CS-Note* 31

Bill J. Ellis
September 13, 2002

Learning by Example - Exception Freedom in Polish Flag

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

Heuristics are discovered by looking for patterns in a collection of examples. To find heuristics that help automate SPARK exception freedom proofs, a number of SPARK examples will need to be considered.

These examples will be documented in full detail, mistakes and all, trying to identify the reasoning behind each step.

1.1 Finding Heuristics From Examples

Heuristics will be identified using failure analysis. At the point of failure questions are asked of why the proof failed, such as:

- What information is missing?
- How would missing information be discovered from the failure state?
- If this information was provided, what else might go wrong?

The technique is successful as the failed proof provides a starting point from which to work from.

1.2 Polish Flag

The polish flag problem has been considered in some detail already. Proof of partial correctness is outlined in CS-note 10. It is also used as a running example in CS-note 12 to illustrate the nature of exception freedom proofs.

The starting point for this note is the polish flag with only core annotations. See Appendix A for full details on this example. It is expected that this is the normal starting point for the majority of SPARK code subjected to exception freedom proof.

2 Thinking outwards from VC 6

2.1 Failure analysis 1

2.1.1 Situation

#	From	To	Proved In vcg	Proved In siv	Proved In plg	TO DO
1	start	rtc check @ 9		YES		
2	start	rtc check @ 10		YES		
3	start	assert @ 12		YES		
4	12	assert @ 12		YES		
5	12	assert @ 12		YES		
6	12	rtc check @ 13				YES
7	12	rtc check @ 14		YES		
8	12	rtc check @ 16				YES
9	12	rtc check @ 17		YES		
10	12	rtc check @ 18				YES
11	12	rtc check @ 19		YES		
12	12	assert @ finish	YES			

2.1.2 Looking at VC 6

Location information

```
For path(s) from default assertion of line 12 to run-time check associated with
statement of line 13:

procedure_partition_section_6.
```

Code snippet

```
11        loop
12            exit when I=J;
13            if Flag(I)=Red then
14                I:=I+1;
```

Before simplification

```
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 .
```

After simplification

```
H1:    for_all(i___1 : integer, 1 <= i___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 .
```

• **VC shows -** Proving this VC shows that in the expression Flag(I)=Red[1] the variable I is within the valid range for array Flag.

• **About unsimp-H1[2] -** Hypothesis `unsimp-H1` has been carried from the EXAMINER generated precondition. On entering this procedure the imported values must all meet their type. Variable *flag* is an array as specified in the code as ArrayOfColours. Tracing the definition of this through the code gives the actual type of *flag*: details of this can be found:

Tracing definition

```
type ArrayOfColours is array (IndexRange) of Colour;
subtype IndexRange is Integer range IndexLower .. IndexUpper;
IndexUpper: constant := 4;
IndexLower: constant := 1;
type Colour is (Red, White);

type ArrayOfColours is array (1 .. 4) of (Red, White);
```

This matches the information available in `unsimp-H1`, albeit in a different notation.

• **Missing information -** There is no information about the variable *i* or *j* other than that they are not equal. As *i* is directly present in the conclusions it is viewed as more significant missing information.

• **Looking at the code for `i` -** Variable *i* is of type JustBiggerRange. Prior to this section of code the assertion that *I* is in this range could be added:

```
11        loop
12            exit when I=J;
13            --# assert I in JustBiggerRange;
14            if Flag(I)=Red then
15                I:=I+1;
```

[1] There are variables in the code and in the VCs, often with similar names. To help avoid confusion all code segments will be in sans serif font (This is code in sans serif font) and VC segments in slanted font (*This is part of a VC in slanted font*).

[2] The simplified VCs (simp) and unsimplified VCs (unsimp) often have different numbering for the same hypotheses or conclusions. A prefix is added to distinguish between the two.

