CS-Note* 35

Bill J. Ellis
January 9, 2003

Learning by Example - Table Utilities (Find)

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

The learn by example strategy, briefly outlined in CS-note 31, is continued for the table utilities example started in CS-note 32.

1.1 Table Utilities

Reasoning with functions in SPARK requires additional effort in creating proof functions. For this reason a procedure version of the Find example is being used. Such changes should not impact on the topic of this note.

2 Delete inspired Questions, Observations and Insights

A number of interesting points have arisen since previously tackling the Delete procedure in CS-note 32.

2.1 Questions (And answers)

2.1.1 How are Procedures Treated In SPARK?

Where a procedure is called in a SPARK subprogram the corresponding VCs are modelled using the policies:

- **Check at point of call -** A check statement is added to ensure that the preconditions of the called procedure are satisfied just prior to the procedure call. This results in an additional VC to be proved. Note that if there is no precondition on the called procedure then no check is required.

  Where dealing with exception freedom there is always an implicit precondition that the arguments passed to the called procedure are within the specified type.

- **Procedure as assignment statements -** The operation of the procedure is captured by introducing new variables to describe its exported values. By introducing new variables the internal operation of the procedure is not carried through. There is no automatically generated relationship between the input values and exported values, as might be expected if the algorithm in the called procedure were embedded (inlined) inside the calling procedure. Hiding the details of the called procedure ensures that the proofs remain manageable. Otherwise overly technical details of deeply nested procedures may find their way into top level VCs.

  Although the relationship between input and exported values is not automatically generated from the code, it can be added through a postcondition. If the called procedure has a postcondition it is is filled with the concrete names of the initial values and exported values and added as a hypothesis after the procedure call.

  Where dealing with exception freedom there is always an implicit postcondition that the exported values returned from the called procedure are within the specified type.

See Appendix A for example VCs generated from a procedure Quad that calls another procedure Double. Note how the Quad VCs include new variables *x__1* and *x__2* as a result of calling the Double procedure.

2.1.2 What is the Role Of Initialisation In Exception Freedom Proofs?

Accessing a variable before initialisation is not a runtime error in Ada. A good Ada compiler flags warnings where it detects variables being used prior to their initialisation. More sophisticated and formal detection is provided using data flow analysis in SPARK.

During the generation of exception freedom VCs it is assumed that all variables have well defined values prior to their use. Thus there must be a supplemental proof that every variable is initialised prior to its use. This proof is usually performed automatically by the SPARK data flow analysis returning no errors.

However, where the data flow analysis detects a potential error an alternative proof must be provided to show that this potential error does not occur. The success of the data flow analysis depends on its granularity. Currently the data flow analysis considers each variable and each field of a record separately, however it considers an array a single item. Thus the data flow analysis is weaker at automatically detecting the proper initialisation of arrays.

Fortunately, Ada supports an aggregate structure which can be used to initialise arrays in one statement. This initialisation is correctly detected by the data flow analysis, as it writes to the whole array in one go. This kind of array initialisation can be used to perform most required initialisation without leaving the scope data flow analysis.

There may be occasions where it is logically not required to initialise the whole array, or a more sophisticated kind of initialisation is required that is outside the scope of Ada aggregate. In these cases suitable proof effort is required to prove initialisation[1]. Alternatively, the array can be naively or unusually pre-initialised using an aggregate to gain the the automatic initialisation proof of the data flow analysis. In practise all arrays are expected to be initialised using an array aggregate.

2.1.3 What Should the Process of Finding Information From VCs be Called?

Key to proving a program meets some property, including exception freedom, is adding additional information. Historically this information is found by examining the program source code (bottom-up techniques) or from examining the program specification (top-down techniques).

By their very nature, VCs contain fragments of the program source code and specification. It is possible to analyse these VCs to gain additional information. This is generally most appropriate at the point where the VC proof fails. Although containing fragments of information used in bottom-up and top-down techniques they are considered sufficiently removed from this primary source as to warrant a separate classification. The process of finding information from VCs shall be called VC Analysis.

2.2 Observations

2.2.1 Evidence for Using FRiENDLY

In proving Delete the following VC was reached:

```
[H1a: fld_size(table) <= 100
 H1b: fld_size(table) => 0
 H1c: find(table, value) >= 0
 H1d: find(table, value) <= 100
 H1e: not (find(table, value) = 0)
 H1f: find(table, value) <= fld_size(table) - 1
 H1g: 0 < find(table, value)
 ...]
->
C1a: fld_size(table) => 2
```

A case split on `H1f` was used, with this choice being justified by matching the variable *`fld_size(table)`* on the same side of the inequality in the conclusion. This reasoning would be more straight forward if the original conjecture was used:

```
C2a: fld_size(table) - 1 >= base_index_type__first .
```

Now the full structure *`fld_size(table) - 1`* can be matched. This issue of transformed structures is addressed by FRiENDLY as described in CS-note 17.

2.2.2 Unnecessary Case Split

Although the above case split leads to a proof, an alternative and more straight forward proof is available[2].

