation 13: With the default timeout of 1s GNATprove is unable to prove the two loop invariants. It also leaves the range check undischarged on line 14 of the package specification (in the expression function Sum_Between). Increasing the timeout from 1s to 15s enables GNATprove to successfully discharge the VCs for the loop invariants - the total time for this is 37s. The range check is actually unprovable as it stands. To discharge it requires strengthening the postcondition on Sum_Between, but that would prevent it from being used in the proof of the loop invariants so this will be deferred until completion of L525-024.

Observation 14: It is interesting to note the differences in the function Summed_Between in the SPARK 2005 and SPARK 2014 versions of this example. In SPARK 2005 it appears as a proof function and in SPARK 2014 it is an expression function. Initially the expression function was given without a separate declaration but GNATprove was unable to prove the range check for the recursive call on the second line of the expression. This was addressed by adding a separate declaration with the necessary postcondition aspect, allowing all checks to be proved.

Observation 15: In order to complete the SPARK 2005 version of this proof it was necessary to supply the Simplifier with the following user-defined proof rule.

Observation 16: In order to get GNATprove to prove all the VCs for T3Q5.Sort the option -proof=path_wp was specified. This causes GNATprove to generate a VC for each path to each check. It can be useful to try this option when GNATprove is unable to prove all checks, although it generally increases the time taken so it is not recommended as the default.

Observation 17: For some subprograms there is a pragma providing a precondition or postcondition on the subprogram specification, and a second pragma giving a refined version of the precondition or postcondition on the subprogram body. These refined preconditions and postconditions are currently ignored by GNATprove but will be analysed in future. For an example see the procedure AuditLog.ClearLogEntries.

Observation 18: We were concerned that the presence of a non-SPARK construct (access type) in file.ads might prevent the analysis of other units that depended on this unit or this type. This concern arose because in SPARK 2005 the access type would have represented a syntax error which, if not hidden, would have prevented the

analysis of any other unit which depended on file.ads. Some experimentation with GNATprove determined that the presence of the offending construct does not prevent the analysis of any dependent units.

Observation 19: When GNATprove encounters a package body containing compilation errors it prevents the analysis of other units that depend on the corresponding package specification. On the other hand, if the package body is missing altogether then dependent units can be analysed.

Observation 20: The failure to analyse a package body (which does exist) appears to render the corresponding package specification unavailable for analysis. But if the package body is removed altogether then the corresponding specification can be used in the analysis of units which depend on it. This is because GNATprove will not analyse a unit unless each unit on which it depends has a specification and either a body which can be successfully compiled or else the body is absent. This is a significant difference from SPARK 2005, in which analysis of package bodies is totally independent from analysis of units that depend on their specifications. ([M422-020] covers the internal discussion of this issue.)

Observation 21: When analysing AuditLog, 38 additional VCs were generated when -gnato11 was specified, compared to when -gnato13 was specified. These represent 38 additional checks for overflow in proof expressions. However GNATprove was able to discharge all of the additional VCs, resulting in no overall difference to the number of undischarged VCs. This was somewhat unexpected and may be an indication of the high quality of the original proof annotations.

Observation 22: As a piece of further work it would be interesting to investigate the remaining undischarged VCs for AuditLog and determine how best to tackle them.

3.11 Discussion and Conclusions

In this section we discuss the main findings during this study with regards to the goals of the Hi-Lite project, but also with regards to general usability. Where relevant we make comparisons to SPARK 2005. The conclusions stated here are drawn mainly from the examples presented in this report but also draw on our experiences working on the VerifyThis [HMW13] problems.

3.11.1 Goals of Hi-Lite

For convenience, we here list the goals extracted from the text of the technical Hi-Lite project proposal, that are relevant to our evaluation:

- The objective of project Hi-Lite is to give developers of high integrity applications the means to verify completely a safety property or a logic property of an implementation.
- Our goal is to provide ‘’light” tools and methods for the development of high integrity applications, that a developer may apply to an ongoing implementation on his setup box to verify some safety properties and logic properties.
- The first objective of project Hi-Lite is to define a language of logic annotations that are also executable and that allow the expression of unit tests. By defining a common specification language, we will facilitate the transition from testing techniques to static analysis techniques on the same project.
- The second objective of project Hi-Lite is to combine verification tools that rank among the best available, by using the specification language as a common basis. It is a matter of both adding the verification capabilities of each tool, and combining tools that produce annotations with tools that consume them to get better results together than with each tool separately.
- We take as our main objectives for this specification language the possibility to express unit tests as partial specifications, and the ability to run specifications in tests.
- A key objective of the specification language is to facilitate the automatic verification and automatic generation of annotations in a modular framework.
- The integration of discrete tools with the IDE developed by AdaCore, GPS and GNATbench, has as objectives to facilitate the selective launch of analyses, the examination of intermediate and final results, and the validation of results by the user.

3.11.2 Our Users

Our usability point of view: On the one hand, we would like to attract new users, to encourage writing contracts for the reduction of testing work as well as for the benefits of formal verification. On the other hand, we want to keep rewarding the (arguably rare breed) of existing SPARK 2005 users who already write contracts and perform proof for industrial applications. The experience of these users is very valuable. Naturally, we would like their sometimes challenging practical formal verification work to be made easier in SPARK 2014 than in SPARK 2005.

