Andrew Ireland
March 28, 2002

# The Polish Flag Problem

## 1 Introduction

The aim of this note is to begin to explore the limitations of our current proof plans for reasoning about programs written in SPARK. Two verification attempts for a simple partitioning program are presented. The program contains simple loop and conditional constructs. The first attempt involves a verification where the SPARK code is annotated with a suitable loop invariant. The second attempt investigates the potential for automatically generating the loop invariant based upon the analysis of a failed proof attempt. Both verification attempts raise issues that need to be addressed in terms of our proof plans as well as the NuSPADE architecture.

## 2 Problem Definition

I consider a SPARK solution to what is known as the *Polish Flag Problem*. The general idea is to partition a mixture of coloured objects into distinct colours. In the case of the Polish Flag Problem, there are two distinct colours, *i.e.* red and white, corresponding to the colours of the Polish National Flag[1]. A solution[2] to the problem based upon arrays is given in appendix A.

An array `Flag` is used to represent the mixture of colours. It is assumed that all the elements of `Flag` are either `Red` or `White`. This assumption is expressed by the following precondition:

```
--# pre (for all I in IndexRange => (Flag(I)=Red or Flag(I)=White));
```

where `IndexRange` defines the range of valid indices for the array `Flag`.

The required postcondition takes the following form:

```
--# 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)));
```

This asserts that on termination all the `Red` elements within `Flag` will proceed the `White` elements, where the existential variable `P` is used to indicate the lower bound of the `White` elements. Note that to accommodate the situation where `Flag` contains no `White` elements, the upper bound of `P` is defined to be `Flag'Last+1`.

The basic idea behind the algorithm is that a lower segment of `Red` elements and an upper segment of `White` elements are maintained during the computation. Two local variables, `I` and `J` are used in defining the upper and lower bounds of each segment respectively during this computation. Sandwiched between the lower and upper segments (`I..J-1` inclusive) is a mixture of coloured elements. This "basic idea" is expressed formally by the following loop invariant:

```
--# 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));
```

Note that on termination `I=J` and the consequently the mixed colours segment (`I..J-1`) will be empty.

## 3 Proof Planning: Verification

In terms of verification, the SPARK Examiner generates 4 (non-RTC) verification conditions (VCs) for the implementation of the Polish Flag Problem which are given in appendix B. Presented below are pretty printed versions of each VC together with a high-level proof plan.

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

[1]This is a simplification of Dijkstra's *Dutch National Flag Problem* which requires three colours.

[2]Thanks to Bill for the code and the associated analysis.

$$ele(upd(X,Y,Z),Y) \Rightarrow Z \tag{1}$$
$$(X-1)+1 \Rightarrow X \tag{2}$$

Figure 1: Simplification rules

$$\boxed{(X+1)}^{\uparrow} - 1 \Rightarrow \boxed{(X-1)+1}^{\uparrow} \tag{3}$$
$$(\forall X : integer.((X \geq L) \wedge (X \leq \boxed{U+1}^{\uparrow})) \rightarrow P(X)) \Rightarrow \boxed{(\forall X : integer.((X \geq L) \wedge (X \leq U)) \rightarrow P(X)) \wedge P(U+1)}^{\uparrow} \tag{4}$$
$$(\forall X : integer.((X \geq \boxed{L-1}^{\uparrow}) \wedge (X \leq U)) \rightarrow P(X)) \Rightarrow \boxed{(\forall X : integer.((X \geq L) \wedge (X \leq U)) \rightarrow P(X)) \wedge P(L-1)}^{\uparrow} \tag{5}$$
$$\neg(X = Z) \rightarrow ele(\boxed{upd(W,X,Y)}^{\uparrow}, Z) \Rightarrow ele(W,Z) \tag{6}$$

Figure 2: Wave rules

## 3.1 Pre-loop verification (VC1)

The pre-loop verification deals with correctness from the start of the code to the given loop invariant. The actual VC (no simplification) takes the form[3]:

$\ldots$

$\rightarrow$

C1: $indexrange\_first \leq indexrange\_first$
C2: $indexrange\_last + 1 \leq indexrange\_last + 1$
C3: $indexrange\_first \leq indexrange\_last + 1$
C4: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq indexrange\_first - 1)) \rightarrow ele(flag, [q]) = red)$
C5: $(\forall r : integer.((r \geq indexrange\_last + 1) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$

### 3.1.1 Proofs of C1-C3

C1 through to C3 are trivial. Note that in the case of C3 the application dependent proof rules (`rls` files) generated by the Examiner are required.

### 3.1.2 Proofs of C4 and C5

In the case of C4 the range $((q \geq indexrange\_first) \wedge (q \leq indexrange\_first - 1))$ is contradictory. Consequently, C4 is trivially true. A similar argument holds for C5.