2.2 Failure analysis 2

2.2.1 Situation

#	From	To	Proved In: vcg	Proved In: siv	Proved In: plg	TO DO
1	start	rtc check @ 9		YES		
2	start	rtc check @ 10		YES		
3	start	assert @ 13		YES		
4	13	assert @ 13		YES		
5	13	assert @ 13		YES		
6	13	rtc check @ 14				YES
7	13	rtc check @ 15		YES		
8	13	rtc check @ 17				YES
9	13	rtc check @ 18		YES		
10	13	rtc check @ 19				YES
11	13	rtc check @ 20		YES		
12	start	assert @ finish	YES			
13	13	assert @ finish	YES			
14	13	assert @ finish	YES			

2.2.2 Looking at VC 6

Location information

```
For path(s) from assertion of line 13 to run-time check associated with
statement of line 14:

procedure_partition_section_6.
```

Code snippet

```
12            exit when I=J;
13            --# assert I in JustBiggerRange;
14            if Flag(I)=Red then
15               I:=I+1;
```

Before simplification

```
H1:    i >= justbiggerrange__first .
H2:    i <= justbiggerrange__last .
H3:     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:    i >= indexrange__first .
C2:    i <= indexrange__last .
```

After simplification

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

• **Extra information -** Extra information is now available as `unsimp-H1` and `unsimp-H2`. This describes the known range of *i* just prior to reaching expression being considered. Adding this information has led to additional proof effort required in earlier VCs. However, the SIMPLIFIER has been successful in proving these.

• **Assertion is not invariant -** The additional assertion was added just prior to the expression under consideration. This is not the correct place for an invariant. Thus the extra assertion provides an additional cut in the code, resulting in an extra two VCs. Both of these are discharged by the SIMPLIFIER. Extra information is likely to be better placed inside the most recent proof statement, either the precondition or last assertion[3].

• **Lost information -** The information that `i <> j` has been lost! This is a result of the extra cut not carrying through this know information from the failure of the loop exit condition. Rather than moving the assertion to the invariant position, a check statement shall be considered first.

```
12            exit when I=J;
13            --# check I in JustBiggerRange;
14            if Flag(I)=Red then
15               I:=I+1;
```

2.3 Failure analysis 3

2.3.1 Situation

#	From	To	Proved In: vcg	Proved In: siv	Proved In: plg	TO DO
1	start	rtc check @ 9		YES		
2	start	rtc check @ 10		YES		
3	start	assert @ 12		YES		
4	12	assert @ 12		YES		
5	12	assert @ 12		YES		
6	12	check stm @ 13				YES
7	12	rtc check @ 14				YES
8	12	rtc check @ 15		YES		
9	12	rtc check @ 17				YES
10	12	rtc check @ 18		YES		
11	12	rtc check @ 19				YES
12	12	rtc check @ 20		YES		
13	12	assert @ finish	YES			

2.3.2 Looking at VC 6

Location information

```
For path(s) from default assertion of line 12 to check associated with
statement of line 13:

procedure_partition_section_6.
```

Code snippet

```
11         loop
12            exit when I=J;
13            --# check I in JustBiggerRange;
14            if Flag(I)=Red then
```

Before simplification

```
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 >= justbiggerrange__first .
C2:    i <= justbiggerrange__last .
```

After simplification

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

• **Not proving check statement -** VC 6 is now a proof of the added check statement. The previous VC 6 is now VC 7. VC 6 is very similar to VC 7, with one important difference. The range check is within all of JustBiggerRange rather than one smaller from the end of IndexRange. VC 6 can be proved with the extra information proposed for helping prove VC 7. It is apparent that this information must come from outside the loop. Adding a new check statement would prove VC 6, but introduce a new unproved VC exactly the same as VC 6. This gives evidence that extra information required should be added inside the usual proof annotations (precondition, invariant, postcondition). The following extends the loops invariant to provide the extra information:

[3] A postcondition might be modified in the process of proving code that calls it. An example involving procedure calls should help explore this.

```
11          loop
12              --# assert I in JustBiggerRange;
13              exit when I=J;
14              if Flag(I)=Red then
```

• **Look at VC 7 -** Information is assumed in VC 7 that has not been proved. However, it is expected that this proof will be completed with the modified invariant. That VC 7 is not automatically solved is a new problem.

2.3.3 Looking at VC 7

Location information

```
For path(s) from default assertion of line 12 to run-time check associated with
statement of line 14:

procedure_partition_section_7.
```

Code snippet

```
11          loop
12              exit when I=J;
13              --# check I in JustBiggerRange;
14              if Flag(I)=Red then
15                  I:=I+1;
```

Before simplification

```
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 >= justbiggerrange__first .
H4:    i <= justbiggerrange__last .
        ->
C1:    i >= indexrange__first .
C2:    i <= indexrange__last .
```

After simplification

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

• **Type information proved a conclusion -** As *indexrange_first* and *justbiggerrange_first* share the same starting value (the constant 1) `unsimp-H3` is the same as `unsimp-C1`. Thus `unsimp-C1` can be trivially proved using axiom $X \vdash X$.