3.11.3 Benefits of Executable Contracts

We are impressed with the benefits of executable contracts that SPARK 2014 and GNATprove offer. We can classify these benefits in at least three categories:

1. Specification validation.
2. Faster debugging of failed proof attempts.
3. Modular verification with mixed test and proof.

Specification validation

Traditionally, contracts have been interpreted quite differently depending on whether used for formal program verification or for run-time assertion checking. For formal program verification, assertions have typically been interpreted as formulae in classical first-order logic, like in SPARK 2005. This is convenient for users who interact mathematically with a prover. At the same time this approach has the drawback that it is easy to produce invalid specifications, typically specifications that involve partial functions, for example division by zero, or accessing out of array bounds. An effect of having the same semantics in assertions as in the program code - as in SPARK 2014 - is that run-time exceptions must be considered, and avoided, in the assertions as well as in the program. This introduces more proof obligations and there is a risk of producing more false alarms. On the other hand this approach can support the user in writing correct specifications. In the design of SPARK 2014, the risk of introducing false alarms has been minimised by providing an overflow checking mode where proof obligations for intermediate overflow in annotations are eliminated (by using a bignum library), as these would lead to false alarms (unless run-time assertion execution is on, in this case you would use a strict overflow checking mode).

In this study, we did indeed get extra verification conditions (VCs) for run-time exception freedom in annotations (18 out of 65 VCs for a limited part of Tokeneer). 9 out of these VCs were unproved. They were all related to a system invariant (passed around as pre and post conditions) which was improved by adding the property lacking for RTEF (runtime exception freedom) proof. The RTE in question was an underflow range check which in specification terms corresponded to unintentionally including a negative count of file entries. The modifications to the invariant improves the quality of the contracts so we do not count these additional VCs as false alarms.

Furthermore, for the same limited part of Tokeneer, when comparing the number of VCs for the *eliminate* versus *strict* overflow checking modes, 38 additional VCs (in addition to the original 65 VCs, 103 VCs in total) were generated in strict mode. However GNATprove was able to discharge all of the additional VCs, resulting in no false alarms. This can be compared with results from the VerifyThis competition [HMW13], where 10 additional VCs are generated and automatically discharged for the simpler challenge 1, whereas 60 additional VCs are generated for the more complex challenge 2, most of them requiring modifications to annotations in order to be proved. It can be expected that the more complex arithmetic computations in annotations, the more work it will be for the user with strict overflow checking mode.

Debugging of failed proof attempts

In the proving process for non-trivial functional properties the user is required to write loop invariants, often involving universal quantifiers and sometimes non-linear arithmetic. It is often tricky to get these loop invariants right. Additionally the incompleteness of provers for the formalisms we work in typically lead to false alarms. During the work with the VerifyThis problems, significant time was saved by resorting to execution of loop invariants, in debugging such failed proof attempts, postponing writing additional lemmas until absolutely necessary.

Modular verification with mixed test and proof

Our thesis is this will offer benefits, but this activity was not completed as part of the case study. The Tokeneer project includes a test suite, which might be used to drive the execution of the annotations for this activity.

3.11.4 Abstraction

Abstraction is critical for formal verification to scale to industrial size projects. There is currently very little in the way of support for abstraction in SPARK 2014 proof expressions. The good news is that there are plans to implement abstract state and refined user-defined pre- and post-contracts. The Tokeneer case study would provide a useful example on which to evaluate these new language features once they are implemented.

Recommendations 01 and 02 relate to issues with abstraction.

3.11.5 User Interaction in GPS and GNATprove

During the course of producing this case study the GNATprove tool was used extensively within the GPS environment. This section summarises the high-level findings relating to this method of user interaction.

1. A significant benefit of this way of working is the direct feedback given to the user by highlighting the relevant source code line when a check passes or fails. This makes it very obvious to the user where the check is in the source code without any need for inspecting Verification Conditions or other output files. The ability to display the path leading to a failed check can also be very useful.
2. Following on from the previous point, GPS highlights the source code lines in red or green as the analysis progresses. This gives useful feedback on the progress of the proof.
3. The ability to invoke 'Prove Line', 'Prove Subprogram', 'Prove File' or 'Prove All' speeds up the user interaction, avoiding the repetition of potentially time-consuming proofs of subprograms other than the one the user is currently interested in. It also allows experimentation with longer timeouts on proofs of particular checks without having to apply those timeouts to the rest of the proofs.
4. In relation to the previous two points, in the future it might be worth considering a mode where 'Prove Line' was automatically invoked on the line being edited (or 'Prove Subprogram' on the subprogram being edited) so that lines could be highlighted red or green to show their proof status as the user edited them.
5. The tools currently lack facilities for providing feedback to the user on the proof status of larger projects (such as the Tokeneer example), hence we used command-line tools to extract proof statistics (e.g. number of checks proved and not proved) from the textual output. A proof status summary, something like that given by the POGS tool for SPARK 2005, would be very useful. (See Recommendation 08.)
6. The first point in this list highlights the benefits of GPS and GNATprove in terms of user-friendliness, displaying lines in green and red to indicate whether checks have been proved or not. When a check cannot be proved it may be obvious to the user what needs to be done to correct the problem, or it may be possible to provide more information, for example by displaying the path to the check. However, there will remain cases when the user does not understand why a proof attempt is failing and they would like more information from the VC to help explain precisely what cannot be proved. A means of presenting the information from the VCs in a more user-friendly form would be useful. (See Recommendation 07.)
7. There are many options configuring the behaviour of the tools. Whilst these provide flexibility to the user they can also be confusing. When a proof attempt fails there are various options that can be tried such as increasing the timeout, changing the proof mode to –proof=path_wp, or changing the expression semantics via -gnato13. Then these changes are remembered by GPS and are applied to the command line the next time the 'Prove ...' command is invoked. Some thought needs to be given to what the most sensible default options are and how best to deal with the user interaction for managing these options. (This is Recommendation 09.)