.

```
[H2a: find(table, value) >= 0
 H2b: find(table, value) <= fld_size(table) - 1
 ...]
->
C2a: fld_size(table) - 1 >= base_index_type__first .
```

The common variable *`find(table, value)`* in `H2a` and `H2b` can be exploited to add a new hypothesis. Rearranging to make the inequalities match:

```
find(table, value) >= 0
fld_size(table) - 1 >= find(table, value)
```

Using transitivity it can be found that:

[1]Proof of initialisation is not directly supported by SPADE. An informal proof or hand crafted VCs will be required.
[2]Thanks to Andrew Ireland for spotting this one.

```
fld_size(table) - 1 >= 0
```

Which matches the original presentation of the conclusion, if substituting the constant *base_index_type__first* for its actual value *0*. This use of transitivity is not uncommon. It is certainly a candidate for a proof planning method.

2.2.3 Prevent EXAMINER from Hiding VCs

The EXAMINER does not bother to output VCs that are trivially[3] true, instead reporting:

```
procedure_find_9.
*** true .          /* trivially true VC removed by Examiner */
```

Such trivial VCs tend to occur where their are minimal annotations.

The EXAMINER behaves in this manner for sensible efficiency reasons. It takes time for the SIMPLIFIER to load these VCs, which may have many hypotheses, only to find that the conclusions can be discharged directly.

Unfortunately, these hypotheses may include valuable information. By adding false or suitably difficult annotations the SIMPLIFIER can no longer discharge these and thus the full hypotheses information is outputted.

2.2.4 Type Is Always Helpful

Type information is straight forward to discover using bottom-up techniques. In many cases type information alone is sufficient to discharge many of the more trivial VCs. Adding type information as a first step, prior to any VC failure analysis, may be more efficient as the real problems are reached more quickly.

2.3 Insights

2.3.1 Strengthen Specifications Using Context

Where subprograms[4] are called from within a subprogram they may need to be strengthened to complete a proof. For example, to prove Delete it was necessary to strengthen the postcondition of Find.

Originally it was thought that as a procedure is tackled its specification should be strengthened as much as possible. However, such unconstrained strengthening is likely to result in excessive information, possibly hindering the chance of finding an automatic proof. Instead, specifications of called subprograms may be strengthened while tackling their caller subprograms. The information present in tackling the caller subprograms adds additional, and valuable, constraints to how to strengthen the called subprograms.

2.3.2 Consistency Of Proof Where Changing Specifications

Where specifications are being automatically refined there becomes a need to ensure the consistency of the proof effort. It is essential that all of the code is proved under a single consistent specification.

Dealing with calling subprograms first leads to a top down proof strategy. Whenever the specification on a called subprogram is modified all of the procedures that call this subprogram will need to be proved again.

It may be possible to avoid this need for repeating proofs by using some lightweight reasoning. If it can be shown that the old postcondition follows from the new postcondition then any proofs using the old specification will continue to be valid. Such a technique has the potential to leave a misleading (confusing) collection of proofs. However, if performance becomes restrictive, this might bring some efficiency gains.

2.3.3 Strengthening Preconditions

The stronger a precondition the less modes of operation a subprogram has. For example, a sort subprogram may take an array and bounds `Lower` and `Upper`. With no specification the sort subprogram must behave correctly where `Lower` > `Upper`. However, if the precondition `Lower` < `Upper` is added the subprogram need no longer support this behaviour.

There may be benefits in automatically strengthening preconditions in subprograms. Where this is done it is important that all subprograms that call this subprogram are proved with the extra precondition in place. It is unlikely that the need for repeating proof can be avoided here. This would require proving that the new precondition follows from the the old precondition. In other words, that the changes made to the precondition did not logically add any new information. This certainly might be true in some unusual cases, but would not be expected in general.

[3]The conclusion is a Boolean expression that simplifies to true.
[4]Subprograms are either procedures or functions in SPARK.

2.3.4 Using VC Analysis to Patch Failures

In proving Delete some effort was placed on strengthening the specification of Find using bottom-up techniques. Although a solution was reached, it appears unlikely that the reasoning involved would scale up to bigger problems. An alternative is to use a VC Analysis technique. At the point of failing to prove the following Delete VC, occurring after a call to Find, it can be discovered that this is a non-theorem:

```
[H3a: fld_size(table) >= 0
 H3b: fld_size(table) <= 100
 H3c: find(table, value) >= 0
 H3d: find(table, value) <= 100
 H3e: not (find(table, value) = 0)
 ...]
->
C3a: fld_size(table) >= 1 .
```

In [2] some progress has been made in correcting false conjectures. To turn a non-theorem into a theorem some additional information in the form of a new hypothesis HYP needs to be added. For practical reasons this information should:

- Be consistent with all existing hypotheses. Making a non-theorem by introducing a false hypothesis, either directly or indirectly, is unlikely to be useful.
- Not be trivial, such as repeating the conclusion. Although logically sound, such trivial solutions are unlikely to be useful.
- Add the minimum necessary information to turn the non-theorem into a theorem.

While all of these points are valid in our problem, the latter need not be seen as an unbreakable rule. Certainly it is considered beneficial to minimise the amount of unnecessary information added in completing a program proof. However, identifying and using the exact minimum is likely to be difficult and bring little practical benefits.

In this case the missing HYP is *`not (fld_size(table) = 0)`*. Techniques for finding this were considered during the proof of Delete. Note that this information may be difficult to generate in general. However, in exception freedom proofs it seems that the frequency of inequalities might give a solid starting point to find such information. Nevertheless, this is viewed as the difficult blocking step where more sophisticated reasoning is required.

The new information can be added through code annotations. Using some bottom-up analysis it might be suggested that Find should add this information. The only other available option is the precondition to the procedure[5].

