CS-Note* 12

Bill J. Ellis
May 9, 2002

Assisting in the Proof of Spark Exception Freedom

Contents

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

Programs written in SPARK can be subjected to formal analysis. One such form of analysis is to prove that SPARK code is free from exceptions, in particular the exceptions used to support run time errors. This proof is achieved by discharging a number of EXAMINER generated verification conditions (VCs) using the SIMPLIFIER[1] and SPADE.

The process of proving exception freedom is very similar to the process for proving partial correctness. However, the kind of theorem proving problems that emerge, and hence the potential for automation, appear to be different.

This note tries to gain an appreciation of the issues associated with proving exception freedom. Particular emphasis is placed on exploring what the current practises are and what potential actions NuSPADE[2] can perform to help.

2 Runtime Exceptions

As default Ada compilers generate checks that are performed during the execution of a program. These runtime checks ensure that the integrity of the Ada program is not violated. For example, by checking that arrays are always accessed within their bounds.

Where these runtime checks successfully detect an error, the program is aborted and a suitable message is displayed[3]. In this manner Ada provides a safer runtime environment, refusing to perform any operation that would compromise the programming language. Such a safer environment is expected to be of real assistance in the development and maintenance of Ada systems.

2.1 Proving SPARK Exception Freedom

The SPARK Approach advocates a method for writing high integrity software systems. This method is supported by using the programming language SPARK (A specially annotated subset of Ada) and a number of tools designed to work with this language.

Proving exception freedom is often essential for programs developed in SPARK. Otherwise it is possible for an error to go undetected that has the potential to bring the entire system to a sudden halt. Or, if the runtime checks are switched off during compilation, for the program to become ambiguous[4] and continue execution. Relevant information about some of the larger systems developed in SPARK are shown below:

- **SHOLIS (Ship Helicopter Operational Limits Instrumentation System) [4, 1]**
  This system was broken down into extremely critical sections of code (SIL4) and less critical sections of code (non-SIL4). The separation of these categories of code where shown using information and data flow analysis. The SIL4 code was annotated with both Core Annotations and Proof Annotations[5] and subjected to specification proofs. The non-SIL4 code was only annotated with Core Annotations. All of the code (SIL4 and non-SIL4) was subjected to proof of exception freedom (See [4] section 4.2 and [1] section 3). The large majority of resulting VCs (84%) were proved automatically by the SIMPLIFIER (See [4] section 4.3). After proving exception freedom the Ada runtime checks could be safely removed during compilation. This was a key factor in meeting the response times for the system.

- **CA (Certification Authority) for Multos smart card scheme [3, 1]**
  Some proofs of exception freedom were conducted (See [3] section titled "The use of Spark in the CA"). Only the most "basic form of annotation and analysis" offered by the EXAMINER was used. Thus the code was only annotated with the Core Annotations.

- **LOCKHEED C130J [2, 1]**
  In building the system Proof Annotations were added, extracted from the formal requirements specification. However, proof of exception freedom was conducted at a later phase after testing (see [1] section 5).

Note that it is not uncommon to conduct proof of exception freedom in SPARK code that only has Core Annotations. Further note that such proof work is performed to some extent in all of the systems mentioned. The following is a brief summary of the key reasons that proof of exception freedom is often conducted for SPARK source code:

[1] Some problems were encountered using version 6 of the SIMPLIFIER. Thus all of the examples in this note were conducted using the SIMPLIFIER version 5 on windows NT.

[2] Or another software tool, possibly working in conjunction with NuSPADE.

[3] These exceptions could be caught inside Ada and dealt with accordingly. This note is more concerned with the SPARK Approach which advocates preventing such errors in the first place, rather than patching them up if they occur.

[4] Once ambiguous, a program can potentiality perform any sequence of instructions. It can shoot you in the foot in a myriad of exotic and novel ways.

[5] see section 3.

- No Proof Annotations required.
- Proof effort required is typically light-weight and has a good chance of being proved automatically by the SIMPLIFIER.
- Any runtime error will have an extreme effect, causing either the entire system to abort or to become ambiguous. Showing this can not happen provides a good layer of extra confidence.
- Where exception freedom is shown across the whole system the Ada runtime checks can be safely removed. This may be essential to meet the response times of real time systems.

2.2 Necessary Proof to Show SPARK Exception Freedom

The SPARK tools are capable of proving exception freedom of SPARK code. Much of this effort is eliminated, given the adoption of the SPARK language. The result is that the proof of exception freedom is achieved by proving a very focused collection of VCs. All of the potential exceptions in Ada are listed below, showing how it is proved that they do not occur in SPARK:

- **User Exception**
  The user can create and raise their own exceptions.
  - **Proof of absence in SPARK**
    The SPARK programming language forbids the user from creating or raising exceptions.
- **Storage_Error (Ada Runtime Exception)**
  This category of error occurs where the program is unable to find the memory to perform a particular operation.
  - **Proof of absence in SPARK**
    The SPARK programming language does not allow for any dynamic memory allocation[6]. It is thus possible to statically determine all of the memory that will be required and thus, by providing this memory, ensure that this error can not occur.
- **Program_Error(Ada Runtime Exception)**
  This category of error occurs where a one of a variety program errors are detected. These are mainly concerned with more advanced program behaviour available in Ada.
  - **Proof of absence in SPARK**
    The SPARK programming language excludes all of the program constructs that may lead to these errors.
- **Tasking_Error (Ada Runtime Exception)**
  This category of error occurs where tasks (Ada threads) are used inappropriately.
  - **Proof of absence in SPARK**
    The SPARK programming language excludes the use of tasks. Thus task errors can not occur in SPARK.
- **Constraint_Error (Ada Runtime Exception)**
  This category of error occurs where the program tries to perform operations that are outside the legal values. For example, accessing an array outside its bounds or dividing by zero. Note that in Ada 95 Constraint_Error subsumes the Ada 83 Numeric_Error.
  - **Proof of absence in SPARK**
    Unlike the exceptions listed above, some of the errors in this category can occur in SPARK. The possible errors are listed below:

Check	Can occur in spark code
Access_Check	No
Discriminant_Check	No
Tag_Check	No
Division_Check	Yes
Index_Check	Yes
Range_Check	Yes
Overflow_Check	Yes

[6]This is not guaranteed across all Ada compilers, but is generally the case (See [4] section 5.3).

In the cases where errors can potentially occur in SPARK code it must be proved that they do not. This is achieved by performing a formal proof.

3 SPARK Annotations

SPARK code is a subset of Ada with additional annotations. These annotations are used to provide additional information about the indented behaviour of the code.

3.1 Annotate for Confidence

Only for the most safety critical parts of SPARK code is a partial correctness proof performed. This has important implications on the amount of additional annotations that might be expected in SPARK code that is subjected to proof of exception freedom.

The following table lists the two main kinds of analysis and their corresponding categories of annotations, either Core Annotations or Proof Annotations. For each of these it shows which annotation statements are required and which extra annotations might be desired, to provide extra information that may help in later proof efforts. Note that analysis giving higher levels of confidence also require all of the required annotations giving lower levels of confidence. The only exception is information dependency (`derives`) annotation, which are only required for information flow analysis.

Kind of analysis	**Level of confidence (↑)**	**Additional Annotations**	
		Required	**Desired**
Dynamic analysis (Proof annotations)	Partial Correctness VCs	pre-condition (`pre`), post-condition (`post`), function post-condition (`return`) invariant (`assert`)	NONE
	Overflow runtime VCs	NONE	pre-condition (`pre`), post-condition (`post`), function post-condition (`return`) invariant (`assert`)
	Regular runtime VCs (Index, range and division checks)	NONE	pre-condition (`pre`), post-condition (`post`), function post-condition (`return`) invariant (`assert`)
Static analysis (Core annotations)	Information flow	information dependencies (`derives`)	NONE
	Data flow	global variables (`global`), inherit (`inherit`), own (`own`), initialises (`initializes`)	NONE

3.1.1 Add Import Information

Where variables are imported to a procedure or function their type is checked at runtime. If the type is found to be invalid a runtime error will be raised and the program aborted. This means that inside a procedure or function it must be the case that the import types were met.

Based on this reasoning the EXAMINER can safely insert additional information, as if as a pre-condition[7], stating that each imported variable is within its declared type.

[7]Exactly as a pre-condition in the normal logical sense. However, it should not be thought of as adding pre-condition annotation, otherwise the pre-condition in the next section would also refer to this added import information. To confuse matters further the next section does deal with import information, however, it does not decorate this with '~' as it would if the import information were inherited from the pre-condition.

3.1.2 Generate (or Modify) Invariant

To accommodate for the generation of runtime VCs where only Core Annotations are provided the EXAMINER must insert a default invariant. Instead of merely inserting the weakest invariant (`--#assert true;`) the EXAMINER inserts an invariant based on the problem. Note that where an invariant has been supplied by the user, the invariant that would have been generated by the EXAMINER is conjoined to the user invariant. The extra information made available by the EXAMINER is:

- **Repeat any pre-condition -** If a pre-condition has been specified it is repeated inside the invariant, suitably annotating variables that are both imported and exported with '~' to show that they are in terms of their initial values.
- **Add import type information -** Where it was possible to add import information they are repeated in the invariant.
- **Add for-loop type -** If the invariant being generated (or modified) is for a for-loop then the invariant states that the loop iterator is always within the range being iterated over.