3.11.6 Comparison to SPARK 2005

All the examples in this case study were analysed both with the SPARK 2005 tools and the SPARK 2014 tools, enabling comparisons to be made. This section gives a high-level summary of the main points that emerged from this comparison.

1. For small examples, the conversion of existing SPARK 2005 code to SPARK 2014 was relatively straightforward. However, SPARK 2014 does not yet support modelling of abstract and refined state and refining contracts, which is crucial for larger applications. Analysis of the whole of Tokeneer would therefore have

required a significant rewrite, departing from the intended design (which was largely formalised in Z). Consequently, evaluation of a complete larger industrial case study has been assigned to future work and this study has focused on studying particular aspects of SPARK2014 and GNATprove on a smaller part of Tokeneer. The Ada language subset is larger in SPARK 2014 than in SPARK 2005 so most executable code statements and declarations did not need to be changed. (Current exceptions to this are tagged types and tasking, which are supported by SPARK 2005 but not SPARK 2014. These features will be incorporated into the SPARK 2014 language in future.) The SPARK 2005 annotations specifying constraints on subprogram contracts, loop invariants and other assertions were, in general, fundamentally similar to their SPARK 2014 equivalents and much of the translation could be automated. The main differences related to proof functions which had to be modelled as executable functions in SPARK 2014. (Note that global and derives aspects were not considered as part of this study.)

2. The SPARK 2014 proof tools remain significantly slower than their SPARK 2005 equivalents. However they are still under very active development and are continuously improving both in terms of speed and in terms of completeness (and the latter has a positive impact on the former). The fact that GNATprove successfully proves all checks and postconditions for the SPARK 2014 version of the Central Heating Controller example is very encouraging.

3. One of the expected benefits of GNATprove was that it would be easier to take an existing Ada code base, analyse it to see how much was in SPARK 2014 and how much of that could be proved, then work on the code to increase the percentage that could be analysed and proved. This was partially realised with the Tokeneer analysis. In our initial attempts to analyse all the core code we were surprised by how much code was excluded from analysis, even though we believed it to be 'in SPARK'. It transpired that this was because non-SPARK package bodies were being analysed and these were 'with'ing packages that were not available for analysis. That caused analysis of the bodies to fail which in turn caused the analysis of the corresponding package specifications to fail, and so on. This was contrary to our expectation that errors in a package body should not affect the analysis of other units which depended on its specification, and reflects the fact that GNATprove is based on compiler technology. (See Observation 19.) The workaround is simply not to present the offending bodies for analysis, by removing them from the project directory. Another option would be for GNATprove to have a setting where errors in bodies did not affect the analysis of specifications. In other ways GNATprove does provide more flexibility than SPARK 2005 in terms of its ability to analyse code which is a mixture of SPARK and non-SPARK.

3.11.7 Summary of Main Overall Benefits

At a very high-level, the main benefits of SPARK 2014 can be summarised as follows.

- The ability to validate specifications through executable assertions.
- The ability to verify correctness in a test context through executable assertions.
- Usability improvements to the proof process brought about by GPS and GNATprove integration.
- The flexibility and potential cost savings offered by hybrid verification.

(It should be noted that the use of executable assertions in a test context was not investigated as part of this study. It is identified as an area for further work.)

3.11.8 Issues

Most of the issues listed below are related to language features or other functionality that has not yet been implemented in the SPARK 2014 tools. In general there are either concrete plans to address them or ongoing discussions to determine appropriate solutions.

Major issues:

- Abstraction and refinement of contracts necessary for industrial applications, and the design of these features in SPARK 2014 have to be evaluated once they have been implemented. (See Recommendations 01 and 02.)

- The time taken to discharge proofs needs to be improved, either by improving the VC generation, the prover itself, or both. (Performance statistics are provided in the appendix of this report.)

Medium Issues:

- The attributes `Old` and `Loop_Entry` may only be referenced in a very limited set of locations. There are cases where it is desirable to reference them in other locations in order to express properties required for a proof but the language rules do not permit this. One option would be to relax the language rules. If that is not possible then ghost variables may offer an acceptable solution. (See Recommendation 04.)
- When attempting to debug failed proof attempts it can sometimes be very useful to inspect the detail of the VCs in question, either to determine what needs to be done to complete the proof automatically or to construct a manual argument. This is relatively straightforward in SPARK 2005, but SPARK 2014 currently lacks facilities for this. (See Recommendation 05.)

Minor Issues are covered by the remaining Recommendations and Observations summarised earlier and are not repeated again here.