Adding this extra hypothesis directly to Find would lead to false VCs, as it can be the case that *`(fld_size(table) = 0)`*. However, the extra information is not needed in all situations. It is only required in the context of this VC. The added specification might be:

```
fld_size(table) >= 0          and
fld_size(table) <= 100        and
find(table, value) >= 0       and
find(table, value) <= 100     and
not (find(table, value) = 0) and
...
->
not (fld_size(table) = 0)
```

This should be sufficient for a proof. However, only *`not (find(table, value) = 0)`* is required on the left side of the implication. Trimming down the unnecessary VCs may be achieved by attempting proof of Find using only the left side of the implication as annotations. All those conclusions that can be solved are covered by the existing information in Find, and can be removed. Only that which could not be proved need be used in the left side of the implication in the final generated annotation.

[5] It may be that strengthening the precondition of Delete would be sufficient. The problem of identifying where to introduce new information may be more difficult than first appears.

2.3.5 Mixing VCs and Code is a Dilemma

As described above, the failed VC in Delete gives a strong indication that there is missing information and the form this information should take. The intention is to strengthen the code annotations such that the required information is introduced. There are various problems here:

- **Identify key annotations -** Only the key annotations should be added. These are expected to be of the form $Constraints \rightarrow Information$. As considered above, constraints can be omitted which can always be inferred from the existing information. That is, if the information to add at procedure X is $A \wedge B \rightarrow C$, and at X it is always known that A holds then the key annotation to add is $B \rightarrow C$.
- **Relating failed VC to annotation point -** The failed VC will be in terms of variables that are within a particular scope. It is necessary to relate the failed VC to the point where annotation will be introduced. This is not straight forward, as it requires knowing:
  - The program variable names for each VC variable.
  - If annotating a called subprogram, which variables are passed into this subprogram and what names they take.
  - Avoiding any name clashes.

> **A note on...**
> The RUNCHECK system [1] adds tags into VCs to identify which direction was taken at each program branch, and when loops are entered and exited. Using this information in combination with the failed VC, the last loop which modified a variable in the failed VC can be identified. Based on a heuristic this point is where RUNCHECK adds additional information in an attempt to prove the failed VC.
> Note that the tag information is also used for generating the key annotation required. The constraints are equivalent to the conjunction of branch tests that must be true to reach the failed VC. This information is directly available through the tags.

- **Restricting (dividing) the information -** If annotating the postcondition of a called subprogram it is only possible to describe the variables of that subprogram. The information in the failed VC, inside a different caller subprogram, may suggest adding further information that can not be described at this point.

  A subset of information from the failed VC which is in scope at annotation points can be used instead. In these cases the subset may be sufficient, or perhaps annotation will need to be added to more than one point in the program.
- **Identify where to annotate -** There are many points where the additional annotations might be introduced. These include the precondition of the subprogram, the postconditions of any previously called subprograms, the invariants of any enclosing or previous loops and appropriately placed `--#check;` statements.

  The correct place to add the annotation is where it is most efficient. It is more correct to add the annotation once to a called subprogram postcondition rather than many caller subprograms. The annotations should be close to the code that manipulates the properties being described, and probably within the same scope. As noted above, it may be possible or necessary to add multiple annotations to introduce the information required.

  The available locations might be identified by tracing backwards from the program point that spawned the failed VC. Each point that transforms the variables of the failed VC and allows for annotation is a potential candidate. The ideal location might be found by various heuristics, for example preferring to annotate called subprograms, looking for more variables to be modified at the annotation point, targeting closer annotation points.

All of these problems stem from wanting to treat VCs and program code together. These are different entities, separated by the process of generating VCs, as described in Floyd-Hoare Logic.

The RUNCHECK system adds tags to the VCs to describe code information. This dilutes the idea of VCs being entirely independent from the code. Alternatively, the information could be stored as a separate resource, code-meta-data. The connection between this code-meta-data and the VCs could be established through some association pointers. This view is illustrated in 1.

It is the case that the code contains information that will help in discharging VCs. However, developing mechanisms that exploit this information are unlikely to be straight forward.