3.1.3 Mixed VC Generation

In practise, doing overflow runtime VCs implicitly includes doing the regular runtime VCs. Further, doing the regular runtime VCs implicitly includes doing specification VCs. This is unfortunate as the runtime VCs become mixed with specification VCs in well annotated code. Further, the order of this mixing is not intuitive. Specification VCs are seen as offering the strongest confidence, and thus should not be generated at the weaker level of runtime VCs.

Although not confirmed[8] it is guessed that this behaviour stems from the insertion of check statements (`--#check <this>;`) directly into the code being analysed to generate the runtime VCs. Annotations present in this code are never removed, thus the result of generating runtime VCs is a more annotated code file to analyse.

[8]This would require much deeper knowledge of the operation of the EXAMINER.

3.1.4 Dipping into Desired Annotations

Where annotations are desired rather than required they present a possible point at which extra information can be introduced to help complete a proof. For example, in generating runtime VCs, an invariant need not be specified. In these cases a default invariant is automatically inserted. However, this may not be sufficiently strong to complete the proof. Thus the user may add their own stronger invariant instead.

These additional annotations extend the knowledge to a suitable level for proof. Where dealing with runtime VCs, there is no need to provide specification VC strength annotations. This means that, although sharing the same annotation keywords, the kind of annotations seen for runtime VCs and specification VCs are quite different. This is illustrated in Figure 1, where the exception proofs only dip into the Proof Annotations while the specification proofs make heavy use of Proof Annotations.

Finally note that although the nature of runtime-kind and specification-kind of annotations may be different, they can co-exist. The runtime-kind and specification-kind of information should both true in each others context. However, where proving VCs of one kind, information of the other kind may only serve to clutter things up.

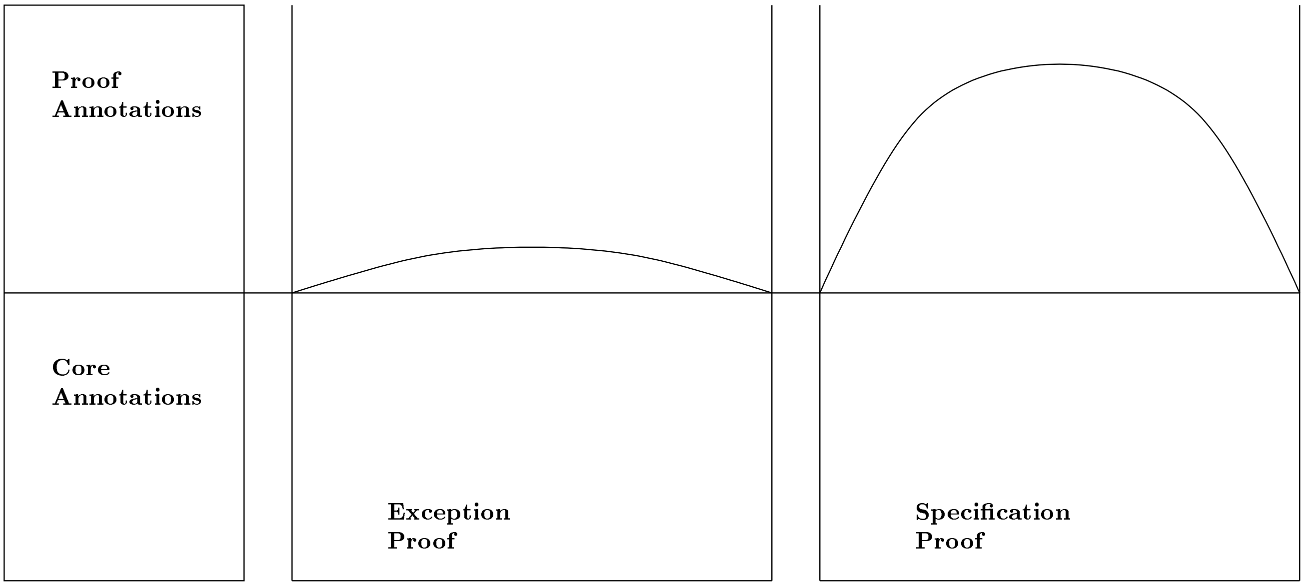


Figure 1: Exception proofs only dip into Proof Annotations

4 About Examples

In the following examples a standard table will be used to describe the key details. This table will take the form:

<table>
<tr><td>Generated From:
Generated To:</td><td></td></tr>
<tr><td>To Demonstrate:</td><td><table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td></td><td></td><td></td></tr></table></td></tr>
<tr><td>Proof In:</td><td><table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td></td><td></td><td></td><td></td></tr></table></td></tr>
<tr><td>Key To Proof:</td><td><table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td></tr>
</table>

- **Generated From and To -** The points between which the VC has been generated.
- **To Demonstrate -** Each group of conclusions is listed, marking these as exception checks if appropriate and giving a description of what the group of conclusions are showing.
- **Proof In -** The system in which the most automatic proof of each conclusion can be established. Note that every conclusion proved by the SIMPLIFIER could also be proved by SPADE, however the SIMPLIFIER always takes precedence as it is more automatic.
- **Key To Proof -** The key information used in finding a proof. Where a proof is not possible all of these will be marked '-', otherwise the following definitions will be used:
  - **RLS -** Using information from the RLS rule file.
  - **User Annotation -** Using information provided through the user annotation. For example, a user provided invariant.
  - EXAMINER **Annotation -** Using the extra annotation inserted by the EXAMINER. This is the generated (or modified part of the) invariant.
  - **Results From -** The key to many exception proofs is the information made available by assuming the successful discharge of an earlier VC. This lists the numbers of any VCs that introduced information key to reaching a proof of this conclusion. Note that the word 'Import' may also be used here, to refer to the extra import information inserted by the EXAMINER (see section 3.1.1).
  - **Proof Work -** A scaled indicator of the complexity in achieving a proof. Note that it is not necessarily the case that a non-complex proof will be proved by the SIMPLIFIER. The following gives an rough description for each scale:

1	Direct match with hypothesis.
2	Trivial or light-weight proof work. For example reasoning about inequalities.
3	A clever insight is required. Typically these should be just outside the scope of the SIMPLIFIER.
4	A number of insights are required. These are challenging proofs that have a number of steps.
5	Proof involves deep insight, including but not limited to the generation of lemmas to complete a proof.

5 Partition Example - Proof Annotated

See Appendix A and B for full details on this example.

5.1 procedure_partition_section_1

Generated From: Start.
Generated To: Run-time check associated with statement of line 11: `I:=Flag'First`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That `Flag'First` is inside the range of `I`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		
C2	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1	X				1
C2	X				1

5.2 procedure_partition_section_2

Generated From: Start.
Generated To: Run-time check associated with statement of line 12: `J:=Flag'Last+1`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That `Flag'Last+1` is inside the range of `J`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		
C2	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1	X				1
C2	X				1

5.3 procedure_partition_section_3

Generated From: **Generated To:**	Start. Assertion of line 15.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1, C2, C3, C4, C5</td><td></td><td>That the user provided invariant follows from the pre-condition.</td></tr><tr><td>C6, C7</td><td></td><td>That the EXAMINER extended invariant follows from the pre-condition.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td>X</td><td></td><td></td></tr><tr><td>C2</td><td>X</td><td></td><td></td></tr><tr><td>C3</td><td>X</td><td></td><td></td></tr><tr><td>C4</td><td></td><td>X</td><td></td></tr><tr><td>C5</td><td></td><td>X</td><td></td></tr><tr><td>C6</td><td>X</td><td></td><td></td></tr><tr><td>C7</td><td>X</td><td></td><td></td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td>X</td><td></td><td></td><td></td><td>1</td></tr><tr><td>C2</td><td>X</td><td></td><td></td><td></td><td>1</td></tr><tr><td>C3</td><td>X</td><td></td><td></td><td></td><td>1</td></tr><tr><td>C4</td><td></td><td></td><td></td><td></td><td>3</td></tr><tr><td>C5</td><td></td><td></td><td></td><td></td><td>3</td></tr><tr><td>C6</td><td></td><td></td><td></td><td>Import</td><td>1</td></tr><tr><td>C7</td><td></td><td>X</td><td></td><td></td><td>1</td></tr></table>

5.4 procedure_partition_section_4

Generated From: **Generated To:**	Assertion of line 15. Assertion of line 15, taking the then branch.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2, C3, C4, C5		That the user provided invariant on one iteration follows from the user provided invariant on a subsequent iteration.
C6, C7		That the EXAMINER extended invariant on one iteration follows from the EXAMINER extended invariant on a subsequent iteration.

Proof In:

	SIMPLIFIER	SPADE	**Not provable**
C1	X		
C2	X		
C3	X		
C4		X	
C5	X		
C6	X		
C7	X		

Key To Proof:

	RLS	**User Annotation**	EXAMINER **Annotation**	**Results From**	**Proof Work (effort: 1→5)**
C1		X			2
C2		X			1
C3		X			2
C4		X			3
C5		X			1
C6			X		1
C7			X		1

5.5 procedure_partition_section_5

Generated From: **Generated To:**	Assertion of line 15. Assertion of line 15, taking the else branch.
To Demonstrate:	see below

Conclusions	Exception	Demonstrate
C1, C2, C3, C4, C5		That the user provided invariant on one iteration follows from the user provided invariant on a subsequent iteration.
C6, C7		That the EXAMINER extended invariant on one iteration follows from the EXAMINER extended invariant on a subsequent iteration.