[3] Note that I use *ele* and *upd* as abbreviations for *element* and *update* respectively.

## 3.2 Loop-then-branch verification (VC2)

H1: $indexrange\_first \leq i$
H2: $j \leq indexrange\_last + 1$
H3: $i \leq j$
H4: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i - 1)) \rightarrow ele(flag, [q]) = red)$
H5: $(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$
H6: $\neg(i = j)$
H7: $ele(flag, [i]) = red$
$\rightarrow$
C1: $indexrange\_first \leq i + 1$
C2: $j \leq indexrange\_last + 1$
C3: $i + 1 \leq j$
C4: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq \boxed{(\underline{i} + 1)}^{\uparrow} - 1)) \rightarrow ele(flag, [q]) = red)$
C5: $(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$

### 3.2.1 Proofs of C1-C3

C1 and C2 follow from hypotheses H1 and H2 respectively. From H3 and H6 we can deduce $i < j$ from which it is trivial to establish C3.

### 3.2.2 Proof of C4

Using wave-rule (3), C4 is rippled to give:

$$(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq \boxed{(\underline{i - 1}) + 1}^{\uparrow})) \rightarrow ele(flag, [q]) = red) \tag{7}$$

Further rippling of (7) requires range splitting, and in particular wave-rule schema (4). Note that the application of this wave-rule schema requires second-order matching and gives rise to:

$$\boxed{\underline{(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i - 1)) \rightarrow ele(flag, [q]) = red)} \wedge ele(flag, [(i - 1) + 1]) = red}^{\uparrow} \tag{8}$$

Rippling is complete and a match with H4 reduces (8) to:

$$ele(flag, [(i - 1) + 1]) = red \tag{9}$$

Note that (9) simplifies via (2) to give a match with H7.

### 3.2.3 Proof of C5

Finally, C5 matches H5 so the proof is immediate.

## 3.3 Loop-else-branch verification (VC3)

H1: $indexrange\_first \leq i$
H2: $j \leq indexrange\_last + 1$
H3: $i \leq j$
H4: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i - 1)) \rightarrow ele(flag, [q]) = red)$
H5: $(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$
H6: $\neg(i = j)$
H7: $\neg(ele(flag, [i]) = red)$
$\rightarrow$
C1: $indexrange\_first \leq i$
C2: $j - 1 \leq indexrange\_last + 1$
C3: $i \leq j - 1$
C4: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i - 1)) \rightarrow$
$ele(\boxed{upd(upd(\underline{flag}, [i], ele(flag, [j - 1])), [j - 1], ele(flag, [i]))}^{\uparrow}, [q]) = red)$
C5: $(\forall r : integer.((r \geq \boxed{\underline{j} - 1}^{\uparrow}) \wedge (r \leq indexrange\_last)) \rightarrow$
$ele(\boxed{upd(upd(\underline{flag}, [i], ele(flag, [j - 1])), [j - 1], ele(flag, [i]))}^{\uparrow}, [r]) = white)$

### 3.3.1 Proofs of C1-C3

C1 and C2 follow from hypotheses H1 and H2 respectively. Note that C3 follows from H3 and H6.

### 3.3.2 Proof of C4

In the case of C4, consider the composite updates (wave-fronts):

$$ele(\boxed{upd(\boxed{upd(\underline{flag},[i],ele(flag,[j-1]))}^{\uparrow},[j-1],ele(flag,[i]))}^{\uparrow},[q]) \tag{10}$$

These updates only effect elements $i$ and $j-1$ which are not within the range of $q$. Intuitively this means that the nested wave-fronts can be eliminated. Underlying this reduction is wave-rule (6). However, the current ripple method fails to apply (6) within this context. The problem is best described via the preconditions of the ripple method:

**Ripple method preconditions:**

1. *there exists a wave-front W within the goal formula*
2. *there exists a wave-rule R that matches W*
3. *any conditions C associated with R follow from the hypotheses*

Within the context of (10), two applications of wave-rule (6) are required. The first application generates a side condition of the form $\neg(j-1=q)$ while the second application generates a side condition of the form $\neg(i=q)$. Currently precondition 3 of the ripple method is not strong enough to establish either of these conditions. We need to strengthen the precondition in order to deal with bounded quantification. This extension will also require us to develop a stronger tactic (generic proof log). Basically this strengthening will focus on forward inference. For example, in the case of condition $\neg(j-1=q)$ the reasoning goes as follows:

> Given H1 and H4 we can derive $i<j$, from which $i \leq (j-1)$ follows. From the upper bound on $q$, *i.e.* $q \leq (i-1)$, we can derive $q<i$. Finally, by transitivity we get $q<j-1$, which implies $\neg((j-1)=q)$.