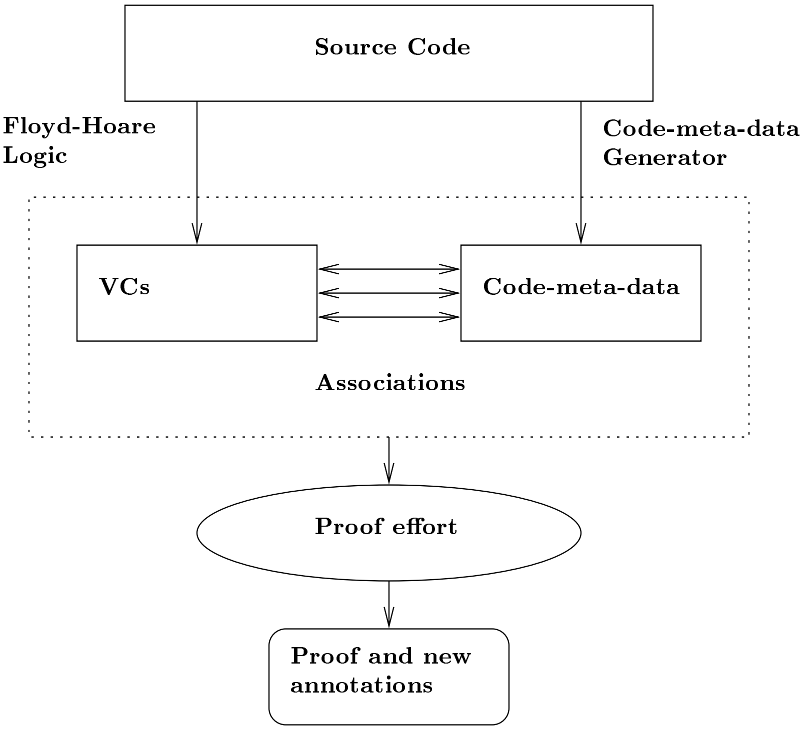

Figure 1: Working with code as meta data

3 Pre Analysis

It was suggested while incrementally tackling Delete that some analysis can be performed prior to the proof. This has the potential to reduce the number of iterations required to find a complete proof.

3.1 Adding Type Information

The procedure Find is annotated with type information:

```
procedure Find(Table: in Table_Type;
               Target: in Contents_Type;
               Result: out Base_Index_Type)
is
begin
   Result:= 0;
   for I in Index_Type range 1 .. Table.Size loop
      if Table.Elem(I) = Target then
         Result := I;
         exit;
      end if;
   end loop;
end Find;
```

The EXAMINER adds type information for the imported values. This precondition might[6] look like:

```
--#pre
--#    (Table.Size in Base_Index_Type)        and
--#    (for all I in Index_Type =>
--#         (Table.Elem(I) in Contents_Type)) and
--#    (Target in Contents_Type)              and
--#    (Result in Base_Index_Type);
```

[6]As these are generated internally by the EXAMINER the exact form is not known.

The EXAMINER adds type information into the invariant of the for loop. This only seems to consider the base type, and thus might look like:

```
--#assert
--#    (I in Index_Type);
```

It is expected that the range type will be desired also. It is added to the EXAMINER generated invariant:

```
--#assert
--#    (I >= 1 and I <= Table.Size);
```

3.2 Adding Extra Annotation To Prove Delete

Annotation is added to the postcondition of Find to satisfy proofs in Delete:

```
  --#post (Result/=0 -> Table.Size/=0);
```

This is inspired from the failed VC in tackling Delete:

```
H1: for_all(i___1:base_index_type,i___1>=index_type__first and i___1<=index_type
__last->element(fld_elem(table),[i___1])>=contents_type__first and element(fld_e
lem(table),[i___1])<=contents_type__last) .
H2: fld_size(table)>=base_index_type__first .
H3: fld_size(table)<=base_index_type__last .
H4: value>=contents_type__first .
H5: value<=contents_type__last .
H6: i__1>=base_index_type__first .
H7: i__1<=base_index_type__last .
H8: not i__1=0 .
       ->
C2: fld_size(table)-1>=base_index_type__first .
```

And finding that the extra information desired is:

```
not (fld_size(table) = 0)
```

The process of finding that the key constraints in the hypotheses is *`(not i__1)=0`*, discovering that *`i__1`* is Result inside Find and that strengthening the Find postcondition is a good place to add this information is far from straight forward to automate. These problems were elaborated earlier in section 2.3.5.

3.3 Generate and Simplify VCs

The annotation is added as described above to the Find procedure of the table utilities example. See Appendix B.1 for the code of this example. The EXAMINER is used to generate exception freedom VCs for the Find procedure. See Appendix B.3 for the VCs of this example and for the additional associated files. The VCs are passed to the SIMPLIFIER. See Appendix B.4 for the simplified VCs of this example. The result of this process is summarised in a table through POGS:

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

4 Looking at VC 2

Location information

```
For path(s) from start to run-time check associated with statement of line 8:

procedure_find_2.
```

Code snippet

```
6      begin
7         Result:= 0;
8         for I in Index_Type range 1 .. Table.Size loop
9            --#assert
10           --#    (I >= 1 and I <= Table.Size);
11           if Table.Elem(I) = Target then
```

Before simplification

```
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    target >= contents_type__first .
H6:    target <= contents_type__last .
H7:    0 >= base_index_type__first .
H8:    0 <= base_index_type__last .
        ->
C1:    (1 <= fld_size(table)) -> ((fld_size(table) >=
           index_type__first) and (fld_size(table) <=
           index_type__last)) .
C2:    (1 <= fld_size(table)) -> ((1 >= index_type__first) and (1 <=
           index_type__last)) .
C3:    fld_size(table) >= base_index_type__first .
C4:    fld_size(table) <= base_index_type__last .
C5:    1 >= base_index_type__first .
C6:    1 <= base_index_type__last .
```

After simplification

```
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    target >= - 1000 .
H5:    target <= 1000 .
       ->
C1:    1 <= fld_size(table) -> fld_size(table) >= 1 and fld_size(table) <= 100 .
```

This VC is concerned with the expression I in Index_Type range 1 .. Table.Size. If the range 1 .. Table.Size is not null (1 <= Table.Size) then it must be shown that both 1 and Table.Size lie inside their super type Index_Type. It has already been shown that this is the case for 1 but remains to be shown for Table.Size. It also must be proved here that both 1 and Table.Size are always within the base type of Index_Type, Base_Index_Type. This has already been proved by the SIMPLIFIER.