Proof In:

	SIMPLIFIER	SPADE	**Not provable**
C1	X		
C2	X		
C3	X		
C4		X	
C5		X	
C6		X	
C7	X		

Key To Proof:

	RLS	**User Annotation**	EXAMINER **Annotation**	**Results From**	**Proof Work (effort: 1→5)**
C1		X			1
C2		X			2
C3		X			2
C4		X			3
C5		X			3
C6			X		3
C7			X		1

5.6 procedure_partition_section_6

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 22: `Flag(I)=Red`.
To Demonstrate:	see below

Conclusions	Exception	Demonstrate
C1, C2	X	That `Flag(I)` is using a value for `I` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	**Not provable**
C1	X		
C2		X	

Key To Proof:

	RLS	**User Annotation**	EXAMINER **Annotation**	**Results From**	**Proof Work (effort: 1→5)**
C1				4, 5	1
C2				4, 5	2

5.7 procedure_partition_section_7

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 23: `I:=I+1`.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1, C2</td><td>X</td><td>That the value assigned to <code>I</code> in <code>I:=I+1</code> is within the range of <code>I</code>. Note that <code>I</code> is an integer of type <code>JustBiggerRange</code>.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td>X</td><td></td><td></td></tr><tr><td>C2</td><td>X</td><td></td><td></td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td></td><td></td><td></td><td>6</td><td>2</td></tr><tr><td>C2</td><td></td><td></td><td></td><td>6</td><td>2</td></tr></table>

5.8 procedure_partition_section_8

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 25: `J:=J-1`.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1, C2</td><td>X</td><td>That the value assigned to <code>J</code> in <code>J:=J-1</code> is within the range of <code>J</code>. Note that <code>J</code> is an integer of type <code>JustBiggerRange</code>.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td>X</td><td></td><td></td></tr><tr><td>C2</td><td>X</td><td></td><td></td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td>X</td><td></td><td></td><td>4, 5</td><td>2</td></tr><tr><td>C2</td><td>X</td><td></td><td></td><td>4, 5</td><td>2</td></tr></table>

5.9 procedure_partition_section_9

Generated From:	Assertion of line 15.
Generated To:	Run-time check associated with statement of line 26: `T:=Flag(I)`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That the value assigned to `T` in `T:=Flag(I)` is within the range of `T`. Note that `T` is an enumerated type of range `Colour`.
C3, C4	X	That `Flag(I)` is using a value for `I` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	**Not provable**
C1		X	
C2		X	
C3	X		
C4	X		

Key To Proof:

	RLS	**User Annotation**	EXAMINER **Annotation**	**Results From**	**Proof Work (effort: 1→5)**
C1				4, 5	2
C2				4, 5	2
C3				6	1
C4				6	1

5.10 procedure_partition_section_10

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 27: `Flag(I):=Flag(J)`.
To Demonstrate:	see table below
Proof In:	see table below
Key To Proof:	see table below

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That the value assigned to `Flag(I)` in `Flag(I):=Flag(J)` is within the range of `Flag(I)`. Note that `Flag(I)` is an array element of enumerated type of range `Colour`.
C3, C4	X	That `Flag(J)` is using a value for `J` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.
C5, C6	X	That `Flag(I)` is using a value for `I` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1		X	
C2		X	
C3	X		
C4	X		
C5	X		
C6	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1				4, 5	2
C2				4, 5	2
C3	X			8	1
C4	X			4, 5, 8	1
C5				6	1
C6				6	1

5.11 procedure_partition_section_11

Generated From:	Assertion of line 15.
Generated To:	Run-time check associated with statement of line 28: `Flag(J):=T`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That the value assigned to `Flag(J)` in `Flag(J):=T` is within the range of `Flag(J)`. Note that `Flag(J)` is an array element of enumerated type of range `Colour`.
C3, C4	X	That `Flag(J)` is using a value for `J` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		
C2	X		
C3	X		
C4	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1				9	1
C2				9	1
C3				8	1
C4				8	1

5.12 procedure_partition_section_12

Generated From:	Assertion of line 15.
Generated To:	Finish.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1		That the post-condition follows from the invariant where the loop terminates.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1		X	

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1				4, 5	4

5.13 Notes

5.13.1 Extra Import Information

The hypothesis H2 in procedure_partition_section_1 is generated as a result of the EXAMINER adding import information (see section 3.1.1). It is repeated below: EXAMINER

```
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
```

This expresses that the imported variable Flag must meet its type specification on entering the procedure. Note that this is a helpful piece of information, and can be used to prove C1 and C2 of procedure_partition_section_9 and C1 and C2 of procedure_partition_section_10.

The user supplied pre-condition contains a very similar statement to the one generated above. However, this pre-condition is not repeated in the invariant and thus does not exist to discharge procedure_partition_section_9 and procedure_partition_section_10.

5.13.2 Extra Invariant Information

The hypothesis H6 and H7 in procedure_partition_section_4 (and elsewhere) are generated as a result of the EXAMINER modified invariant (see section 3.1.2). These hypothesis are repeated below:

```
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
           red) or (element(flag~, [i_]) = white))) .
```

The modified invariant is successful in carrying through the added import information, for use in proving procedure_partition_section_9 and procedure_partition_section_10 as noted above.

5.13.3 Extra Proof Obligations

Perhaps not surprisingly, the extra information inserted by the EXAMINER leads to extra conclusions that must be solved.

5.13.4 Using The RLS Rule File

The proof of many runtime VCs make use of known constants. These are held in an RLS rule file generated by the EXAMINER. For example, the following lines were particularly useful in this example:

```
partition_se_rules(3): indexrange__first may_be_replaced_by 1.
partition_se_rules(4): indexrange__last may_be_replaced_by 4.
partition_se_rules(30): justbiggerrange__first may_be_replaced_by 1.
partition_se_rules(31): justbiggerrange__last may_be_replaced_by 5.
```

5.13.5 Using Numeric Constants

In many proofs it is necessary to reason about variables in relationship to other variables. Where there are no hypothesis that express such relationships then these proofs become impossible. However, this can often be overcome by replacing the troublesome variables with numeric constants. This establishes a strong relationship between the variables making proof more likely. For example, consider:

```
H1:    true.
        ->
C1:    x >= y.
```

This can not be proved, as there is no information relating x to y. However, if the following rules are available a proof is trivial:

```
rule(1): x may_be_replaced_by 100.
rule(2): y may_be_replaced_by 10.
```

Note that it may be the case that the need for constants is only required because there is not sufficiently strong relationships described formally in the annotations. Further, in the proof of more abstract procedures and functions such constants may not exist.

5.13.6 Power of the SIMPLIFIER

One of the major advantages of conducting a proof of exception freedom is the ability of the SIMPLIFIER to automatically discharge a large number of the VCs. While it is not expected for the SIMPLIFIER to prove conclusions of proof work 3 or higher, there are a few examples where the SIMPLIFIER could conceivably be improved[9]:

- **Reaching the SIMPLIFIER limits**

  In `procedure_partition_section_6` the SIMPLIFIER does not solve the following:

  ```
  procedure_partition_section_6.
  H1:    1 <= i .
  H2:    j <= 5 .
  H3:    i <= j .
  ...
  H8:    i <> j .
         ->
  C1:    i <= 4 .
  ```

  The solution is to infer `i < j` from `i <= j` and `i <> j`. Then infer `i < 5` from `i < j` and `j <= 5`. Finally infer that `i <= 4` is true from `j <= 5`.

  This example seems to be just above what the SIMPLIFIER is capable of proving. It may present a starting point from which possible enhancements are proposed. As these kinds of problems may be tackled inside a decision procedure, it is possible that existing systems may already embody the required functionality. Such systems may be exploited in building these enhancements into NuSPADE.

- **Avoid quantification**

  In `procedure_partition_section_9` the SIMPLIFIER does not solve the following:

  ```
  H1:    1 <= i .
  H2:    j <= 5 .
  H3:    i <= j .
  ...
  H6:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
            [i___1]) and element(flag,[i___1]) <= white) .
  ...
  H8:    i <> j .
  H9:    i <= 4 .
  H10:   element(flag,[i]) <> red .
  H11:   j >= 2 .
         ->
  C1:    red <= element(flag,[i]) .
  C2:    element(flag,[i]) <= white .
  ```

  The solution is to strip off (unwrap) the quantifier in H6 and instantiate it to be i. This gives:

  ```
  H12:  i >= 1 and i <= 4 -> red <= element(flag, [i]) and element(flag,
        [i]) <= white
  ```

  It can be show that `i` falls within this range, using H1 and H9 directly (forwardchain on H12). This gives access to a modified H12:

[9] Or a further tool, possibly NuSPADE, could be used to provide deeper automatic proof of exception freedom problems.