• **Re-addressing** `j` **-** Conclusion `simp-C1` (*i <= 4*) remains. Everything know about *i* is *i >=1*, *i <= 5* and *i <> j*. As nothing is know about *j*, this last hypothesis does not give any information. This knowledge might be extracted by feeding what is known into a constraint solver:

```
| ?- (I#>=1) #/\ (I #=<5) #/\ (I #\= J).

I in 1..5,
J in inf..sup ?

yes
```

That is, *i* is in the range 1..5 and *j* is in the range minus infinity (inf) to positive infinity (sup). In other words there is no information on *j*.

Recall that type information was not added on *j* as it was not in a conclusion. This decision may be flawed. It is difficult to envisage a scenario where extra type information was detrimental to completing a proof. Perhaps the constraint solver may be used at an early stage to identify all variables that have no type information, and subsequently add this information. This is the stage at now:

```
--# assert I in JustBiggerRange and
--#        J in JustBiggerRange;
```

Note that the constraint solver only applies to inequalities. The variable *flag* does not require extra type information as it is available in `unsimp-H1`. Identifying that this is the case may be difficult. An alternative would be to provide type information for all variables, regardless. Perhaps exploiting known behaviour of the the EXAMINER would allow many common duplications to be avoided.

• **Maybe this is provable -** It might be the case that no additional information is required. The proof is not completed due to the limitations of the SIMPLIFIER or NUSPADE. It might be beneficial to demonstrate that VC 7 is currently a non-theorem. In some trivial cases this might be provided through a constraint solver. In VC 7 this would involve specifying the constraints and adding a request for entailment[4] that *i <= 4*.

2.4 Failure analysis 4

2.4.1 Situation

#	From	To	Proved In: vcg	siv	plg	TO DO
1	start	rtc check @ 10		YES		
2	start	rtc check @ 11		YES		
3	start	assert @ 13		YES		
4	13	assert @ 13		YES		
5	13	assert @ 13		YES		
6	13	rtc check @ 15				YES
7	13	rtc check @ 16		YES		
8	13	rtc check @ 18				YES
9	13	rtc check @ 19		YES		
10	13	rtc check @ 20		YES		
11	13	rtc check @ 21		YES		
12	13	assert @ finish	YES			

2.4.2 Looking at VC 6

Location information

```
For path(s) from assertion of line 13 to run-time check associated with statement of line 15:

procedure_partition_section_6.
```

Code snippet

```
12          loop
13              --# assert I in JustBiggerRange and J in JustBiggerRange;
14              exit when I=J;
15              if Flag(I)=Red then
16                  I:=I+1;
```

Before simplification

```
H1:    i >= justbiggerrange__first .
H2:    i <= justbiggerrange__last .
H3:    j >= justbiggerrange__first .
H4:    j <= justbiggerrange__last .
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 .
```

[4] The method for doing this is currently not known. A deeper analysis of constraint solvers might be worthwhile.

After simplification

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

• **Proved VC 10 -** It is noted that VC 10 has now been solved. Adding the type information for *`i`* and *`j`* gives the simplifier enough information to prove this. VC 10 will be considered in more detail later.

• **Not enough information -** There is still an unprovable conclusion `simp-C1`. Returning to the constraint solver, adding the additional information gives:

```
| ?-  (I#>=1) #/\ (I #=<5) #/\
      (J#>=1) #/\ (J #=<5) #/\
      (I #\= J).

I in 1..5,
J in 1..5 ?

yes
```

This is not a sufficient constraint to show that *`i <= 4`*.

• **Classifying information -** All those hypotheses with constant inequalities are providing type information. The unsimplified VCs make this clearer showing the constant names, for example *`justbiggerrange_first`*. Removing all of these type specifiers leaves `simp-H5` and `simp-H6`. It is known that `simp-H5` is information about *`flag`* which is not of interest here. This leaves `simp-H6`, *`i <> j`*. Assuming it is known this is inside a loop (a likely piece of information) this must be negation of the exit condition. There may have been other entries like `simp-H6`. For example, if the loop was inside a conditional statement, or the user has added some extra assertions. Identifying for certain that this is the loop guard could be shown by returning to the code.