4.1 Proving unsimp-C1

Conclusion unsimp-C1 is tackled. Its key details are:

```
[H1a: fld_size(table) >= base_index_type__first .
 H1b: fld_size(table) <= base_index_type__last .
 ...]
->
C1a: (1 <= fld_size(table)) -> ((fld_size(table) >=
      index_type__first) and (fld_size(table) <=
      index_type__last)) .
```

There is no information in the hypotheses to describe *index_type__last*. However, the rls file can be used to add the constant values for *index_type__last* and *base_index_type__last*, effectively putting both of these in a common form and thus making the hypotheses applicable to the conclusion:

```
[H2a: fld_size(table) >= 0 .
 H2b: fld_size(table) <= 100 .
 ...]
->
C2a: (1 <= fld_size(table)) ->
        ((fld_size(table) >= 1) and (fld_size(table) <= 100)) .
```

This is now trivially true. The conclusion is essentially off the form $A \rightarrow A \wedge B$, which can be rewritten to $A \rightarrow B$, then $(\neg A) \vee B$. As B is *fld_size(table) <= 100*, it will match with H2b, thus the conclusion is true. This proof might also be performed using proof by implication, that by assuming A it is possible to show A and B.

4.2 Proving unsimp-C1 in the SIMPLIFIER

The SIMPLIFIER does not discharge this fairly trivial VC. It seems that the SIMPLIFIER does consider forward chaining implication hypotheses, which is a similar action as seen in proof by implication. It also seems that the SIMPLIFIER does tackle conclusions of the form $A \rightarrow B$, however it does this by *separately* tackling $\neg A$ and B. The key to proof here is considering the two together and exploiting the law of excluded middle[7] available in SPADE classical logic. This would make a plausible explanation for this odd weakness of the SIMPLIFIER and might even suggest some SIMPLIFIER modifications.

Unfortunately, this seems a difficult property to neatly drop into the SIMPLIFIER. The law of excluded middle already exists:

```
/* OR4 */    simplify((not X) or X,true) :- !.
```

However, as all hypotheses are global within the SIMPLIFIER, the subgoaling and isolated scope that proof by implication entails is difficult to model. It might be possible to instead perform the rewrite from an implication $A \rightarrow B$ to $(\neg A) \vee B$ and rely on further logical transformations. However, it seems that distributivity rules key to transforming the goal into $(\neg X) \vee X$ are not available, possibly to guarantee termination of the SIMPLIFIER.

[7] For any proposition, either it or its negation is true. Thus for any proposition P, $P \vee (\neg P)$ is true.

A Double Example

A.1 Code

A.1.1 Specification

```
package DoubleExample is
    procedure Double(X : in out Integer);
    --# derives X from X;

    procedure Quad(X : in out Integer);
    --# derives X from X;
end DoubleExample;
```

A.1.2 Body

```
1       package body DoubleExample is
2           procedure Double(X : in out Integer)
3           is
4           begin
5               X := X * 2;
6           end Double;
7
8           procedure Quad(X : in out Integer)
9           is
10           begin
11               Double(X);
12               Double(X);
13           end Quad;
14        end DoubleExample;
```

A.2 Exception Freedom VCs

A.2.1 Double

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


                       DATE : 09-AUG-2002 13:13:14.75

                       procedure DoubleExample.Double




For path(s) from start to run-time check associated with statement of line 5:

procedure_double_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
        ->
C1:    x * 2 >= integer__first .
C2:    x * 2 <= integer__last .


For path(s) from start to finish:

procedure_double_2.
*** true .          /* trivially true VC removed by Examiner */
```

A.2.2 Quad

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


                       DATE : 09-AUG-2002 13:13:14.78

                        procedure DoubleExample.Quad




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

procedure_quad_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
        ->
C1:    x >= integer__first .
C2:    x <= integer__last .


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

procedure_quad_2.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    x >= integer__first .
H5:    x <= integer__last .
H6:    x__1 >= integer__first .
H7:    x__1 <= integer__last .
        ->
C1:    x__1 >= integer__first .
C2:    x__1 <= integer__last .


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

procedure_quad_3.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    x >= integer__first .
H5:    x <= integer__last .
H6:    x__1 >= integer__first .
H7:    x__1 <= integer__last .
H8:    x__1 >= integer__first .
H9:    x__1 <= integer__last .
        ->
C1:    x__1 >= integer__first .
C2:    x__1 <= integer__last .


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

procedure_quad_4.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    x >= integer__first .
H5:    x <= integer__last .
H6:    x__1 >= integer__first .
H7:    x__1 <= integer__last .
H8:    x__1 >= integer__first .
H9:    x__1 <= integer__last .
H10:   x__1 >= integer__first .
H11:   x__1 <= integer__last .
H12:   x__2 >= integer__first .
H13:   x__2 <= integer__last .
        ->
C1:    x__2 >= integer__first .
C2:    x__2 <= integer__last .


For path(s) from start to finish:

procedure_quad_5.
*** true .          /* trivially true VC removed by Examiner */
```