3.11.9 Further work

This section lists a number of suggestions for possible future work arising from this case study.

- Testing and executing assertions: evaluation of the verification benefit of executing assertions, by using the existing Tokeneer test suite.
- Application of SPARK 2014 and GNATprove to *hidden* parts of the SPARK 2005 code, to evaluate the practical significance for an industrial application (Tokeneer) of SPARK 2014 being a larger subset of Ada than SPARK 2005, and again to analyse the verification benefits from being able to test contracts that are not amenable to formal verification.
- Evaluation of generative mode (where the annotations have not been specified by the programmer and are determined automatically by the SPARK 2014 tools based on the code). Which annotations could have been generated on Tokeneer?
- Evaluation of new SPARK 2014 language features for abstraction in proof expressions, once these features have been implemented, using Tokeneer as an example.
- Completing the conversion of Tokeneer to SPARK 2014. One of the main tasks would be to convert all uses of SPARK_IO to Ada.Text_IO.

3.12 Tools and Performance

3.12.1 Tools used

The following tools were used:

- GNATprove 0.2w (20130414)
- GPS 5.3.0w (20130415)
- GNAT Pro 7.2.0w (20130414-47)
- SPARK Pro toolset 11.0.0

3.12.2 Performance

The results reported in this section are from analysis on a desktop PC (i7 860, 2.8GHz, 4 CPUs, 6Gb RAM) running 64-bit Linux (Debian).

The timings for GNATprove were generated by running “Clean Proofs” followed by “Prove Subprogram” from within GPS and noting the elapsed time reported in the Messages tab. It is acknowledged that these times include

the time spent by GNATprove analysing and generating VCs as well as the time spent by Alt-Ergo attempting to discharge the VCs, plus any additional overhead from GPS.

The SPARK 2005 timings were generated by running the command "time spadesimp subprog.vcg" and taking the reported 'real' time on the assumption that this is closest to the measure reported by GPS. (The Simplifier executable is called spadesimp.) It is acknowledged that this does not include the time spent by the Examiner analysing and generating VCs, although this has been measured separately and is only 00.15s in total for all seven subprograms in the fisr example (Exchange).

For GNATprove (SPARK 2014) the option -j4 was used to specify 4 parallel processes to make use of the available processors. For SPARKsimp (SPARK 2005) the equivalent option -p=4 was used.

3.12.3 Exchange procedure

Subprogram	GNATprove	SPARK 2005	Comments
Exchange	Proved 00.56s	Proved 00.08s	
Exchange_No_Post	Proved 00.30s	Proved 00.08s	
Exchange_No_Post_Unused	Proved 00.35s	Proved 00.08s	
Exchange_No_Post_Uninitialized	Proved 00.35s	Not proved 00.08s	1
Exchange_With_Post_Unused	Not proved 01.55s	Not proved 00.09s	2
Exchange_With_Post_Uninitialized	Not proved 01.47s	Not proved 00.10s	2
Exchange_RTE	Not proved 01.77s	Not proved 00.11s	2

Comments:

1. Uninitialized variable results in unprovable (false) VCs in SPARK 2005. This is detected by GNATprove when run in flow analysis mode.
2. VCs not provable due to deliberate error. GNATprove timings reflect the 1s timeout used.

3.12.4 Stacks, Queues and QueueOperations

Subprogram	GNATprove	SPARK 2005	Comments
ReverseQueue	Proved 00.40s	Proved 00.08s	

3.12.5 Stacks, Queues and QueueOperations with Proof

Subprogram	GNATprove	SPARK 2005	Comments
ReverseQueue	Proved 06.06s	Proved 00.59s	1

Comments:

1. The timeout for GNATprove was increased to 2s for this example. The SPARK 2005 time consists of 0.516s for the Simplifier to prove 25 out of 27 VCs and 0.069s for Victor + Alt-Ergo to prove the remaining 2 VCs.

3.12.6 Central Heating Controller

RTE Proof

Subprogram	GNATprove	SPARK 2005	Comments
HeatingSystem_DFA	Proved 02.79s	Proved 01.71s	

Correctness Proof

Subprogram	GNATprove	SPARK 2005	Comments
HeatingSystem_Proof	Proved 08.41s	Proved 02.64s	1

Comments:

1. There were 97 SPARK 2005 VCs of which 96 were proved by the Simplifier and the remaining one was proved by Victor + Alt-Ergo. GNATprove generated and proved 37 VCs.

3.12.7 Advanced SPARK 2005 Training Course

Subprogram	GNATprove	SPARK 2005	Comments
Increment	Proved 00.61s	Proved 00.14s	
Increment2	Proved 00.88s	Proved 00.17s	
Swap, NandGate, NextDay_* (t1q3)	Proved 01.32s	Proved 00.55s	1
Swap, NandGate, NextDay_* (alt)	Proved 01.32s	Proved 00.46s	
ISQRT	Proved 02.18s	Proved 00.29s	
Bounded_Add	Proved 01.12s	Proved 00.34s	2
Raise_To_Power	Not proved	Not proved	3

Comments:

1. These are the combined times for proving all the subprograms named as they are all located in the same package. (Timings for individual subprograms could be obtained by using 'Prove Subprogram' with GNATprove and by proving the individual VC files with the SPARK 2005 tools if this information was required.) GNATprove generates and discharges checks for four postconditions and a range check. The SPARK 2005 tools generate and discharge 24 VCs of which 23 are discharged by the Simplifier and the remaining one requires Victor + Alt-Ergo.
2. Of the 15 SPARK 2005 VCs, 14 were discharged by the Simplifier and 1 by Victor + Alt-Ergo.
3. Come back to this - open todo item in main body of report.