• **Loop heuristics -** It is known that the loop will terminate when *`not(i <> j)`*. As this is currently false, it must be the case that the variables *`i`* and *`j`*, *`i`* or *`j`* change within the loop[5]. Three situations are identified:

1. *`(i < j)`*. *`i`* is less than *`j`*, with both eventually moving toward each other.

2. *`(i > j)`*. *`i`* is greater than *`j`*, with both eventually moving toward each other.

3. *`(i < j) or (i > j)`*. It is possible for *`i`* to be less than *`j`* and greater than *`j`* in the process of the algorithm.

It is conjectured that the latter of these is uncommon in most code. It is more algorithmic to gradually and incrementally reach the exit condition, rather than a more disjoint approach. It seems that both *`i < j`* and *`i > j`* are good candidates for enhancing the invariant.

• SPARK **invariants -** In some SPARK code the invariant must hold where the exit condition is true. There is an alternative way of annotating SPARK loops that overcomes this, however, it is at the expense of additional problems[6]. Knowing which kind of invariant is required is additional information that may be gained directly from bottom-up analysis or indirectly through patching a failed proof.

• **Invariant must hold at exit -** It is assumed that bottom-up analysis has revealed that the invariant for this examples must hold when the loop exits. based on this information additional checks can be made. The loop exit condition I=J must not contradict anything in the invariant. This would contradict with both *`i < j`* and *`i > j`*. The contradiction can be shown using the constraint solver:

```
| ?- I in 1..5, J in 1..5, (I#<J) #/\
     (I#=J).

no

| ?- I in 1..5, J in 1..5, (I#>J) #/\
     (I#=J).

no
```

There is no values that will satisfy the invariant, type information about *`i`* and *`j`* and either the added *`i<j`* or *`i>j`*, and the exit condition *`i=j`*.

Alternatively the same information may be gathered more generally by conducting a separate proof plan. This would aim to prove that the invariant conjoined with the exit condition is false:

```
H1:    i >= 1 and i <= 5 .
H2:    j >= 1 and j <= 5 .
H3:    i = j .
H4:    i < j .
       ->
C1:    false .

H1:    i >= 1 and i <= 5 .
H2:    j >= 1 and j <= 5 .
H3:    i = j .
H4:    i > j .
       ->
C1:    false .
```

[5]Or the loop does not terminate, and the variables never change.

[6]Using the `while` command with an assertion in the `while` body will give a regular invariant. However, now the assertion need not be reached if the `while` condition is initially false. This results in additional VCs for paths around the while loop.

• **More principled -** Patching failed proofs is applicable to all VCs. Use of a constraint solver or doing a separate plan above is guided by additional information (invariant covers exit condition) that might not always be available. Essentially this extra proof is replacing the use of a failed proof attempt for another, smaller and more focused, failed proof attempt. Instead the choice of `i<j` or `i>j` could be made and the resulting failing proof plan can be patched.

• **Using patching -** It is known that `i<j` is a good choice. However, that `i>j` is just as valid. Eliminating the latter is necessary. The following invariant is used, in the hope that failure identifies that `i<j` is a better choice or merely that `i>j` was a bad choice, and that the alternative `i<j` should be tried:

```
--# assert I in JustBiggerRange and
--#        J in JustBiggerRange and
--#        I>J;
```

2.5 Failure analysis 5

2.5.1 Situation

#	From	To	Proved In: vcg	siv	plg	TO DO
1	start	rtc check @ 10		YES		
2	start	rtc check @ 11		YES		
3	start	assert @ 13				YES
4	13	assert @ 13		YES		
5	13	assert @ 13		YES		
6	13	rtc check @ 16				YES
7	13	rtc check @ 17		YES		
8	13	rtc check @ 19				YES
9	13	rtc check @ 20		YES		
10	13	rtc check @ 21		YES		
11	13	rtc check @ 22		YES		
12	13	assert @ finish	YES			

2.5.2 Looking at VC 3

Location information

For path(s) from start to assertion of line 13:

procedure_partition_section_3.