Note that a similar pattern of reasoning is required for $\neg(i=q)$.

Once the nested wave-fronts are eliminated then C4 becomes:

$$(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i-1)) \rightarrow ele(flag,[q]) = red) \tag{11}$$

Note that (11) matches H4.

### 3.3.3 Proof of C5

Finally, in the case of C5 a range split is suggested by the $\boxed{\underline{j}-1}^{\uparrow}$ wave-front, and wave-rule schema (5) is applicable. The application of (5) requires second order matching and reduces C5 to a formula of the form:

$$\boxed{(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(\boxed{\ldots \underline{flag} \ldots}^{\uparrow},[r]) = white)} \wedge$$
$$\boxed{ele(upd(upd(flag,[i],ele(flag,[j-1])),[j-1],ele(flag,[i])),[j-1]) = white}^{\uparrow} \tag{12}$$

Note that the left conjunct of (12) takes the form:

$$(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow$$
$$ele(\boxed{upd(upd(\underline{flag},[i],ele(flag,[j-1])),[j-1],ele(flag,[i]))}^{\uparrow},[r]) = white) \tag{13}$$

The composite update effects elements $i$ and $j-1$ which are outside the range of $r$. Consequently the nested wave-fronts within (13) can be eliminated via (6) as described in §3.3.2. The resulting goal formula takes the form:

$$\boxed{(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag,[r]) = white)} \wedge$$
$$\boxed{ele(upd(upd(flag,[i],ele(flag,[j-1])),[j-1],ele(flag,[i])),[j-1]) = white}^{\uparrow} \tag{14}$$

H5 now matches with (14) leaving a residue of the form:

$$ele(upd(upd(flag, [i], ele(flag, [j-1])), [j-1], ele(flag, [i])), [j-1]) = white \tag{15}$$

Note that simplification using (1) reduces (15) to:

$$ele(flag, [i]) = white \tag{16}$$

The final step involves deducing $ele(flag, [i]) = white$ from H7, *i.e.* $\neg(ele(flag, [i]) = red)$.

### 3.4 Post-loop verification (VC4)

H1: $indexrange\_first \leq i$
H2: $j \leq indexrange\_last + 1$
H3: $i \leq j$
H4: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i-1)) \rightarrow ele(flag, [q]) = red)$
H5: $(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$
H6: $i = j$
$\rightarrow$
C1: $(\exists p : integer.((p \geq indexrange\_first) \ \wedge (p \leq indexrange\_last + 1)) \wedge$
$(\forall q : integer.((q \geq indexrange\_first) \ \wedge (q \leq p-1)) \rightarrow ele(flag, [q]) = red) \wedge$
$(\forall r : integer.((r \geq p) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white))$

Instantiating the existential witness $p$ to be $i$ gives a new goal of the form:

$$\begin{aligned}(i \geq indexrange\_first) \wedge \\ (i \leq indexrange\_last + 1) \wedge \\ (\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq i-1)) \rightarrow ele(flag, [q]) = red) \wedge \\ (\forall r : integer.((r \geq i) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white))\end{aligned}$$

The first conjunct follows from H1 while the second conjunct follows from H6 and H7. Note that the third conjunct matches H4. Using H6 to rewrite the fourth conjunct we get a match with H5:

$$(\forall r : integer.((r \geq j) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$$

## 4 Proof Planning: Invariant Discovery

I now consider a proof planning attempt in the absence of a user supplied loop invariant. The aim is to use the constraints of the proof methods to guide the generation of an appropriate invariant automatically. My starting point is the postcondition given in §2, i.e.

$$\begin{aligned}&(\exists p : integer.((p \geq indexrange\_first) \wedge (p \leq indexrange\_last + 1)) \ \wedge \\ &\quad (\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq p-1)) \rightarrow ele(flag, [q]) = red) \ \wedge \\ &\quad (\forall r : integer.((r \geq p) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white))\end{aligned}$$

Using this assertion as a candidate loop invariant, I now revisit VC3 (see §3.3):

H1: $indexrange\_first \leq i$
H2: $j \leq indexrange\_last + 1$
H3: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq P-1)) \rightarrow ele(flag, [q]) = red)$
H4: $(\forall r : integer.((r \geq P) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$
H5: $\neg(i = j)$
H6: $\neg(ele(flag, [i]) = red)$
$\rightarrow$
C1: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq P-1)) \rightarrow$
$ele(\boxed{upd(upd(\boxed{flag}, [i], ele(flag, [j-1])), [j-1], ele(flag, [i]))}^{\uparrow}, [q]) = red)$
C2: $(\forall r : integer.((r \geq P) \wedge (r \leq indexrange\_last)) \rightarrow$
$ele(\boxed{upd(upd(\boxed{flag}, [i], ele(flag, [j-1])), [j-1], ele(flag, [i]))}^{\uparrow}, [r]) = white)$