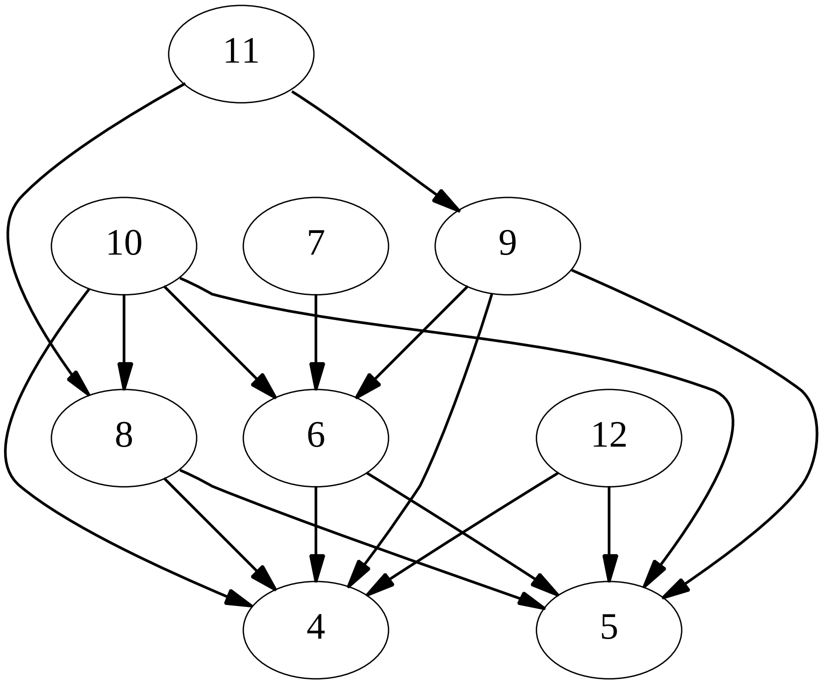
```
H12:  red <= element(flag, [i]) and element(flag, [i]) <= white
```

Which, after normalisation[10], matches with C1 and C2.

The SIMPLIFIER does not get this proof as it does not consider quantification[11]. A very limited support for qualification might be enough to prove problems as seen above. This is expected to occur fairly frequently, as the quantified expression was originally inserted by the EXAMINER. Other program proofs would have similar quantified expressions inserted and thus have the potential for encountering the same sticking point.

5.13.7 Proof Chain

It is common for the proof of one VC to rely on the proof of a previous VC. For example, most of the proofs in the above example make use of the information made available through the proof of `procedure_partition_section_4` and `procedure_partition_section_5`. The complete nature of the proof chain for this example is shown in Figure 2.



(Node N ⇒ `procedure_partition_section_N`)

Figure 2: Proof chain (Proof Annotated example)

Note how information introduced through the invariant is at the root of the proof chain (nodes 4 and 5). This illustrates how important a good invariant is in completing the proof of this problem.

[10] In this case, breaking the single expression into its two conjuncts.

[11] This is no longer the case! The latest version of the SIMPLIFIER is reported to deal with exactly the kind of problem considered here.

6 Partition Example - Core Annotated

See Appendix A and C for full details on this example.

6.1 procedure_partition_section_1

Generated From: Start.
Generated To: Run-time check associated with statement of line 11 `I:=Flag'First`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That `Flag'First` is inside the range of `I`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		
C2	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1	X				1
C2	X				1

6.2 procedure_partition_section_2

Generated From: Start.
Generated To: Run-time check associated with statement of line 12 `J:=Flag'Last+1`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That `Flag'Last+1` is inside the range of `J`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		
C2	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1	X				1
C2	X				1

6.3 procedure_partition_section_3

Generated From: Start.
Generated To: Assertion of line 15.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1		That the EXAMINER extended invariant follows from the pre-condition.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1			X	Import	1

6.4 procedure_partition_section_4

Generated From: Assertion of line 15.
Generated To: Assertion of line 15, taking the then branch.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1	X	That the EXAMINER extended invariant on one iteration follows from the EXAMINER extended invariant on a subsequent iteration.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1			X		1

6.5 procedure_partition_section_5

Generated From: Assertion of line 15.
Generated To: Assertion of line 15, taking the else branch.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1	X	That the EXAMINER extended invariant on one iteration follows from the EXAMINER extended invariant on a subsequent iteration.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1		X	

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1			X		3

6.6 procedure_partition_section_6

Generated From: Assertion of line 15.
Generated To: Run-time check associated with statement of line 22: `Flag(I)=Red`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That `Flag(I)` is using a value for `I` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1			X
C2			X

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1	-	-	-	-	-
C2	-	-	-	-	-

6.7 procedure_partition_section_7

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 23: `I:=I+1`.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1, C2</td><td>X</td><td>That the value assigned to <code>I</code> in <code>I:=I+1</code> is within the range of <code>I</code>. Note that <code>I</code> is an integer of type <code>JustBiggerRange</code>.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td>X</td><td></td><td></td></tr><tr><td>C2</td><td>X</td><td></td><td></td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td>X</td><td></td><td></td><td>6</td><td>2</td></tr><tr><td>C2</td><td>X</td><td></td><td></td><td>6</td><td>2</td></tr></table>

6.8 procedure_partition_section_8

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 25: `J:=J-1`.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1, C2</td><td>X</td><td>That the value assigned to <code>J</code> in <code>J:=J-1</code> is within the range of <code>J</code>. Note that <code>J</code> is an integer of type <code>JustBiggerRange</code>.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td></td><td></td><td>X</td></tr><tr><td>C2</td><td></td><td></td><td>X</td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr><tr><td>C2</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table>

6.9 procedure_partition_section_9

Generated From:	Assertion of line 15.
Generated To:	Run-time check associated with statement of line 26: `T:=Flag(I)`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That the value assigned to `T` in `T:=Flag(I)` is within the range of `T`. Note that `T` is an enumerated type of range `Colour`.
C3, C4	X	That `Flag(I)` is using a value for `I` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1		X	
C2		X	
C3	X		
C4	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1			X		2
C2			X		2
C3				6	1
C4				6	1

6.10 procedure_partition_section_10

Generated From:	Assertion of line 15.
Generated To:	Run-time check associated with statement of line 27: `Flag(I):=Flag(J)`.

To Demonstrate:

Conclusions	Exception	Demonstrate
C1, C2	X	That the value assigned to `Flag(I)` in `Flag(I):=Flag(J)` is within the range of `Flag(I)`. Note that `Flag(I)` is an array element of enumerated type of range `Colour`.
C3, C4	X	That `Flag(J)` is using a value for `J` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.
C5, C6	X	That `Flag(I)` is using a value for `I` inside the range of array `Flag`. Note that `Flag` is an array of size `IndexRange`.

Proof In:

	SIMPLIFIER	SPADE	Not provable
C1		X	
C2		X	
C3	X		
C4		X	
C5	X		
C6	X		

Key To Proof:

	RLS	User Annotation	EXAMINER Annotation	Results From	Proof Work (effort: 1→5)
C1			X		2
C2			X		2
C3				8	2
C4				8	2
C5				6	1
C6				6	1

6.11 procedure_partition_section_11

Generated From: **Generated To:**	Assertion of line 15. Run-time check associated with statement of line 28: `Flag(J):=T`.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1, C2</td><td>X</td><td>That the value assigned to <code>Flag(J)</code> in <code>Flag(J):=T</code> is within the range of <code>Flag(J)</code>. Note that <code>Flag(J)</code> is an array element of enumerated type of range <code>Colour</code>.</td></tr><tr><td>C3, C4</td><td>X</td><td>That <code>Flag(J)</code> is using a value for <code>J</code> inside the range of array <code>Flag</code>. Note that <code>Flag</code> is an array of size <code>IndexRange</code>.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td>X</td><td></td><td></td></tr><tr><td>C2</td><td>X</td><td></td><td></td></tr><tr><td>C3</td><td>X</td><td></td><td></td></tr><tr><td>C4</td><td>X</td><td></td><td></td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td></td><td></td><td></td><td>9</td><td>1</td></tr><tr><td>C2</td><td></td><td></td><td></td><td>9</td><td>1</td></tr><tr><td>C3</td><td></td><td></td><td></td><td>8</td><td>1</td></tr><tr><td>C4</td><td></td><td></td><td></td><td>8</td><td>1</td></tr></table>

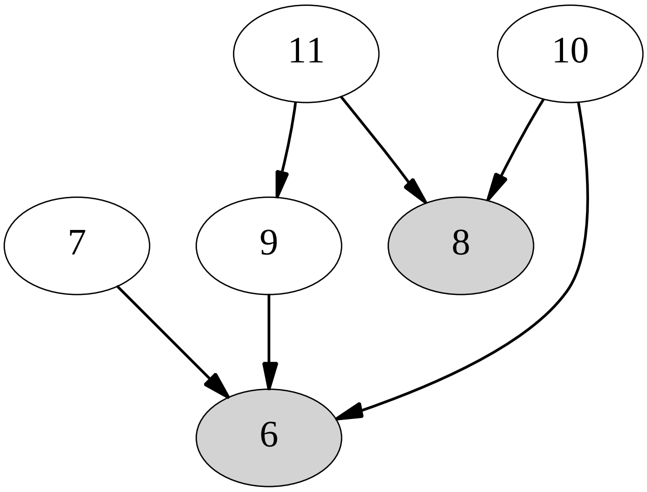
6.12 procedure_partition_section_12

Generated From: **Generated To:**	Assertion of line 15. Finish.
To Demonstrate:	<table><tr><th>Conclusions</th><th>Exception</th><th>Demonstrate</th></tr><tr><td>C1</td><td>X</td><td>That the post-condition follows from the invariant where the loop terminates.</td></tr></table>
Proof In:	<table><tr><th></th><th>SIMPLIFIER</th><th>SPADE</th><th>Not provable</th></tr><tr><td>C1</td><td>X</td><td></td><td></td></tr></table>
Key To Proof:	<table><tr><th></th><th>RLS</th><th>User Annotation</th><th>EXAMINER Annotation</th><th>Results From</th><th>Proof Work (effort: 1→5)</th></tr><tr><td>C1</td><td></td><td></td><td></td><td></td><td>1</td></tr></table>

6.13 Notes

6.13.1 Proof Chain

As for the Proof Annotated example, a proof chain is constructed. This is shown in Figure 3.