Code snippet

```
9     begin
10        I:=Flag'First;
11        J:=Flag'Last+1;
12        loop
13            --# assert I in JustBiggerRange and J in JustBiggerRange and
14            --# I>J;
15            exit when I=J;
```

Before simplification

```
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:    indexrange__first >= justbiggerrange__first .
C2:    indexrange__first <= justbiggerrange__last .
C3:    indexrange__last + 1 >= justbiggerrange__first .
C4:    indexrange__last + 1 <= justbiggerrange__last .
C5:    indexrange__first > indexrange__last + 1 .
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))) .
```

After simplification

```
H1:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 4 -> red <= element(
          flag, [i___1]) and element(flag, [i___1]) <= white) .
       ->
C1:    false .
```

• **Is a non-theorem -** Prior to entering the loop an ordering of `i` and `j` is established. This is in contradiction with the I ¿ J enhanced invariant. This is shown directly by the SIMPLIFIER. This is sufficient evidence to take the second choice:

```
--# assert I in JustBiggerRange and
--#        J in JustBiggerRange and
--#        I<J;
```

2.6 Failure analysis 6

2.6.1 Situation

#	From	To	Proved In: vcg	siv	plg	TO DO
1	start	rtc check @ 10		YES		
2	start	rtc check @ 11		YES		
3	start	assert @ 13		YES		
4	13	assert @ 13				YES
5	13	assert @ 13				YES
6	13	rtc check @ 16		YES		
7	13	rtc check @ 17		YES		
8	13	rtc check @ 19		YES		
9	13	rtc check @ 20		YES		
10	13	rtc check @ 21		YES		
11	13	rtc check @ 22		YES		
12	13	assert @ finish	YES			

2.6.2 Looking at VC 4

Location information

For path(s) from assertion of line 13 to assertion of line 13:

procedure_partition_section_4.

Code snippet

```
12        loop
13            --# assert I in JustBiggerRange and J in JustBiggerRange and
14            --# I<J;
15            exit when I=J;
16            if Flag(I)=Red then
17                I:=I+1;
18            else
19                J:=J-1;
20                T:=Flag(I);
21                Flag(I):=Flag(J);
22                Flag(J):=T;
23            end if;
24        end loop;
```

Before simplification

```
H1:    i >= justbiggerrange__first .
H2:    i <= justbiggerrange__last .
H3:    j >= justbiggerrange__first .
H4:    j <= justbiggerrange__last .
H5:    i < j .
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:    not (i = j) .
H8:    i >= indexrange__first .
H9:    i <= indexrange__last .
H10:   element(flag, [i]) = red .
H11:   i + 1 >= justbiggerrange__first .
H12:   i + 1 <= justbiggerrange__last .
        ->
C1:    i + 1 >= justbiggerrange__first .
C2:    i + 1 <= justbiggerrange__last .
C3:    j >= justbiggerrange__first .
C4:    j <= justbiggerrange__last .
C5:    i + 1 < j .
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))) .
```

After simplification

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

• **Proved VC 8, unproved VC 4 -** The added invariant has resulted in VC 8 being automatically proved, and VC 4 no longer being automatically proved.

• **Rippling -** Rippling identifies the syntactic difference between the induction hypothesis `simp-H3` and the induction conclusion `simp-C1`:

$$i < j \to \boxed{\underline{i}+1} < j \tag{1}$$

This is a blocked and a non-theorem. Failure to prove this might prompt weakening the invariant inequalities to be non strict[7]:

$$i <= j \to \boxed{\underline{i}+1} <= j \tag{2}$$

Assuming rules for integers *X* and *Y*:

$$(X + 1 <= Y) = (X < Y) \tag{3}$$
$$(X <> Y \wedge X <= Y) = (X < Y) \tag{4}$$

Gives rewrite rules:

$$(X + 1 <= Y) \Rightarrow (X < Y) \tag{5}$$
$$(X <> Y \wedge X <= Y) \Rightarrow (X < Y) \tag{6}$$

Applying 5 gives:

$$i <= j \to i < j \tag{7}$$

Using forward reasoning, the hypothesis *`not (i = j)`* can be used can be used with 6 to reduce the induction hypothesis and complete the proof.

$$i < j \to i < j \tag{8}$$

The suggested weakening of the invariant seems to complete the proof.

```
--# assert I in JustBiggerRange and
--#        J in JustBiggerRange and
--#        I<=J;
```

• SIMPLIFIER **first -** Rather than try to complete the refined proof within the planner, the problem might be immediately dispatched to the SIMPLIFIER. In general there may be many different cases to explore, such that this would be too inefficient. However, the possibility remains.