3.12.8 Array Examples

Subprogram	GNATprove	SPARK 2005	Comments
T2Q1.Swap	Proved 00.56s	Proved 00.20s	
T2Q2.Clear	Proved 00.51s	Proved 00.16s	
T2Q3.Find	Proved 00.51s	Proved 00.17s	1
T2Q4.Clear	Proved 02.63s	Proved 00.42s	2
T2Q5.MaxElement_P1B1	Proved 01.52s	Proved 00.27s	
T2Q5.MaxElement_P2B1	Not proved 03.09s	Proved 00.53s	
T2Q5.MaxElement_P3B1	Proved 00.86s	Proved 00.10s	4
T2Q5.MaxElement_P1B2	Proved 01.62s	Proved 00.38s	
T2Q5.MaxElement_P2B2	Not proved 03.29s	Proved 00.83s	3
T2Q5.MaxElement_P3B2	Proved 01.11s	Proved 00.12s	
T2Q5.MaxElement_P1B3	Proved 01.72s	Proved 00.31s	
T2Q6.SumArray	Proved 11.08s	Proved 00.40s	5
T2Q6.SumArray_Shift	Proved 09.37s	Proved 00.80s	5
T2Q7.Find	Proved 01.08s	Proved 00.23s	
T2Q8.CreateFibArray	Not proved 13.42s	Not proved 16.62s	
T2Q8.CreateFibArray_RTCOnly	Not proved 13.20s	Not proved 16.40s	

1. T2Q3 is not mentioned in the body of the report as it is identical to T2Q7 but without a postcondition.
2. Of the 9 SPARK 2005 VCs, 7 were discharged by the Simplifier and 2 by Victor + Alt-Ergo.
3. See observation and explanation in main body of report.
4. In fact GNATprove does not generate or prove any VCs as this example is so trivial.
5. The GNATprove timeout was increased to 12s.

3.12.9 Further Advanced SPARK course examples

Subprogram	GNATprove	SPARK 2005	Comments
T3Q1.Swap and Rotate3 (combined)	Proved 01.32s	Proved 00.39s	1
T3Q2.Double and Quadruple	Proved 01.27s	Proved 00.29s	
T3Q3.DoNothing	Not proved 03.70s	Not proved 1m49s	
T3Q4.SumArray	Proved 04.60s	Proved 00.45s	2
T3Q5.* (Sorting algorithm)	Not proved 10.92s	Proved 20.22s	3

1. Of the 8 SPARK 2005 VCs 7 were proved by the Simplifier and 1 by Victor + Alt-Ergo.
2. Of the 6 SPARK 2005 VCs 4 were proved by the Simplifier and 2 by Victor + Alt-Ergo.
3. There were 32 SPARK 2005 VCs. 24 were proved by the Simplifier, 1 requiring a user-defined rule. The remainder were discharged by Victor + Alt-Ergo.

3.12.10 Tokeneer

Analysis	GNATprove	SPARK 2005	Comments
All core (SPARK) files	62% proved 06m49s	100% proved 06m07s	1
AuditLog	88% proved 00m38s	100% proved 01m46s	2

1. GNATprove was run with a timeout of 5s. The SPARK 2005 proof tools were invoked via the command "sparksimp" with the default settings. As discussed in the body of the report, the full conversion to SPARK 2014 is not complete so it is not possible to make a meaningful comparison between these results. It should also be noted that of the 2433 SPARK 2005 VCs, 100 (4%) were discharged with the aid of user-defined proof rules and 23 (1%) were discharged by manual review. One VC is discharged by Victor + Alt-Ergo, although this was discharged by manual review in the original Tokeneer development (as SPARKbridge was not available at the time). If the original manual review file is reinstated and Victor is not used then the total proof time drops from 6m07s to 1m39s.
2. For the SPARK 2005 version of the code, the AuditLog package produces the 1 VC in Tokeneer that is proved by Victor + Alt-Ergo. If Victor is not invoked and the original manual review file is reinstated then the proof completes in 8 seconds. The GNATprove results were generated with a timeout of 5s. Reducing this timeout to 1s halves the time taken for the proof but with one fewer VC proved. It is possible that increasing the timeout further would result in more VCs being proved at the expense of a longer analysis time, although increasing the timeout to 300s does not prove any additional VCs (but increases the total time to 15m47s).

CHAPTER

FOUR

INTEGRATION IN MYCCM

The activity of Thales in Hi-Lite consists of the adaptation of MyCCM to make it generate code that is compliant the Hi-Lite formal specifications.

4.1 Presentation of MyCCM

MyCCM is a component framework developed by Thales. It allows the design of application architectures with graphical tools, the generation of the corresponding infrastructure code, and then the compilation and execution on top of a runtime.