(Node N ⇒ `procedure_partition_section_N`)
(The gray nodes (8 and 6) were impossible to prove.)

Figure 3: Proof chain (Core Annotated example)

Note that this is an exact subset of the proof chain established for the Proof Annotated example. The reduced number of nodes, and the improvable 8 and 6, are a result of using a much weaker invariant.

The proof chain gives a strong indication that somehow proving nodes 8 and 6 will be sufficient to prove nodes 7,9,10 and 11. The absence of the key invariant nodes (nodes 4 and 5), and the existence of improvable VCs, indicate that the invariant should be strengthened.

6.14 Completing the Proof

There are a few key points in the Core Annotation example where there was not enough information to solve the runtime VCs.

6.14.1 procedure_partition_section_6

Prior to simplification this is:

```
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
        ->
C1:    i >= indexrange__first .
C2:    i <= indexrange__last .
```

And after simplification:

```
H1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H2:    i <> j .
       ->
```

```
C1:    i >= 1 .
C2:    i <= 4 .
```

The variables `I` and `J` are declared to be of type `JustBiggerRange`. This type information can be added to the invariant.

```
    --#        assert ...
    --#        Flag'First<=I and I<=Flag'Last+1 and
    --#        Flag'First<=J and J<=Flag'Last+1
    --#        ...
```

Prior to simplification this gives:

```
H1:    indexrange__first <= i .
H2:    i <= indexrange__last + 1 .
H3:    indexrange__first <= j .
H4:    j <= indexrange__last + 1 .
H5:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H6:    not (i = j) .
        ->
C1:    i >= indexrange__first .
C2:    i <= indexrange__last .
```

And after simplification:

```
H1:    1 <= i .
H2:    i <= 5 .
H3:    1 <= j .
H4:    j <= 5 .
H5:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H6:    i <> j .
       ->
C1:    i <= 4 .
```

Adding the type information alone is not sufficient, as there is not suitably strong information linking `I` with `J`. This is mainly due to the loop terminating on an equality rather than inequality test. Adding the knowledge that `I<=J` is sufficient to complete the proof.

```
    --#        assert ...
    --#        Flag'First<=I and I<=Flag'Last+1 and
    --#        Flag'First<=J and J<=Flag'Last+1 and
    --#        I<=J
    --#        ...
```

6.14.2 procedure_partition_section_8

Prior to simplification this is:

```
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    not (element(flag, [i]) = red) .
        ->
C1:    j - 1 >= justbiggerrange__first .
C2:    j - 1 <= justbiggerrange__last .
```

And after simplification:

```
H1:     for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
           [i___1]) and element(flag,[i___1]) <= white) .
H2:     i <> j .
H3:     i >= 1 .
H4:     i <= 4 .
H5:     element(flag,[i]) <> red .
        ->
C1:     j >= 2 .
C2:     j <= 6 .
```

This is essentially the same problem as for `procedure_partition_section_6` and is also discharged by the annotation suggested for this.

7 Conclusions

Performing a straight forward analysis of the source code and adding type information alone is sufficient to advance proofs of exception freedom in code that only has Core Annotations. However, this is not usually sufficient to complete these proofs.

Additional information may be derived though bottom-up analysis. This was originally explored to generate the last extra information to complete specification proofs (See CS-note 10). It is interesting that it also appears to generate the extra information required in exception freedom proofs. In a few cases the SIMPLIFIER is not sufficiently powerfully to automatically discharge the runtime VCs. There is scope for NuSPADE to automate these proofs.

There is no single quick fix to automate the runtime VCs. However, the combination of a number of different strategies may bring some of the automation desired. As some of these strategies are already being proposed with NuSPADE it is possible that the additional effort required to also capture exception freedom proofs will be minimal.

References

[1] R. Chapman. Industrial experience with spark, 2000.

[2] Martin Croxford and James Sutton. Breaking through the v and v bottleneck, 1995.

[3] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system, 2002.

[4] S.King, J. Hammond, and R. Chapman A. Pryor. Is proof more cost-effective than testing? volume 26, pages 675–686. IEEE, 2000.

A Partition Example - Common semantic information

A.1 Generated RLS File

```
          /*********************************************************/
                          /*Proof Rule Declarations*/
       /*SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01*/
                           /*Demonstration Version*/
          /*********************************************************/


                       /*DATE : 28-MAR-2002 10:29:32.22*/

                   /*procedure PolishFlag.Partition_Section*/


rule_family partition_se_rules:
     X      requires [X:any] &
     X <= Y requires [X:ire, Y:ire] &
     X >= Y requires [X:ire, Y:ire].

partition_se_rules(1): character__pos(X) may_be_replaced_by X.
partition_se_rules(2): character__val(X) may_be_replaced_by X.
partition_se_rules(3): indexrange__first may_be_replaced_by 1.
partition_se_rules(4): indexrange__last may_be_replaced_by 4.
partition_se_rules(5): indexrange__base__first <= indexrange__base__last may_be_deduced.
partition_se_rules(6): indexrange__base__first <= indexrange__first may_be_deduced.
partition_se_rules(7): indexrange__base__last >= indexrange__last may_be_deduced.
partition_se_rules(8): colour__first may_be_replaced_by red.
partition_se_rules(9): colour__last may_be_replaced_by white.
partition_se_rules(10): colour__base__first may_be_replaced_by red.
partition_se_rules(11): colour__base__last may_be_replaced_by white.
partition_se_rules(12): colour__pos(colour__first) may_be_replaced_by 0.
partition_se_rules(13): colour__pos(red) may_be_replaced_by 0.
partition_se_rules(14): colour__val(0) may_be_replaced_by red.
partition_se_rules(15): colour__pos(white) may_be_replaced_by 1.
partition_se_rules(16): colour__val(1) may_be_replaced_by white.
partition_se_rules(17): colour__pos(colour__last) may_be_replaced_by 1.
partition_se_rules(18): colour__pos(succ(X)) may_be_replaced_by
     colour__pos(X) + 1
     if [X <=white, X <> white].
partition_se_rules(19): colour__pos(pred(X)) may_be_replaced_by
     colour__pos(X) - 1
     if [X >=red, X <> red].
partition_se_rules(20): colour__pos(X) >= 0 may_be_deduced_from
     [red <= X, X <= white].
partition_se_rules(21): colour__pos(X) <= 1 may_be_deduced_from
     [red <= X, X <= white].
partition_se_rules(22): colour__val(X) >= red may_be_deduced_from
     [0 <= X, X <= 1].
partition_se_rules(23): colour__val(X) <= white may_be_deduced_from
     [0 <= X, X <= 1].
partition_se_rules(24): succ(colour__val(X)) may_be_replaced_by
     colour__val(X+1)
     if [0 <= X, X < 1].
partition_se_rules(25): pred(colour__val(X)) may_be_replaced_by
     colour__val(X-1)
     if [0 < X, X <= 1].
partition_se_rules(26): colour__pos(colour__val(X)) may_be_replaced_by X
     if [0 <= X, X <= 1].
partition_se_rules(27): colour__val(colour__pos(X)) may_be_replaced_by X
     if [red <= X, X <= white].
partition_se_rules(28): colour__pos(X) <= colour__pos(Y) & X <= Y are_interchangeable
     if [red <= X, X <= white].
partition_se_rules(29): colour__val(X) <= colour__val(Y) & X <= Y are_interchangeable
     if [0 <= X, X <= 1].
partition_se_rules(30): justbiggerrange__first may_be_replaced_by 1.
partition_se_rules(31): justbiggerrange__last may_be_replaced_by 5.
partition_se_rules(32): justbiggerrange__base__first <=
     justbiggerrange__base__last may_be_deduced.
partition_se_rules(33): justbiggerrange__base__first <=
     justbiggerrange__first may_be_deduced.
partition_se_rules(34): justbiggerrange__base__last >=
     justbiggerrange__last may_be_deduced.
```

A.2 Generated FDL File

```
          {*******************************************************}
                              {FDL Declarations}
       {SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01}
                           {Demonstration Version}
          {*******************************************************}


                       {DATE : 28-MAR-2002 10:29:32.20}

                   {procedure PolishFlag.Partition_Section}


title procedure partition_section;

  function round__(real) : integer;
  type colour = (red, white);
  type arrayofcolours = array [integer] of colour;
  const justbiggerrange__last : integer = pending;
  const justbiggerrange__first : integer = pending;
  const colour__last : colour = pending;
  const colour__first : colour = pending;
  const indexrange__last : integer = pending;
  const indexrange__first : integer = pending;
  var j : integer;
  var i : integer;
  var flag : arrayofcolours;

end;
```