2.7 Failure analysis 7

2.7.1 Situation

#	From	To	Proved In: vcg	siv	plg	TO DO
1	start	rtc check @ 10		YES		
2	start	rtc check @ 11		YES		
3	start	assert @ 13		YES		
4	13	assert @ 13		YES		
5	13	assert @ 13		YES		
6	13	rtc check @ 16		YES		
7	13	rtc check @ 17		YES		
8	13	rtc check @ 19		YES		
9	13	rtc check @ 20		YES		
10	13	rtc check @ 21		YES		
11	13	rtc check @ 22		YES		
12	13	assert @ finish	YES			

• **Everything is proved -** The proof is complete. The remaining VC 4 was proved automatically as a sound invariant is returned.

3 Thinking outwards from VC 8

3.1 Failure analysis 1

3.1.1 Situation

#	From	To	Proved In: vcg	siv	plg	TO DO
1	start	rtc check @ 9		YES		
2	start	rtc check @ 10		YES		
3	start	assert @ 12		YES		
4	12	assert @ 12		YES		
5	12	assert @ 12		YES		
6	12	rtc check @ 13				YES
7	12	rtc check @ 14		YES		
8	12	rtc check @ 16				YES
9	12	rtc check @ 17		YES		
10	12	rtc check @ 18				YES
11	12	rtc check @ 19		YES		
12	12	assert @ finish	YES			

3.1.2 Looking at VC 8

Location information

```
For path(s) from default assertion of line 12 to run-time check associated with statement
of line 16:

procedure_partition_section_8.
```

Code snippet

```
11        loop
12            exit when I=J;
13            if Flag(I)=Red then
14                I:=I+1;
15            else
16                J:=J-1;
17                T:=Flag(I);
```

Before simplification

```
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 .
```

After simplification

```
H1:    for_all(i___1 : integer, 1 <= i___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 .
```

• **VC shows -** Proving this VC shows that in the expression J:=J-1 the value assigned to variable J is within the valid range of the values for J.

• **Missing information -** There is no information about *j*. This should be added by extending the loop invariant. The information available about *i* is coming from the assumed proof of VC 6. Assuming that all variables not expected to be described by the EXAMINER have their type elaborated, the new invariant is:

```
--# assert I in JustBiggerRange and
--#        J in JustBiggerRange;
```

[7]This is particularly useful in SPARK VCs, where the invariant may overlap with an exit condition.

3.2 Failure analysis 2

3.2.1 Situation

```
-------------------------------------------------------------------
     |       |                     |  -----Proved In-----  |       |
#    | From  | To                  |  vcg  |  siv  |  plg  | TO DO |
-------------------------------------------------------------------
1    | start | rtc check @ 9       |       |  YES  |       |       |
2    | start | rtc check @ 10      |       |  YES  |       |       |
3    | start |    assert @ 12      |       |  YES  |       |       |
4    | 12    |    assert @ 12      |       |  YES  |       |       |
5    | 12    |    assert @ 12      |       |  YES  |       |       |
6    | 12    | rtc check @ 14      |       |       |       |  YES  |
7    | 12    | rtc check @ 15      |       |  YES  |       |       |
8    | 12    | rtc check @ 17      |       |       |       |  YES  |
9    | 12    | rtc check @ 18      |       |  YES  |       |       |
10   | 12    | rtc check @ 19      |       |  YES  |       |       |
11   | 12    | rtc check @ 20      |       |  YES  |       |       |
12   | 12    |    assert @ finish  |  YES  |       |       |       |
```

3.2.2 Looking at VC 8

Location information

```
For path(s) from assertion of line 12 to run-time check associated
with statement of line 17:

procedure_partition_section_8.
```

Code snippet

```
11        loop
12            --# assert I in JustBiggerRange and J in JustBiggerRange;
13            exit when I=J;
14            if Flag(I)=Red then
15                I:=I+1;
16            else
17                J:=J-1;
18                T:=Flag(I);
```

Before simplification

```
H1:    i >= justbiggerrange__first .
H2:    i <= justbiggerrange__last .
H3:    j >= justbiggerrange__first .
H4:    j <= justbiggerrange__last .
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) .
H7:    i >= indexrange__first .
H8:    i <= indexrange__last .
H9:    not (element(flag, [i]) = red) .
        ->
C1:    j - 1 >= justbiggerrange__first .
C2:    j - 1 <= justbiggerrange__last .
```

After simplification

```
H1:    i >= 1 .
H2:    j >= 1 .
H3:    j <= 5 .
H4:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 4 -> red <= element(
          flag, [i___1]) and element(flag, [i___1]) <= white) .
H5:    i <> j .
H6:    i <= 4 .
H7:    element(flag, [i]) <> red .
       ->
C1:    j >= 2 .
```

• **Proved conclusion -** The type information has allowed the SIMPLIFIER to prove `unsimp-C1`. This is through `unsimp-H4`, *`j <= justbiggerrange_last`*, subsuming `unsimp-C2`, *`j - 1 <= justbiggerrange_last`*.