MyCCM is not a single product; it is a family of generators and runtimes that fit specific domain constraints (e.g. space satellite, image processing, etc.) that range from critical systems to flexible near real-time applications. All the MyCCM frameworks share the same way of designing architectures.

MyCCM models involve the notion of functional components that encapsulate algorithms (i.e. behaviours). The behaviour of a component interacts is held in the component implementation, and interacts with the outside of the component through communication ports. We thus have a complete isolation between the component function and its interactions within the whole architecture.

Communication ports are functional contracts that are provided or required by the components.

The user of a MyCCM tool chain does not immediately focus on the source code. He or she focuses on the design with components that interact one with another. The source code of the interactions is automatically generated. The functional behaviour of the component is then written, so that it can be compiled against the component definition and the component connection code. It thus helps users focus on the "smart" part of the applications, that is the algorithms; the infrastructure part (communication and execution resources) is generated from the architecture models.

In the scope of Hi-Lite, Thales works with a MyCCM framework that generates Ada source code for space satellite on-board software.

4.2 Work in the scope of Hi-Lite

Hi-Lite deals with proof of functional code. Therefore, the study of Thales focuses on the behaviour code of the components. The idea is to have formal specification integrated in the component models, and then translated into the generated code that will be connected with the component behaviour code.

The work done by Thales in the scope of Hi-Lite focuses on two things. First, the exploitation of Hi-Lite formal notations at the level of the architecture models. Second, the generation of source code that is compliant with the Hi-Lite specifications and integrates the formal specifications set in the architecture models.

4.2.1 Specification and transcription of formal specifications

MyCCM manages two kinds of interaction ports:

- interface-based ports, that correspond to operation calls;
- message passing ports.

Formal specifications only apply to interface-based ports, that manage functional contracts (interfaces that define operations). Message passing ports are not altered by Hi-Lite, since they implement a unique functional contract, which is the transmission of data.

A first prototype of generator has been developed to test the integration of MyCCM generated code into the Hi-Lite tool suite. Formal contracts are associated with operations of interfaces in MyCCM models, and then translated into the generated source code the following way.

For ports that provide the interface, the generator produces empty operation skeletons that includes the formal specifications. The user will have then to fill in the empty operation definition with the behaviour code. The generator supports round-trip generation, so that it is possible de regenerate the code without erasing the user code.

For ports that require the interface, the generator simply produces operation declarations (i.e. stub signatures) with the functional specifications. Thus, operation stubs carry the same formal specification as their implementation in the providing component.

The intent for the exploitation in Hi-Lite is the following: For provided operations, it is possible to prove the operation implementations against their formal specifications. For required operations, the formal specification can be used to prove the code that uses these operations.

4.2.2 Adaptation of the code generated by MyCCM

The existing generator produces code that is almost compliant with Hi-Lite, for the part that is studied in the scope of the project (that is, proof of component behaviours).

In order to make components communicate one with another, MyCCM generates data structures that contain references to component ports. These structures are read by the behaviour code to send data or commands through the communication ports.

The structures are set by the generated code, and actually contain pointers. These pointers are simply read by the user code. The pointers are neither modified nor even manipulated by the behaviour code; the whole structure is simply passed as an argument of the component operations.

As Hi-Lite does not manage pointers, they must be completely concealed and in the private parts of the structures, to ensure they are not seen by the behaviour code that will be proven.

The current version of the MyCCM framework does work like this. It has been modified accordingly.

4.3 Summary of the work done

MyCCM was modified to make it compliant with the limitations defined in SPARK 2014. The modifications consisted in:

- removal of discriminants in the MyCCM structure records;
- isolation of the pointer-related calls in separate packages;
- generation of Ada 2012 specifications.

The targeted code for formal proof is the business code encapsulated in the MyCCM components. The MyCCM technical code is out of the scope of the study, as it consists of mainly non-SPARK 2014 compliant code with very limited functional aspect. As a consequence, most of the work focused on an improvement of the MyCCM generator that deals with the implementation of the MyCCM components.

4.3.1 Removal of discriminants in the MyCCM structure records

MyCCM relies on Ada records that contain references to component ports. Each component has a record containing references to the other components it is supposed to communicate with. In the initial version of MyCCM, these records are defines with a discriminant for each reference to set. The rationale for this was to ensure that records are correctly initialised and then not modified during the execution.

The initial declaration of a component structure record was thus the following:

```
type Object_T
  (Activation_Itf_Ptr_P : MCCM.Activable.Object_Ptr_T;
   Context_Ptr_P : Components.ccm_C1_T_Context.Object_Ptr_T)
 is tagged limited record
    Activation_Itf_Ptr : MCCM.Activable.Object_Ptr_T := Activation_Itf_Ptr_P;
    Context_Ptr : Components.ccm_C1_T_Context.Object_Ptr_T := Context_Ptr_P;
 end record;
```

This kind of definition is out of the scope of SPARK 2014. We thus changed the records to remove the discriminants. The new version of the generator now generates the following:

```
type Object_T
 is tagged limited record
   Activation_Itf_Ptr : MCCM.Activable.Object_Ptr_T := Activation_Itf_Ptr_P;
   Context_Ptr : Components.ccm_C1_T_Context.Object_Ptr_T := Context_Ptr_P;
end record;
```

And we generate an additional procedure to set the record fields:

```
procedure Set_Activation_Itf_Ptr (Self : in out Object_T;
                                  Val : in MCCM.Activable.Object_Ptr_T);
procedure Set_Context_Ptr (Self : in out Object_T;
                           Val : in Components.ccm_C1_T_Context.Object_Ptr_T);
```

As a consequence, the compiler cannot ensure that the fields of the records are correctly initialized and that the fields are not changed during the execution of the application.