B Table Utilities Example

B.1 Code

B.1.1 Specification

```
-- Table Utilities for Run-Time Errors Case Study
-- for vc generation with rtc and simplification
-- (Taken from:
-- Proof Of The Absence Od Run-Time Errors from
-- SPARK Programs -- A Case Study)
package Table_Utilities is
    Max_Table_Size: constant := 100;
    type Base_Index_Type is range 0 .. Max_Table_Size;
    subtype Index_Type is Base_Index_Type range 1 .. Max_Table_Size;
     type Contents_Type is range -1000 .. 1000;
     type Array_Type is array (Index_Type) of Contents_Type;
     type Table_Type is record
                         Size: Base_Index_Type;
                         Elem: Array_Type;
                       end record;
     procedure Find(Table: in Table_Type;
                    Target: in Contents_Type;
                    Result: out Base_Index_Type);
     --#derives Result from Table, Target;
     --#post (Result/=0 -> Table.Size/=0);
     --  returns Key s.t. Key/=0 -> Table.Elem(Key)=Target and
     --                   Key=0  -> for all i in 1..Table.Size,
     --                                Table.Elem(i) /= Target;

     procedure Sort(Table: in out Table_Type);
     --#derives Table from Table;
     --  sorts the Table.Elem array into ascending numerical order, preserving
     --  contents and size

     procedure Insert(Table: in out Table_Type;
                      Sorted, Unique: in Boolean;
                      Value: in Contents_Type;
                      Success: out Boolean);
     --#derives Table, Success from Table, Sorted, Unique, Value;
     --  inserts Value into the Table.Elem, such that:
     --    - if Sorted is true, inserts at a point in ascending numerical
     --      order, otherwise inserts at end of array;
     --    - if Unique is true, does not create multiple entries, otherwise
     --      will allow more than one entry with same value;
     --    - if there is no room for the insertion (Table.Size=100 and an
     --      insert is needed), does nothing to Table and returns a Success of
     --      False, otherwise returns a Success of True.

     procedure Delete(Table: in out Table_Type;
                      Value: in Contents_Type;
                      Success: out Boolean);
     --#derives Table, Success from Table, Value;
 end Table_Utilities;
```

B.1.2 Body

```
package body Table_Utilities is
    procedure Find(Table: in Table_Type;
                   Target: in Contents_Type;
```

```
4                       Result: out Base_Index_Type)
5         is
6         begin
7            Result:= 0;
8            for I in Index_Type range 1 .. Table.Size loop
9               --#assert
10              --#     (I >= 1 and I <= Table.Size);
11              if Table.Elem(I) = Target then
12                 Result := I;
13                 exit;
14              end if;
15           end loop;
16        end Find;
17        -------------------------------------------------------------------------
18        procedure Sort(Table: in out Table_Type)
19        is
20           Key: Index_Type;
21           function Find_Smallest(Arr: Array_Type;
22                                  L, U: Index_Type) return Index_Type
23           is
24              K: Index_Type;
25           begin
26              K := L;
27              for I in Index_Type range L+1 .. U loop
28                 if Arr(I) < Arr(K) then
29                    K := I;
30                 end if;
31              end loop;
32              return K;
33           end Find_Smallest;
34           procedure Swap_Elements(T: in out Table_Type; I, J: in Index_Type)
35           --#derives T from T, I, J;
36           is
37              Temp: Contents_Type;
38           begin
39              Temp := T.Elem(I);
40              T.Elem(I) := T.Elem(J);
41              T.Elem(J) := Temp;
42           end Swap_Elements;
43
44        begin   -- Sort
45            for Low in Index_Type range 1 .. Table.Size-1 loop
46                Key := Find_Smallest(Table.Elem, Low, Table.Size);
47                if Key /= Low then
48                    Swap_Elements(Table, Low, Key);
49                end if;
50            end loop;
51        end Sort;
52        -------------------------------------------------------------------------
53        procedure Insert(Table: in out Table_Type;
54                         Sorted, Unique: in Boolean;
55                         Value: in Contents_Type;
56                          Success: out Boolean)
57        is
58            Successful: Boolean;
59            I: Base_Index_Type;
60        begin
61            if Sorted then
62                Find(Table, Value, I);
63                if Unique and I/= 0 then
64                    Successful := true;
65                else
66                    Successful := Table.Size < Max_Table_Size;
67                    if Successful then
68                        --! Sorted, there's space & insert, so find place and insert
69                        I := 1;
70                        loop
71                            exit when I > Table.Size or else
72                                      Table.Elem(I) >= Value;
73                            I := I + 1;
74                        end loop;
75                        for X in reverse Index_Type range I .. Table.Size loop
76                            Table.Elem(X+1) := Table.Elem(X);
77                        end loop;
78                        Table.Elem(I) := Value;
79                        Table.Size := Table.Size + 1;
80                    end if;
81                end if;
82            else
83                if Unique then
84                    --! not Sorted, Unique: insert at end if not already in Table
85                    Find(Table, Value, I);
86                    Successful := I /= 0;
87                    if not Successful then
88                        Successful := Table.Size < Max_Table_Size;
89                        if Successful then
90                            Table.Size := Table.Size + 1;
91                            Table.Elem(Table.Size) := Value;
92                        end if;
93                    end if;
94                else
95                    --! not Sorted, not Unique: insert regardless, at end
96                    Successful := Table.Size < Max_Table_Size;
97                    if Successful then
98                        Table.Size := Table.Size + 1;
99                        Table.Elem(Table.Size) := Value;
100                    end if;
101               end if;
102           end if;
103           Success := Successful;
104        end Insert;
105        -------------------------------------------------------------------------
106        procedure Delete(Table: in out Table_Type;
107                         Value: in Contents_Type;
108                         Success: out Boolean)
109        is
110           I: Base_Index_Type;
111        begin
112           --# check I in Base_Index_Type;
113           Find(Table, Value, I);
114           if I = 0 then
115               Success := false;
116           else
117               Success := true;
118               for J in Index_Type range I .. Table.Size - 1 loop
119                   --#assert J in Index_Type and
120                   --#       J in I .. Table.Size - 1 and
121                   --#       I in Base_Index_Type;
122                   Table.Elem(J) := Table.Elem(J+1);
123               end loop;
124               Table.Size := Table.Size - 1;
125           end if;
126        end Delete;
127
128     end Table_Utilities;
```