Note that the existential variable $p$ has been replaced by meta-variable $P$. In terms of rippling, both C1 and C2 are blocked. These blockages correspond to the failure of precondition 3 of the ripple method. The patch involves range splitting, *i.e.* partitioning ranges $q$ and $r$ respectively so as to isolate the blocked wave-fronts. In the case of C1 the failure analysis suggests the following constraints:

$$((i < indexrange\_first) \vee (i > P - 1)) \wedge ((j - 1 < indexrange\_first) \vee (j - 1 > P - 1))$$

In the case of C2 the constraints are as follows:

$$((i < P) \vee (i > indexrange\_last)) \wedge ((j - 1 < P) \vee (j - 1 > indexrange\_last))$$

In addition we also require $i < j$, which can be derived from the code and the loop condition, *i.e.* $\neg(i = j)$. This is an example what is known as a bottom-up invariant generation. A lot of work on such techniques has been done, we will have to integrate bottom-up techniques within our top-down approach. The application of a constraint solver to the above problems will generate two alternative solutions for $P$. In the case of C1 an instantiation of $i$ is suggested while in the case of C2 an instantiation of $j$ is suggested. These instantiations are driven by the goal of isolating and then reducing the wave-fronts blocked within C1 and C2 of VC3 (§4). Collectively these instantiations suggest weakening the given postcondition into three ranges, *i.e.* a red range, a mixed range and a white range. The application of this patch involves refining the initial loop invariant assertion and re-submitting the code to the SPARK Examiner.

# 5 Conclusion

- Ripple method has a useful role play as expected.
- Ripple preconditions need strengthening with respect to reasoning about bounded quantifiers.
- This strengthening suggests a new proof critic, a range splitting critic that is triggered by a blocked ripple within the context of bounded quantification.
- Range splitting can be viewed as a form case splitting, so this new proof critic can be viewed as a generalization of the previous proof critic attached to the failure of precondition 3 of the ripple method.
- This new critic calls for the use of constraint solving and the ability to (potentially) critique over multiple branches within the goal tree.
- This new critic will have to be integrated with bottom-up techniques for generating invariant assertions.
- On inspecting the output of the SPADE Simplifier for the Polish Flag Problem, it appears that the proof planning presented above would also be applicable to the simplified VCs. However, in general the SPADE Simplifier may transform VCs in such a way that our proof plans are no longer applicable. The verification of the `Find_Smallest` function (see [2]) is a good example of this phenomenon and will be the topic of a future note. The trade-offs between simplification vs proof planning will need consideration.

# 6 Postscript

Previously we automated the discovery of an invariant for the two colour partitioning problem [3]. This was based upon the verification of a algorithm given by Backhouse [1]. Interestingly the invariant discovery techniques that worked previously for us failed on the SPARK implementation of algorithm given here. The reason for our success with the Backhouse implementation can be explained by the postcondition that was used. Basically Backhouse makes explicit use of one of the local variables within the postcondition. This gives a strong hint that range splitting is required. The more principled use of an existential quantifier within the postcondition, as advocated by the SPARK approach, makes for a more interesting proof planning problem!

# References


[1] R.C. Backhouse. *Program Construction and Verification.* Prentice Hall, 1986.

[2] J. Barnes. *High Integrity Ada: The Spark Approach.* Addison-Wesley, 1997.

[3] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

# Appendix A: SPARK code for the Polish Flag Problem

```
-- Coded by Bill J. Ellis
--
IndexUpper: constant := 4;
IndexLower: constant := 1;
subtype IndexRange is Integer range IndexLower .. IndexUpper;
subtype PointerRange is Integer range IndexRange'First .. IndexRange'Last+1;
type Colour is (Red, White);
type ArrayOfColours is array (IndexRange) of Colour;

...

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)));

...

-- 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;
 end Partition_Section;
```

## Appendix B: VCs generated for the Polish Flag Problem

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


                        DATE : 25-MAR-2002 16:53:30.76

                    procedure PolishFlag.Partition_Section




For path(s) from start to assertion of line 94:

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 <= 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)) .


For path(s) from assertion of line 94 to assertion of line 94:

procedure_partition_section_2.
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:    not (i = j) .
H7:    element(flag, [i]) = red .
        ->
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)) .


procedure_partition_section_3.
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:    not (i = j) .
H7:    not (element(flag, [i]) = red) .
        ->
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)) .


For path(s) from assertion of line 94 to finish:

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:    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))))) .
```