The first point (initialization) is not a string issue, as the initialization is performed by generate code. Thus, we are sure the initialization is correct.

The second point is more problematic. Indeed, it is impossible to prevent the business code to access the technical code and do nasty things. After discussion, it has been decided to keep this point unresolved for the moment.

4.3.2 Isolation of the pointer-related calls in separate packages

The second point is also related to the definition of SPARK 2014. The business code encapsulated in a MyCCM component manipulates the component structure in order to send data to other components. So, typically, a component operation is like this:

```
procedure Activate (Servant : in out Servant_T) is
   Context_Ptr : Components.CCM_C1_T_Context.Object_Ptr_T;
   To_C2 : Interfaces.Intf1.Object_Ptr_T;
begin
   --  retrieve the reference of the port of the other component
   Context_Ptr := Components.CCM_C1_T.Get_Context_Ptr
      (Self => Components.CCM_C1_T.Object_T (Servant));
   To_C2 := Components.CCM_C1_T_Context.Get_Connection_TO_C2_Ptr
      (Self => Context_Ptr.all);
   --  call an operation of the other component
   Interfaces.Intf1.op1
      (Self => To_C2,
       a => 4,
       b => 2);
end Activate;
```

This kind of code is not compliant with SPARK 2014 because it manipulates the structure of the component, named Servant in the example, with its pointers.

This kind of code cannot be avoided in frameworks like MyCCM. It allows the isolation between the business code and the MyCCM infrastructure code. As a solution, we chose to put this code in separate procedures, defined in a separate package named Stub. Thus, the code becomes the following:

```
procedure Activate (Servant : in out Servant_T) is
begin
   --  retrieve the reference of the port of the other component
   Context_Ptr := Components.CCM_C1_T.Get_Context_Ptr
      (Self => Components.CCM_C1_T.Object_T (Servant));
   To_C2 := Components.CCM_C1_T_Context.Get_Connection_TO_C2_Ptr
      (Self => Context_Ptr.all);
   --  call an operation of the other component
   C1.Stub.To_C2.op1
      (Servant => Servant,
       a => 4,
       b => 2);
end Activate;
```

The operation code does not manipulate the structure of the Servant parameter any more.

For a single call, this has no impact on performance. However, for a sequence of calls, the performances are slightly worse, as the interface is reconnected each time. Therefore, this alternative solution is to be used together with the original one.

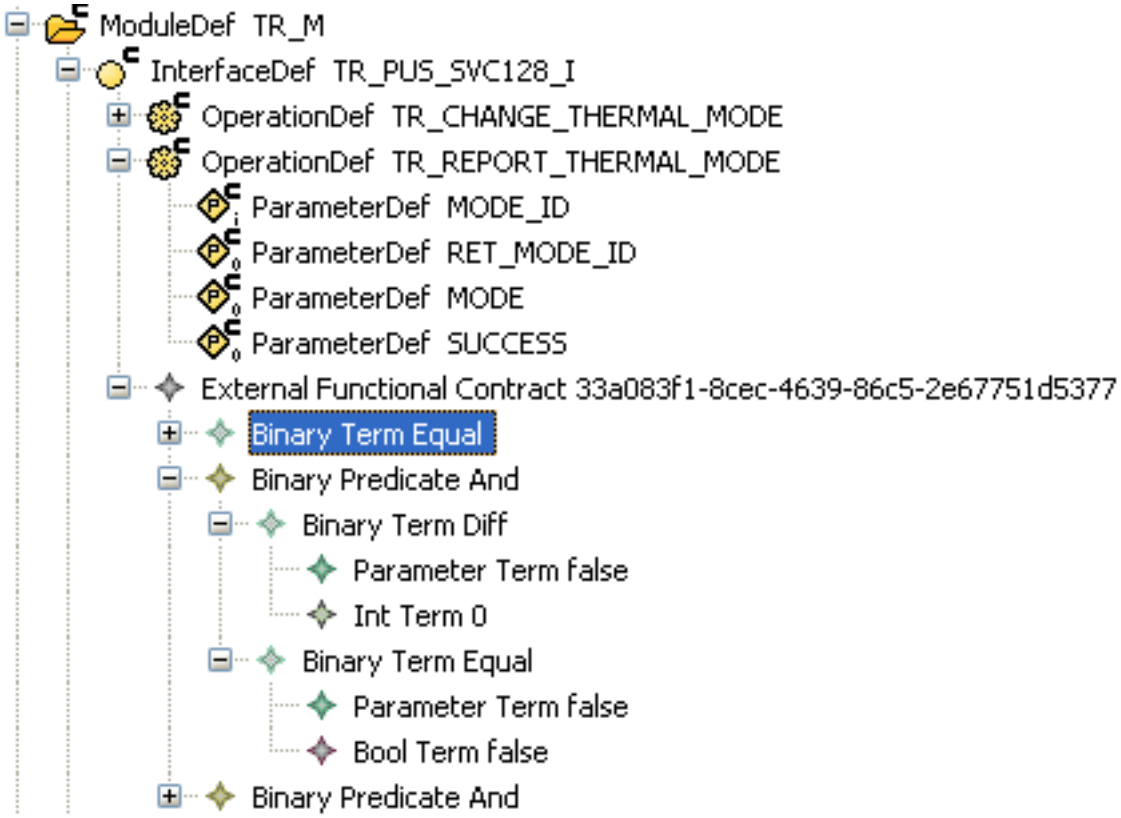
4.3.3 Generation of Ada 2012 specifications