B Partition Example - Proof Annotated

B.1 The Code

B.1.1 Specification (polishflag_proof.ads)

```
package PolishFlag
is
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    -- Type declerations for flags.
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    -- The range of the array used to hold flags.
    IndexUpper: constant := 4;
    IndexLower: constant := 1;
    subtype IndexRange is Integer range IndexLower .. IndexUpper;

    -- An enumerated type colour. This can take only two
    -- different values.
    type Colour is (Red, White);

    -- An array type, of range IndexRange holding values
    -- of type Colour.
    type ArrayOfColours is array (IndexRange) of Colour;

    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    -- Procedures for working with polish flags.
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    procedure Partition_Section(Flag: in out ArrayOfColours);
    --# derives Flag from Flag;
    --# pre (for all I in IndexRange => (Flag(I)=Red or Flag(I)=White));
    --# post for some P in Integer range (Flag'First) .. (Flag'Last+1) =>
    --# ((for all Q in Integer range Flag'First..(P-1) =>
    --#                 (Flag(Q)=Red)) and
    --#  (for all R in Integer range P..Flag'Last =>
    --#                 (Flag(R)=White)));

end PolishFlag;

--End of file
```

B.1.2 Body (polishflag_proof.adb)

```
package body PolishFlag is

    -- Partition the colours of flag.
    procedure Partition_Section(Flag: in out ArrayOfColours)
    is
        subtype JustBiggerRange is Integer range Flag'First .. Flag'Last+1;
        I: JustBiggerRange;
        J: JustBiggerRange;
        T: Colour;
    begin
        I:=Flag'First;
        J:=Flag'Last+1;

        loop
            --# assert Flag'First<=I and
            --#        J<=(Flag'Last+1) and
            --#        I<=J and
            --# (for all Q in Integer range Flag'First..(I-1) => (Flag(Q)=Red)) and
            --# (for all R in Integer range J..Flag'Last => (Flag(R)=White));
            exit when I=J;

            if Flag(I)=Red then
                I:=I+1;
            else
                J:=J-1;
                T:=Flag(I);
                Flag(I):=Flag(J);
                Flag(J):=T;
            end if;
        end loop;
```

```
31
32     end Partition_Section;
33
34  end PolishFlag;
```

B.2 Runtime VCs

B.2.1 Unsimplified

```
          *******************************************************
                        Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01
                             Demonstration Version
          *******************************************************


                         DATE : 28-MAR-2002 10:29:32.19

                     procedure PolishFlag.Partition_Section




For path(s) from start to run-time check associated with statement of line 11:

procedure_partition_section_1.
H1:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag, [i_]) =
           red) or (element(flag, [i_]) = white))) .
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
        ->
C1:    indexrange__first >= justbiggerrange__first .
C2:    indexrange__first <= justbiggerrange__last .


For path(s) from start to run-time check associated with statement of line 12:

procedure_partition_section_2.
H1:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag, [i_]) =
           red) or (element(flag, [i_]) = white))) .
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H3:    indexrange__first >= justbiggerrange__first .
H4:    indexrange__first <= justbiggerrange__last .
        ->
C1:    indexrange__last + 1 >= justbiggerrange__first .
C2:    indexrange__last + 1 <= justbiggerrange__last .


For path(s) from start to assertion of line 15:

procedure_partition_section_3.
H1:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag, [i_]) =
           red) or (element(flag, [i_]) = white))) .
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H3:    indexrange__first >= justbiggerrange__first .
H4:    indexrange__first <= justbiggerrange__last .
H5:    indexrange__last + 1 >= justbiggerrange__first .
H6:    indexrange__last + 1 <= justbiggerrange__last .
        ->
C1:    indexrange__first <= indexrange__first .
C2:    indexrange__last + 1 <= indexrange__last + 1 .
C3:    indexrange__first <= indexrange__last + 1 .
C4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
```

```
          q_ <= indexrange__first - 1)) -> (element(flag, [
          q_]) = red)) .
C5:     for_all (r_: integer, ((r_ >= indexrange__last + 1) and (
          r_ <= indexrange__last)) -> (element(flag, [r_]) =
          white)) .
C6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
C7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag, [i_]) =
          red) or (element(flag, [i_]) = white))) .


For path(s) from assertion of line 15 to assertion of line 15:

procedure_partition_section_4.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
          q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
          indexrange__last)) -> (element(flag, [r_]) =
          white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
          red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
H10:   i <= indexrange__last .
H11:   element(flag, [i]) = red .
H12:   i + 1 >= justbiggerrange__first .
H13:   i + 1 <= justbiggerrange__last .
        ->
C1:    indexrange__first <= i + 1 .
C2:    j <= indexrange__last + 1 .
C3:    i + 1 <= j .
C4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
          q_ <= i + 1 - 1)) -> (element(flag, [q_]) = red)) .
C5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
          indexrange__last)) -> (element(flag, [r_]) =
          white)) .
C6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
C7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
          red) or (element(flag~, [i_]) = white))) .


procedure_partition_section_5.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
          q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
          indexrange__last)) -> (element(flag, [r_]) =
          white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
          red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
```

```
H10:   i <= indexrange__last .
H11:   not (element(flag, [i]) = red) .
H12:   j - 1 >= justbiggerrange__first .
H13:   j - 1 <= justbiggerrange__last .
H14:   element(flag, [i]) >= colour__first .
H15:   element(flag, [i]) <= colour__last .
H16:   i >= indexrange__first .
H17:   i <= indexrange__last .
H18:   element(flag, [j - 1]) >= colour__first .
H19:   element(flag, [j - 1]) <= colour__last .
H20:   j - 1 >= indexrange__first .
H21:   j - 1 <= indexrange__last .
H22:   i >= indexrange__first .
H23:   i <= indexrange__last .
H24:   element(flag, [i]) >= colour__first .
H25:   element(flag, [i]) <= colour__last .
H26:   j - 1 >= indexrange__first .
H27:   j - 1 <= indexrange__last .
        ->
C1:    indexrange__first <= i .
C2:    j - 1 <= indexrange__last + 1 .
C3:    i <= j - 1 .
C4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(update(update(flag, [i], element(
           flag, [j - 1])), [j - 1], element(flag, [i])), [
           q_]) = red)) .
C5:     for_all (r_: integer, ((r_ >= j - 1) and (r_ <=
           indexrange__last)) -> (element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [r_]) = white)) .
C6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [i___1]) >= colour__first) and (element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [i___1]) <= colour__last))) .
C7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
           red) or (element(flag~, [i_]) = white))) .


For path(s) from assertion of line 15 to run-time check associated with statement of line 22:

procedure_partition_section_6.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
           red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
        ->
C1:    i >= indexrange__first .
C2:    i <= indexrange__last .


For path(s) from assertion of line 15 to run-time check associated with statement of line 23:

procedure_partition_section_7.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
```

```
          indexrange__last)) -> (element(flag, [r_]) =
          white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
          red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
H10:   i <= indexrange__last .
H11:   element(flag, [i]) = red .
        ->
C1:    i + 1 >= justbiggerrange__first .
C2:    i + 1 <= justbiggerrange__last .


For path(s) from assertion of line 15 to run-time check associated with statement of line 25:

procedure_partition_section_8.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
          q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
          indexrange__last)) -> (element(flag, [r_]) =
          white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
          red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
H10:   i <= indexrange__last .
H11:   not (element(flag, [i]) = red) .
        ->
C1:    j - 1 >= justbiggerrange__first .
C2:    j - 1 <= justbiggerrange__last .


For path(s) from assertion of line 15 to run-time check associated with statement of line 26:

procedure_partition_section_9.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
          q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
          indexrange__last)) -> (element(flag, [r_]) =
          white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
          i___1 <= indexrange__last)) -> ((element(flag, [
          i___1]) >= colour__first) and (element(flag, [
          i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
          i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
          red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
H10:   i <= indexrange__last .
H11:   not (element(flag, [i]) = red) .
H12:   j - 1 >= justbiggerrange__first .
H13:   j - 1 <= justbiggerrange__last .
        ->
C1:    element(flag, [i]) >= colour__first .
C2:    element(flag, [i]) <= colour__last .
C3:    i >= indexrange__first .
C4:    i <= indexrange__last .
```

For path(s) from assertion of line 15 to run-time check associated with statement of line 27:

```
procedure_partition_section_10.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
           red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
H10:   i <= indexrange__last .
H11:   not (element(flag, [i]) = red) .
H12:   j - 1 >= justbiggerrange__first .
H13:   j - 1 <= justbiggerrange__last .
H14:   element(flag, [i]) >= colour__first .
H15:   element(flag, [i]) <= colour__last .
H16:   i >= indexrange__first .
H17:   i <= indexrange__last .
        ->
C1:    element(flag, [j - 1]) >= colour__first .
C2:    element(flag, [j - 1]) <= colour__last .
C3:    j - 1 >= indexrange__first .
C4:    j - 1 <= indexrange__last .
C5:    i >= indexrange__first .
C6:    i <= indexrange__last .
```