• **Unproved conclusion remains -** The unproved conclusion `simp-C1` puts the problem at essentially the same stage seen in §2.4, only from a *j* rather than *i* perspective. The previous analysis covers this problem.

4 Thinking outwards from VC 10

4.1 Failure analysis 1

4.1.1 Situation

```
-------------------------------------------------------------------
     |       |                     |  -----Proved In-----  |       |
#    | From  | To                  |  vcg  |  siv  |  plg  | TO DO |
-------------------------------------------------------------------
1    | start | rtc check @ 9       |       |  YES  |       |       |
2    | start | rtc check @ 10      |       |  YES  |       |       |
3    | start |    assert @ 12      |       |  YES  |       |       |
4    | 12    |    assert @ 12      |       |  YES  |       |       |
5    | 12    |    assert @ 12      |       |  YES  |       |       |
6    | 12    | rtc check @ 13      |       |       |       |  YES  |
7    | 12    | rtc check @ 14      |       |  YES  |       |       |
8    | 12    | rtc check @ 16      |       |       |       |  YES  |
9    | 12    | rtc check @ 17      |       |  YES  |       |       |
10   | 12    | rtc check @ 18      |       |       |       |  YES  |
11   | 12    | rtc check @ 19      |       |  YES  |       |       |
12   | 12    |    assert @ finish  |  YES  |       |       |       |
```

4.1.2 Looking at VC 10

Location information

```
For path(s) from default assertion of line 12 to run-time check associated with statement
of line 18:

procedure_partition_section_10.
```

Code snippet

```
11        loop
12            exit when I=J;
13            if Flag(I)=Red then
14                I:=I+1;
15            else
16                J:=J-1;
17                T:=Flag(I);
18                Flag(I):=Flag(J);
19                Flag(J):=T;
```

Before simplification

```
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 .
```

After simplification

```
H1:    for_all(i___1 : integer, 1 <= i___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:    red < element(flag, [i]) .
H6:    j >= 2 .
H7:    j <= 6 .
H8:    element(flag, [i]) <= white .
       ->
C1:    red <= element(flag, [j - 1]) .
C2:    element(flag, [j - 1]) <= white .
C3:    j <= 5 .
```

• **VC shows -** Proving this VC shows that in the expression Flag(I):=Flag(J) the variables I and J are within the valid range for array Flag and that the value assigned to Flag(I) is within the valid values for array Flag.

• **Missing information -** There is information about both *j* and *i*, carried from previous assumed proofs. However, the information on *j* is weaker than the type information as previous proofs only showed that J-1 is in the range of J. The type information is added:

```
--# assert I in JustBiggerRange and
--#        J in JustBiggerRange;
```

4.2 Failure analysis 2

4.2.1 Situation

```
-------------------------------------------------------------------
     |       |                     |  -----Proved In-----  |       |
#    | From  | To                  |  vcg  |  siv  |  plg  | TO DO |
```

```
-------------------------------------------------------------------
1    | start | rtc check @ 9      |      | YES  |      |      |
2    | start | rtc check @ 10     |      | YES  |      |      |
3    | start |    assert @ 12     |      | YES  |      |      |
4    | 12    |    assert @ 12     |      | YES  |      |      |
5    | 12    |    assert @ 12     |      | YES  |      |      |
6    | 12    | rtc check @ 14     |      |      |      | YES  |
7    | 12    | rtc check @ 15     |      | YES  |      |      |
8    | 12    | rtc check @ 17     |      |      |      | YES  |
9    | 12    | rtc check @ 18     |      | YES  |      |      |
10   | 12    | rtc check @ 19     |      | YES  |      |      |
11   | 12    | rtc check @ 20     |      | YES  |      |      |
12   | 12    |    assert @ finish | YES  |      |      |      |
```

• **Proved conclusion -** As noted in VC 6, adding type information alone is sufficient to discharge VC 10.

5 Conclusions

The following observations are made:

• **Type information is beneficial -** Adding type information alone proves a number of conclusions. It may be feasible to incrementally add type information such that is just sufficient to complete a proof. However, given the advantages of type information there seems to be no real loss in adding all type in one go.