B.2 Extra Information

B.2.1 RLS File

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


                     /*DATE : 22-OCT-2002 13:55:55.64*/

                     /*procedure Table_Utilities.Find*/


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

find_rules(1): character__pos(X) may_be_replaced_by X.
find_rules(2): character__val(X) may_be_replaced_by X.
find_rules(3): base_index_type__first may_be_replaced_by 0.
find_rules(4): base_index_type__last may_be_replaced_by 100.
find_rules(5): base_index_type__base__first <= base_index_type__base__last may_be_deduced.
find_rules(6): base_index_type__base__first <= base_index_type__first may_be_deduced.
find_rules(7): base_index_type__base__last >= base_index_type__last may_be_deduced.
find_rules(8): index_type__first may_be_replaced_by 1.
find_rules(9): index_type__last may_be_replaced_by 100.
find_rules(10): index_type__base__first <= index_type__base__last may_be_deduced.
find_rules(11): index_type__base__first <= index_type__first may_be_deduced.
find_rules(12): index_type__base__last >= index_type__last may_be_deduced.
find_rules(13): contents_type__first may_be_replaced_by -1000.
find_rules(14): contents_type__last may_be_replaced_by 1000.
find_rules(15): contents_type__base__first <= contents_type__base__last may_be_deduced.
find_rules(16): contents_type__base__first <= contents_type__first may_be_deduced.
find_rules(17): contents_type__base__last >= contents_type__last may_be_deduced.
```

B.2.2 FDL File

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


                     {DATE : 22-OCT-2002 13:55:55.63}

                     {procedure Table_Utilities.Find}


title procedure find;

  function round__(real) : integer;
  type contents_type = integer;
  type array_type = array [integer] of integer;
  type table_type = record
        size : integer;
        elem : array_type
     end;
  type base_index_type = integer;
  const contents_type__last : integer = pending;
  const contents_type__first : integer = pending;
  const index_type__last : integer = pending;
  const index_type__first : integer = pending;
  const base_index_type__last : integer = pending;
  const base_index_type__first : integer = pending;
  var result : integer;
  var loop__1__i : integer;
  var target : integer;
  var table : table_type;

end;
```

B.3 Exception Freedom VCs (Find)

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


                     DATE : 22-OCT-2002 13:55:55.62

                     procedure Table_Utilities.Find




For path(s) from start to run-time check associated with statement of line 7:

procedure_find_1.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
```

```
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    target >= contents_type__first .
H6:    target <= contents_type__last .
        ->
C1:    0 >= base_index_type__first .
C2:    0 <= base_index_type__last .


For path(s) from start to run-time check associated with statement of line 8:

procedure_find_2.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    target >= contents_type__first .
H6:    target <= contents_type__last .
H7:    0 >= base_index_type__first .
H8:    0 <= base_index_type__last .
        ->
C1:    (1 <= fld_size(table)) -> ((fld_size(table) >=
           index_type__first) and (fld_size(table) <=
           index_type__last)) .
C2:    (1 <= fld_size(table)) -> ((1 >= index_type__first) and (1 <=
           index_type__last)) .
C3:    fld_size(table) >= base_index_type__first .
C4:    fld_size(table) <= base_index_type__last .
C5:    1 >= base_index_type__first .
C6:    1 <= base_index_type__last .


For path(s) from start to assertion of line 10:

procedure_find_3.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    target >= contents_type__first .
H6:    target <= contents_type__last .
H7:    0 >= base_index_type__first .
H8:    0 <= base_index_type__last .
H9:    (1 <= fld_size(table)) -> ((fld_size(table) >=
           index_type__first) and (fld_size(table) <=
           index_type__last)) .
H10:   (1 <= fld_size(table)) -> ((1 >= index_type__first) and (1 <=
           index_type__last)) .
H11:   fld_size(table) >= base_index_type__first .
H12:   fld_size(table) <= base_index_type__last .
H13:   1 >= base_index_type__first .
H14:   1 <= base_index_type__last .
H15:   1 <= fld_size(table) .
        ->
C1:    1 >= 1 .
C2:    1 <= fld_size(table) .
C3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
C4:    fld_size(table) >= base_index_type__first .
C5:    fld_size(table) <= base_index_type__last .
C6:    target >= contents_type__first .
C7:    target <= contents_type__last .
C8:    1 >= index_type__first .
C9:    1 <= index_type__last .