For path(s) from assertion of line 15 to run-time check associated with statement of line 28:

```
procedure_partition_section_11.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
           red) or (element(flag~, [i_]) = white))) .
H8:    not (i = j) .
H9:    i >= indexrange__first .
H10:   i <= indexrange__last .
H11:   not (element(flag, [i]) = red) .
H12:   j - 1 >= justbiggerrange__first .
H13:   j - 1 <= justbiggerrange__last .
H14:   element(flag, [i]) >= colour__first .
H15:   element(flag, [i]) <= colour__last .
H16:   i >= indexrange__first .
H17:   i <= indexrange__last .
H18:   element(flag, [j - 1]) >= colour__first .
H19:   element(flag, [j - 1]) <= colour__last .
H20:   j - 1 >= indexrange__first .
H21:   j - 1 <= indexrange__last .
H22:   i >= indexrange__first .
H23:   i <= indexrange__last .
        ->
```

```
C1:    element(flag, [i]) >= colour__first .
C2:    element(flag, [i]) <= colour__last .
C3:    j - 1 >= indexrange__first .
C4:    j - 1 <= indexrange__last .


For path(s) from assertion of line 15 to finish:

procedure_partition_section_12.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
H3:    i <= j .
H4:     for_all (q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:     for_all (r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
H6:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H7:     for_all (i_: integer, ((i_ >= indexrange__first) and (
           i_ <= indexrange__last)) -> ((element(flag~, [i_]) =
           red) or (element(flag~, [i_]) = white))) .
H8:    i = j .
        ->
C1:     for_some (p_: integer, ((p_ >= indexrange__first) and (
           p_ <= indexrange__last + 1)) and (( for_all (q_:
           integer, ((q_ >= indexrange__first) and (q_ <= p_ - 1)) -> (element(
           flag, [q_]) = red))) and ( for_all (r_: integer, ((
           r_ >= p_) and (r_ <= indexrange__last)) -> (element(
           flag, [r_]) = white))))) .
```

B.2.2 Simplified

```
          *******************************************************
                      Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01
                           Demonstration Version
          *******************************************************


      CREATED 28-MAR-2002, 10:29:32  SIMPLIFIED 28-MAR-2002, 10:37:52
               (Simplified by SPADE Simplifier, Version 1.45)

                   procedure PolishFlag.Partition_Section




For path(s) from start to run-time check associated with statement of line 11:

procedure_partition_section_1.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 12:

procedure_partition_section_2.
*** true .          /* all conclusions proved */


For path(s) from start to assertion of line 15:

procedure_partition_section_3.
H1:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag,[i_]) = red or
          element(flag,[i_]) = white) .
H2:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
       ->
C1:    for_all(q_ : integer,q_ >= 1 and q_ <= 0 -> element(flag,[q_]) = red) .
C2:    for_all(r_ : integer,r_ >= 5 and r_ <= 4 -> element(flag,[r_]) = white)
```

```
          .


For path(s) from assertion of line 15 to assertion of line 15:

procedure_partition_section_4.
H1:    1 <= i .
H2:    j <= 5 .
H3:    i <= j .
H4:    for_all(q_ : integer,q_ >= 1 and q_ <= i - 1 -> element(flag,[q_]) = red)
           .
H5:    for_all(r_ : integer,r_ >= j and r_ <= 4 -> element(flag,[r_]) = white)
          .
H6:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H7:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag~,[i_]) = red or
          element(flag~,[i_]) = white) .
H8:    i <> j .
H9:    i <= 4 .
H10:   element(flag,[i]) = red .
       ->
C1:    for_all(q_ : integer,q_ >= 1 and q_ <= i -> element(flag,[q_]) = red) .


procedure_partition_section_5.
H1:    1 <= i .
H2:    j <= 5 .
H3:    i <= j .
H4:    for_all(q_ : integer,q_ >= 1 and q_ <= i - 1 -> element(flag,[q_]) = red)
           .
H5:    for_all(r_ : integer,r_ >= j and r_ <= 4 -> element(flag,[r_]) = white)
          .
H6:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H7:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag~,[i_]) = red or
          element(flag~,[i_]) = white) .
H8:    i <> j .
H9:    i <= 4 .
H10:   element(flag,[i]) <> red .
H11:   j >= 2 .
H12:   red <= element(flag,[i]) .
H13:   element(flag,[i]) <= white .
H14:   red <= element(flag,[j - 1]) .
H15:   element(flag,[j - 1]) <= white .
       ->
C1:    for_all(q_ : integer,q_ >= 1 and q_ <= i - 1 -> element(update(update(
          flag,[i],element(flag,[j - 1])),[j - 1],element(flag,[i])),[q_]) =
          red) .
C2:    for_all(r_ : integer,r_ >= j - 1 and r_ <= 4 -> element(update(update(
          flag,[i],element(flag,[j - 1])),[j - 1],element(flag,[i])),[r_]) =
          white) .
C3:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(
          update(update(flag,[i],element(flag,[j - 1])),[j - 1],element(flag,
          [i])),[i___1]) and element(update(update(flag,[i],element(flag,[j -
          1])),[j - 1],element(flag,[i])),[i___1]) <= white) .


For path(s) from assertion of line 15 to run-time check associated with
          statement of line 22:

procedure_partition_section_6.
H1:    1 <= i .
H2:    j <= 5 .
H3:    i <= j .
H4:    for_all(q_ : integer,q_ >= 1 and q_ <= i - 1 -> element(flag,[q_]) = red)
           .
H5:    for_all(r_ : integer,r_ >= j and r_ <= 4 -> element(flag,[r_]) = white)
          .
H6:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H7:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag~,[i_]) = red or
          element(flag~,[i_]) = white) .
H8:    i <> j .
       ->
```

```
C1:    i <= 4 .


For path(s) from assertion of line 15 to run-time check associated with
          statement of line 23:

procedure_partition_section_7.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 15 to run-time check associated with
          statement of line 25:

procedure_partition_section_8.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 15 to run-time check associated with
          statement of line 26:

procedure_partition_section_9.
H1:    1 <= i .
H2:    j <= 5 .
H3:    i <= j .
H4:    for_all(q_ : integer,q_ >= 1 and q_ <= i - 1 -> element(flag,[q_]) = red)
           .
H5:    for_all(r_ : integer,r_ >= j and r_ <= 4 -> element(flag,[r_]) = white)
          .
H6:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H7:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag~,[i_]) = red or
          element(flag~,[i_]) = white) .
H8:    i <> j .
H9:    i <= 4 .
H10:   element(flag,[i]) <> red .
H11:   j >= 2 .
       ->
C1:    red <= element(flag,[i]) .
C2:    element(flag,[i]) <= white .


For path(s) from assertion of line 15 to run-time check associated with
          statement of line 27:

procedure_partition_section_10.
H1:    1 <= i .
H2:    j <= 5 .
H3:    i <= j .
H4:    for_all(q_ : integer,q_ >= 1 and q_ <= i - 1 -> element(flag,[q_]) = red)
           .
H5:    for_all(r_ : integer,r_ >= j and r_ <= 4 -> element(flag,[r_]) = white)
          .
H6:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H7:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag~,[i_]) = red or
          element(flag~,[i_]) = white) .
H8:    i <> j .
H9:    i <= 4 .
H10:   element(flag,[i]) <> red .
H11:   j >= 2 .
H12:   red <= element(flag,[i]) .
H13:   element(flag,[i]) <= white .
       ->
C1:    red <= element(flag,[j - 1]) .
C2:    element(flag,[j - 1]) <= white .


For path(s) from assertion of line 15 to run-time check associated with
          statement of line 28:

procedure_partition_section_11.
*** true .          /* all conclusions proved */
```

```
For path(s) from assertion of line 15 to finish:

procedure_partition_section_12.
H1:    1 <= j .
H2:    j <= 5 .
H3:    for_all(q_ : integer,q_ >= 1 and q_ <= j - 1 -> element(flag,[q_]) = red)
          .
H4:    for_all(r_ : integer,r_ >= j and r_ <= 4 -> element(flag,[r_]) = white)
          .
H5:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H6:    for_all(i_ : integer,i_ >= 1 and i_ <= 4 -> element(flag~,[i_]) = red or
          element(flag~,[i_]) = white) .
       ->
C1:    for_some(p_ : integer,p_ >= 1 and p_ <= 5 and (for_all(q_ : integer,q_
          >= 1 and q_ <= p_ - 1 -> element(flag,[q_]) = red) and for_all(r_ :
          integer,r_ >= p_ and r_ <= 4 -> element(flag,[r_]) = white))) .
```

C Partition Example - Core Annotated

C.1 The Code

C.1.1 Specification (polishflag_core.ads)

```
package PolishFlag
is
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    -- Type declerations for flags.
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    -- The range of the array used to hold flags.
    IndexUpper: constant := 4;
    IndexLower: constant := 1;
    subtype IndexRange is Integer range IndexLower .. IndexUpper;

    -- An enumerated type colour. This can take only two
    -- different values.
    type Colour is (Red, White);

    -- An array type, of range IndexRange holding values
    -- of type Colour.
    type ArrayOfColours is array (IndexRange) of Colour;

    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    -- Procedures for working with polish flags.
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    procedure Partition_Section(Flag: in out ArrayOfColours);
    --# derives Flag from Flag;

end PolishFlag;

--End of file
```

C.1.2 Body (polishflag_core.adb)

```
package body PolishFlag is

    -- Partition the colours of flag.
    procedure Partition_Section(Flag: in out ArrayOfColours)
    is
        subtype JustBiggerRange is Integer range Flag'First .. Flag'Last+1;
        I: JustBiggerRange;
        J: JustBiggerRange;
        T: Colour;
    begin
        I:=Flag'First;
        J:=Flag'Last+1;

        loop
            exit when I=J;
            -- |
            -- | Padding to keep line numbers consistient.
            -- |
            -- |
            -- |

            if Flag(I)=Red then
                I:=I+1;
            else
                J:=J-1;
                T:=Flag(I);
                Flag(I):=Flag(J);
                Flag(J):=T;
            end if;
        end loop;

   end Partition_Section;

end PolishFlag;
```