• **External tools useful -** The process of patching failed proof attempts seems a more powerful mechanism than employing various helper tools in special situations. However, there is certainly scope for using such special tools to aid in the analysis of failed proof attempts.

• **Little direct reliance on bottom-up techniques -** Information available through bottom-up techniques is often hinted at through using top-down techniques. For example, patching a failed proof attempt hinted that an invariant must also be true at exit, revealing the nature of the SPARK loop and annotations used in the code.

• **Bottom-up techniques potentially very useful -** A great deal of the information gain through patching failed proofs might be directly available in the source. This information might include the kind of invariant being used, extensive type information and the initial state of values prior to entering a loop.

• **Rippling less valuable -** Rippling is designed for mathematical inductive proofs. It exploits syntactic similarities between the induction hypothesis and induction conclusion to guide a proof. Partial correctness VCs between invariants are essentially inductive proofs, and rippling applies. The same is technically true for exception freedom proofs. However, it seems that the more light weight level of reasoning required reduces the applicability of rippling. Here it is often the case that the reason for failure is that the problem is a non-theorem, rather than some missing eureka step. Further, it seems likely that a combination of forward and backwards reasoning is required to complete a proof.

A Polish Flag Example

A.1 Extra Information

A.1.1 RLS File

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


                      /*DATE : 12-JUL-2002 09:21:31.40*/

                  /*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.1.2 FDL File

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


                      {DATE : 12-JUL-2002 09:21:31.40}

                  {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;
```

A.2 Code

A.2.1 Specification

```
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;
```

A.2.2 Body

```
1       package body PolishFlag is
2           procedure Partition_Section(Flag: in out ArrayOfColours)
3           is
4               subtype JustBiggerRange is Integer range Flag'First
.. Flag'Last+1;
5               I: JustBiggerRange;
6               J: JustBiggerRange;
7               T: Colour;
8           begin
9               I:=Flag'First;
10               J:=Flag'Last+1;
11               loop
12                   exit when I=J;
13                   if Flag(I)=Red then
14                       I:=I+1;
15                   else
16                       J:=J-1;
17                       T:=Flag(I);
18                       Flag(I):=Flag(J);
19                       Flag(J):=T;
20                   end if;
21               end loop;
22           end Partition_Section;
23        end PolishFlag;
```

A.3 Exception Freedom VCs

A.3.1 Unsimplified

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


                      DATE : 12-JUL-2002 09:21:31.40

                  procedure PolishFlag.Partition_Section




For path(s) from start to run-time check associated with statement of line 9:

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 10:

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 12:

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 12 to default assertion of line 12:

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 12 to run-time check associated
with statement of line 13:

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 12 to run-time check associated
with statement of line 14:

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 12 to run-time check associated
with statement of line 16:

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 12 to run-time check associated
with statement of line 17:

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 12 to run-time check associated
with statement of line 18:

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 12 to run-time check associated
with statement of line 19:

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 12 to finish:

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

A.3.2 Simplified

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


     CREATED 12-JUL-2002, 09:21:31  SIMPLIFIED 12-JUL-2002, 09:21:38
               (Simplified by SPADE Simplifier, Demo Version)

                     procedure PolishFlag.Partition_Section




For path(s) from start to run-time check associated with statement of line 9:

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


For path(s) from start to run-time check associated with statement of line 10:

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

```
For path(s) from start to default assertion of line 12:

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


For path(s) from default assertion of line 12 to default assertion of line 12:

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


procedure_partition_section_5.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 12 to run-time check associated with
          statement of line 13:

procedure_partition_section_6.
H1:    for_all(i___1 : integer, 1 <= i___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 12 to run-time check associated with
          statement of line 14:

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


For path(s) from default assertion of line 12 to run-time check associated with
          statement of line 16:

procedure_partition_section_8.
H1:    for_all(i___1 : integer, 1 <= i___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 12 to run-time check associated with
          statement of line 17:

procedure_partition_section_9.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 12 to run-time check associated with
          statement of line 18:

procedure_partition_section_10.
H1:    for_all(i___1 : integer, 1 <= i___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:    red < element(flag, [i]) .
H6:    j >= 2 .
H7:    j <= 6 .
H8:    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 12 to run-time check associated with
          statement of line 19:

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


For path(s) from default assertion of line 12 to finish:

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