For path(s) from assertion of line 10 to assertion of line 10:

procedure_find_4.
H1:    loop__1__i >= 1 .
H2:    loop__1__i <= fld_size(table) .
H3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H4:    fld_size(table) >= base_index_type__first .
H5:    fld_size(table) <= base_index_type__last .
H6:    target >= contents_type__first .
H7:    target <= contents_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   loop__1__i >= index_type__first .
H11:   loop__1__i <= index_type__last .
H12:   not (element(fld_elem(table), [loop__1__i]) = target) .
H13:   not (loop__1__i = fld_size(table)) .
        ->
C1:    loop__1__i + 1 >= 1 .
C2:    loop__1__i + 1 <= fld_size(table) .
C3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
C4:    fld_size(table) >= base_index_type__first .
C5:    fld_size(table) <= base_index_type__last .
C6:    target >= contents_type__first .
C7:    target <= contents_type__last .
C8:    loop__1__i + 1 >= index_type__first .
C9:    loop__1__i + 1 <= index_type__last .


For path(s) from assertion of line 10 to run-time check associated with statement of line 11:

procedure_find_5.
H1:    loop__1__i >= 1 .
H2:    loop__1__i <= fld_size(table) .
H3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
```

```
H4:    fld_size(table) >= base_index_type__first .
H5:    fld_size(table) <= base_index_type__last .
H6:    target >= contents_type__first .
H7:    target <= contents_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
        ->
C1:    loop__1__i >= index_type__first .
C2:    loop__1__i <= index_type__last .


For path(s) from assertion of line 10 to run-time check associated with statement of line 12:

procedure_find_6.
H1:    loop__1__i >= 1 .
H2:    loop__1__i <= fld_size(table) .
H3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H4:    fld_size(table) >= base_index_type__first .
H5:    fld_size(table) <= base_index_type__last .
H6:    target >= contents_type__first .
H7:    target <= contents_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   loop__1__i >= index_type__first .
H11:   loop__1__i <= index_type__last .
H12:   element(fld_elem(table), [loop__1__i]) = target .
        ->
C1:    loop__1__i >= base_index_type__first .
C2:    loop__1__i <= base_index_type__last .


For path(s) from start to finish:

procedure_find_7.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    target >= contents_type__first .
H6:    target <= contents_type__last .
H7:    0 >= base_index_type__first .
H8:    0 <= base_index_type__last .
H9:    (1 <= fld_size(table)) -> ((fld_size(table) >=
           index_type__first) and (fld_size(table) <=
           index_type__last)) .
H10:   (1 <= fld_size(table)) -> ((1 >= index_type__first) and (1 <=
           index_type__last)) .
H11:   fld_size(table) >= base_index_type__first .
H12:   fld_size(table) <= base_index_type__last .
H13:   1 >= base_index_type__first .
H14:   1 <= base_index_type__last .
H15:   not (1 <= fld_size(table)) .
        ->
C1:    (0 <> 0) -> (fld_size(table) <> 0) .


For path(s) from assertion of line 10 to finish:

procedure_find_8.
H1:    loop__1__i >= 1 .
H2:    loop__1__i <= fld_size(table) .
H3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H4:    fld_size(table) >= base_index_type__first .
H5:    fld_size(table) <= base_index_type__last .
H6:    target >= contents_type__first .
H7:    target <= contents_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   loop__1__i >= index_type__first .
H11:   loop__1__i <= index_type__last .
H12:   element(fld_elem(table), [loop__1__i]) = target .
H13:   loop__1__i >= base_index_type__first .
H14:   loop__1__i <= base_index_type__last .
        ->
C1:    (loop__1__i <> 0) -> (fld_size(table) <> 0) .


procedure_find_9.
H1:    loop__1__i >= 1 .
H2:    loop__1__i <= fld_size(table) .
H3:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H4:    fld_size(table) >= base_index_type__first .
H5:    fld_size(table) <= base_index_type__last .
H6:    target >= contents_type__first .
H7:    target <= contents_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   loop__1__i >= index_type__first .
H11:   loop__1__i <= index_type__last .
H12:   not (element(fld_elem(table), [loop__1__i]) = target) .
H13:   loop__1__i = fld_size(table) .
        ->
C1:    (result <> 0) -> (fld_size(table) <> 0) .
```

B.4 Simplified Exception Freedom VCs (Find)

```
          *******************************************************
                      Semantic Analysis of SPARK Text
```

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


       CREATED 22-OCT-2002, 13:55:55  SIMPLIFIED 22-OCT-2002, 13:56:32
                (Simplified by SPADE Simplifier, Demo Version)

                        procedure Table_Utilities.Find



For path(s) from start to run-time check associated with statement of line 7:

procedure_find_1.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 8:

procedure_find_2.
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    target >= - 1000 .
H5:    target <= 1000 .
       ->
C1:    1 <= fld_size(table) -> fld_size(table) >= 1 and fld_size(table) <= 100 .


For path(s) from start to assertion of line 10:

procedure_find_3.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 10 to assertion of line 10:

procedure_find_4.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 10 to run-time check associated with
          statement of line 11:

procedure_find_5.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 10 to run-time check associated with
          statement of line 12:

procedure_find_6.
*** true .          /* all conclusions proved */


For path(s) from start to finish:

procedure_find_7.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 10 to finish:

procedure_find_8.
*** true .          /* all conclusions proved */


procedure_find_9.
*** true .          /* all conclusions proved */
```

References

[1] Steven M. German. Automating proofs of the absence of common runtime errors. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 105–118. ACM Press, 1978.

[2] R. Monroy, A. Bundy, and A. Ireland. Proof plans for the correction of false conjectures. *Lecture Notes in Computer Science*, 822:54–68, 1994.