C.2 Runtime VCs

C.2.1 Unsimplified

```
          *******************************************************
                      Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01
                           Demonstration Version
          *******************************************************


                       DATE : 03-APR-2002 10:11:24.85

                   procedure PolishFlag.Partition_Section




For path(s) from start to run-time check associated with statement of line 11:

procedure_partition_section_1.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
        ->
C1:    indexrange__first >= justbiggerrange__first .
C2:    indexrange__first <= justbiggerrange__last .


For path(s) from start to run-time check associated with statement of line 12:

procedure_partition_section_2.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H3:    indexrange__first >= justbiggerrange__first .
H4:    indexrange__first <= justbiggerrange__last .
        ->
C1:    indexrange__last + 1 >= justbiggerrange__first .
C2:    indexrange__last + 1 <= justbiggerrange__last .


For path(s) from start to default assertion of line 15:

procedure_partition_section_3.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H3:    indexrange__first >= justbiggerrange__first .
H4:    indexrange__first <= justbiggerrange__last .
H5:    indexrange__last + 1 >= justbiggerrange__first .
H6:    indexrange__last + 1 <= justbiggerrange__last .
        ->
C1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .


For path(s) from default assertion of line 15 to default assertion of line 15:

procedure_partition_section_4.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
```

```
H5:    element(flag, [i]) = red .
H6:    i + 1 >= justbiggerrange__first .
H7:    i + 1 <= justbiggerrange__last .
        ->
C1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .


procedure_partition_section_5.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    not (element(flag, [i]) = red) .
H6:    j - 1 >= justbiggerrange__first .
H7:    j - 1 <= justbiggerrange__last .
H8:    element(flag, [i]) >= colour__first .
H9:    element(flag, [i]) <= colour__last .
H10:   i >= indexrange__first .
H11:   i <= indexrange__last .
H12:   element(flag, [j - 1]) >= colour__first .
H13:   element(flag, [j - 1]) <= colour__last .
H14:   j - 1 >= indexrange__first .
H15:   j - 1 <= indexrange__last .
H16:   i >= indexrange__first .
H17:   i <= indexrange__last .
H18:   element(flag, [i]) >= colour__first .
H19:   element(flag, [i]) <= colour__last .
H20:   j - 1 >= indexrange__first .
H21:   j - 1 <= indexrange__last .
        ->
C1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [i___1]) >= colour__first) and (element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [i___1]) <= colour__last))) .


For path(s) from default assertion of line 15 to run-time check associated with statement of line 22:

procedure_partition_section_6.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
        ->
C1:    i >= indexrange__first .
C2:    i <= indexrange__last .


For path(s) from default assertion of line 15 to run-time check associated with statement of line 23:

procedure_partition_section_7.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    element(flag, [i]) = red .
        ->
C1:    i + 1 >= justbiggerrange__first .
C2:    i + 1 <= justbiggerrange__last .


For path(s) from default assertion of line 15 to run-time check associated with statement of line 25:
```

```
procedure_partition_section_8.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    not (element(flag, [i]) = red) .
        ->
C1:    j - 1 >= justbiggerrange__first .
C2:    j - 1 <= justbiggerrange__last .
```

For path(s) from default assertion of line 15 to run-time check associated with statement of line 26:

```
procedure_partition_section_9.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    not (element(flag, [i]) = red) .
H6:    j - 1 >= justbiggerrange__first .
H7:    j - 1 <= justbiggerrange__last .
        ->
C1:    element(flag, [i]) >= colour__first .
C2:    element(flag, [i]) <= colour__last .
C3:    i >= indexrange__first .
C4:    i <= indexrange__last .
```

For path(s) from default assertion of line 15 to run-time check associated with statement of line 27:

```
procedure_partition_section_10.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    not (element(flag, [i]) = red) .
H6:    j - 1 >= justbiggerrange__first .
H7:    j - 1 <= justbiggerrange__last .
H8:    element(flag, [i]) >= colour__first .
H9:    element(flag, [i]) <= colour__last .
H10:   i >= indexrange__first .
H11:   i <= indexrange__last .
        ->
C1:    element(flag, [j - 1]) >= colour__first .
C2:    element(flag, [j - 1]) <= colour__last .
C3:    j - 1 >= indexrange__first .
C4:    j - 1 <= indexrange__last .
C5:    i >= indexrange__first .
C6:    i <= indexrange__last .
```

For path(s) from default assertion of line 15 to run-time check associated with statement of line 28:

```
procedure_partition_section_11.
H1:     for_all (i___1: integer, ((i___1 >= indexrange__first) and (
           i___1 <= indexrange__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H2:    not (i = j) .
H3:    i >= indexrange__first .
H4:    i <= indexrange__last .
H5:    not (element(flag, [i]) = red) .
H6:    j - 1 >= justbiggerrange__first .
H7:    j - 1 <= justbiggerrange__last .
```

```
H8:    element(flag, [i]) >= colour__first .
H9:    element(flag, [i]) <= colour__last .
H10:   i >= indexrange__first .
H11:   i <= indexrange__last .
H12:   element(flag, [j - 1]) >= colour__first .
H13:   element(flag, [j - 1]) <= colour__last .
H14:   j - 1 >= indexrange__first .
H15:   j - 1 <= indexrange__last .
H16:   i >= indexrange__first .
H17:   i <= indexrange__last .
        ->
C1:    element(flag, [i]) >= colour__first .
C2:    element(flag, [i]) <= colour__last .
C3:    j - 1 >= indexrange__first .
C4:    j - 1 <= indexrange__last .


For path(s) from default assertion of line 15 to finish:

procedure_partition_section_12.
*** true .          /* trivially true VC removed by Examiner */
```

C.2.2 Simplified

```
          *******************************************************
                     Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01
                          Demonstration Version
          *******************************************************


      CREATED 03-APR-2002, 10:11:24  SIMPLIFIED 03-APR-2002, 10:22:19
               (Simplified by SPADE Simplifier, Version 1.45)

                   procedure PolishFlag.Partition_Section




For path(s) from start to run-time check associated with statement of line 11:

procedure_partition_section_1.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 12:

procedure_partition_section_2.
*** true .          /* all conclusions proved */


For path(s) from start to default assertion of line 15:

procedure_partition_section_3.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 15 to default assertion of line 15:

procedure_partition_section_4.
*** true .          /* all conclusions proved */


procedure_partition_section_5.
H1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H2:    i <> j .
H3:    i >= 1 .
H4:    i <= 4 .
H5:    element(flag,[i]) <> red .
H6:    j >= 2 .
H7:    red <= element(flag,[i]) .
```

```
H8:    element(flag,[i]) <= white .
H9:    red <= element(flag,[j - 1]) .
H10:   element(flag,[j - 1]) <= white .
H11:   j <= 5 .
       ->
C1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(
          update(update(flag,[i],element(flag,[j - 1])),[j - 1],element(flag,
          [i])),[i___1]) and element(update(update(flag,[i],element(flag,[j -
          1])),[j - 1],element(flag,[i])),[i___1]) <= white) .


For path(s) from default assertion of line 15 to run-time check associated with
          statement of line 22:

procedure_partition_section_6.
H1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H2:    i <> j .
       ->
C1:    i >= 1 .
C2:    i <= 4 .


For path(s) from default assertion of line 15 to run-time check associated with
          statement of line 23:

procedure_partition_section_7.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 15 to run-time check associated with
          statement of line 25:

procedure_partition_section_8.
H1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H2:    i <> j .
H3:    i >= 1 .
H4:    i <= 4 .
H5:    element(flag,[i]) <> red .
       ->
C1:    j >= 2 .
C2:    j <= 6 .


For path(s) from default assertion of line 15 to run-time check associated with
          statement of line 26:

procedure_partition_section_9.
H1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H2:    i <> j .
H3:    i >= 1 .
H4:    i <= 4 .
H5:    element(flag,[i]) <> red .
H6:    j >= 2 .
H7:    j <= 6 .
       ->
C1:    red <= element(flag,[i]) .
C2:    element(flag,[i]) <= white .


For path(s) from default assertion of line 15 to run-time check associated with
          statement of line 27:

procedure_partition_section_10.
H1:    for_all(i___1 : integer,i___1 >= 1 and i___1 <= 4 -> red <= element(flag,
          [i___1]) and element(flag,[i___1]) <= white) .
H2:    i <> j .
H3:    i >= 1 .
H4:    i <= 4 .
H5:    element(flag,[i]) <> red .
H6:    j >= 2 .
H7:    j <= 6 .
```

```
H8:    red <= element(flag,[i]) .
H9:    element(flag,[i]) <= white .
       ->
C1:    red <= element(flag,[j - 1]) .
C2:    element(flag,[j - 1]) <= white .
C3:    j <= 5 .


For path(s) from default assertion of line 15 to run-time check associated with
          statement of line 28:

procedure_partition_section_11.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 15 to finish:

procedure_partition_section_12.
*** true .          /* all conclusions proved */
```