The two previous sections described modifications in the code generated by MyCCM in order to make it compatible with SPARK 2014. This last section describes the introduction of formal specifications in the generated code.

In the scope of Hi-Lite, formal specifications only concern formal contracts, that is, definitions of operations. Therefore, formal specifications must be associated with the definitions of operations in interfaces. To do so, we defined a meta-model that is an extension of the core meta-model for the definition of interfaces. This meta-model extension defines the notion of predicate that can be associated with operations.

The structure of the meta-model is inspired by ACSL. We did not chose to use the structure defined by Ada 2012, as the MyCCM meta-model aim to be language-independent. ACSL seemed to be more general than the Ada 2012 concepts.

The meta-model is implemented using the Eclipse Modeling Framework (EMF), which automatically created a very basic editor that is merged with the MyCCM editor. We can thus create models (though not in a convenient way for the moment), as shown on this picture:



From this kind of model, the new version of the code generator can produce Ada 2012 specifications. As only the business code is concerned by Hi-Lite, the formal specifications are integrated in the code of the component implementation. For example:

```
procedure TR_REPORT_THERMAL_MODE
    (Servant : in out Servant_T;
     MODE_ID : in Basic_Types.Uint32_T;
     RET_MODE_ID : out Basic_Types.Uint32_T;
     MODE : out Basic_Types.Uint32_T;
     SUCCESS : out Basic_Types.Bool_T)with
     Post => ((MODE_ID)=(RET_MODE_ID))
         and (((MODE_ID)/=(0))and((SUCCESS)=(false)))
         and (((MODE_ID)=(0))and((SUCCESS)=(true)));
```

This way, the business code must be compliant with this specification. For operation that are called from the business code, we also generate the necessary formal specifications in the stub package.

4.3.4 Conclusion

The modifications made in the MyCCM generator and the meta-model extension allows the definition of formal contracts associated with operations of interfaces, and ensure the code generated from MyCCM to encapsulate the business code is compliant with the SPARK 2014 restrictions.

These modifications had a limited impact on the original framework. The two main drawbacks are:

- the communication infrastructure code is less hardened and can possibly be modified at runtime by some business code;
- there can be a slight performance loss.

4.4 Experiments

The experiments aimed at testing the validity of the new generation chain. For this, we chose a very limited part of a satellite architecture: we isolated a function in the MyCCM model of the architecture. This function is provided by a thermal regulation component. We removed some part of the business code that calls external libraries to avoid having an example too large. In the end, the function we want to test on is the following:

```
procedure TR_REPORT_THERMAL_MODE
    (Servant : in out Servant_T;
     MODE_ID : in Basic_Types.Uint32_T;
     RET_MODE_ID : out Basic_Types.Uint32_T;
     MODE : out Basic_Types.Uint32_T;
     SUCCESS : out Basic_Types.Bool_T)
 is
    --  Start of user code for private variables of TR_REPORT_THERMAL_MODE
    --  End of user code
 begin
    --  Start of user code for procedure implementation of TR_REPORT_THERMAL_MODE
    Ret_Mode_Id := Mode_Id;
    Mode := Mode_Id;
    Success := (Mode = 0);
    --  End of user code
 end TR_REPORT_THERMAL_MODE;
```

As we can see, besides its business behaviour that is hidden here, there is a relationship between the operation parameters. Our experiment focused on the specification of these relationship, to ensure the tools correctly check the validity of the code.

The specifications we made are the following (represented in Ada 2012 above):

- Success is true if and only if Mode is equal to 0;
- the output parameter Ret_Mode_Id is equal to the input parameter Mode_Id;
- the output parameter Mode is equal to the input parameter Mode_Id;

The example is very basic. It is not meant to check if the Hi-Lite tools can process such a simple proof. Its purpose is to check if a code generated from an architecture model extended with formal specifications can be automatically proved against the business code written by the user.

According to gnatprove, most of the files that are generated are not compliant with SPARK 2014. This situation was expected, as most of the MyCCM code contains pointers and other forbidden constructions. Our study is thus only focused on the files that contain the business code. For our experiment, the package is reported to be 100% compliant with SPARK 2014, thus indicating that the code can be processed by gnatprove.

MyCCM is therefore able to generate business code skeleton that can be processed for formal proof.

BIBLIOGRAPHY

[HMW13] “SPARK 2014 and GNATprove - A Competition Report from Builders of an Industrial-Strength Verifying Compiler”, by Duc Hoang, Yannick Moy and Angela Wallenburg, submitted for publication in the journal of Software Tools for Technology Transfer