CS-Note* 45

Bill J. Ellis
June 13, 2003

Program Prover Needs the Programs!

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

In program verification three different sources of information are available:

- **Top-down analysis -** Extracting information from a programs specification.
- **Bottom-up analysis -** Extracting information from a programs source code.
- **VC analysis -** Extracting information from a programs VCs. The VCs contain a combination of the information that might be found through top-down and bottom-up analysis.

All three sources may contain information that could assist in a proof. However, especially in the case of the VCs, there is a possibility of overlap. Thus that which could be discovered from one information source may also be discovered from another. Where overlap exists a choice should be made to retrieve the information from the source which is most straight forward to access. Accessing the VCs is straight forward, however their are issues with suitability accessing the source:

- **A different approach -** Source code is necessary to generate VCs. However, once the VCs have been generated the source code is often removed from the verification process. It may be thought that the VCs give a richer description of the source such that the source itself becomes superfluous.
- **Strengthening the proof framework -** Integrating appropriate access to the source within a proof framework is not straight forward. Their is a desire for proof tools to have a definite problem to work with. If the VCs are this definite problem then the source code is out of place. It may be more convenient to strengthen the VCs with some code level information rather than actually provide access to the source.
- **Additional tool support -** Analysing the source code will at least require implementing a parser and analysis tools build around the parsed output.
- **Availability of the source code -** Their is a practical issue in that source code is the working part of a system and so subject to greater security constraints.

Recognising these issues the leap to the full treatment of program analysis was proceeded by searching for alternative strategies.

2 VC reverse engineering

VC reverse engineering is the name given here to the the process of reconstructing source code from the VCs alone. If feasible this would demonstrate that the VCs do contain all of the information that might be found in the source code. Further, the process of VC reverse engineering would reveal source code information without having to build a separate collection of tools to analyse the real source code. To establish the applicability of this approach it is attempted to reverse engineer the minimally annotated Phil Thorley example, shown in Appendix A, from its exception freedom VCs.

2.1 Note about SPARK VCs

As documented in [1] §6.8 the SPARK VCs contain comments describing the their relationship to the source code. These are of the form:

```
For path(s) from start to run-time check associated with statement of line 6:
```

It is possible to have multiple actions on a single code line in SPARK, preventing a definite statements based on these comments. However, for style reasons, it is uncommon for several statements to be packed onto the same line. Assuming reasonable layout, the line numbers uniquely identify cut points, providing the exact structure of the program.

Further, at the start of each VC file their is a header followed by a subprogram specifier indicating the kind of subprogram and its name of the form:

```
procedure PhilThornleyExample.SumArray
```

2.2 All comments

2.2.1 All cutpoints

```
For path(s) from start to run-time check associated with statement of line 6:
+procedure_sumarray_1.
```

```
For path(s) from start to default assertion of line 7:
+procedure_sumarray_2.
```

```
For path(s) from default assertion of line 7 to default assertion of line 7:
+procedure_sumarray_3.
+procedure_sumarray_4.
```

```
For path(s) from default assertion of line 7 to run-time check associated with statement of line 8:
+procedure_sumarray_5.
```

```
For path(s) from default assertion of line 7 to run-time check associated with statement of line 9:
+procedure_sumarray_6.
```

```
For path(s) from default assertion of line 7 to finish:
+procedure_sumarray_7.
+procedure_sumarray_8.
```

2.2.2 Subprogram specifier

```
procedure PhilThornleyExample.SumArray
```

2.3 Formulating skeleton

Adding labels:

```
Start...............|
Assertion at line 7..|
Statement at line 8..|
Statement at line 9..|
Finish..............|
```

Adding structure:

```
Start...............|
....................| procedure SumArray(some arguments)
....................|     statements A;
....................|     loop
Assertion at line 7..|
....................|         case
....................|             when something => statements B;
....................|             when others => statements C;
....................|         end case;
Statement at line 8..|     statements D; <--| May exist inside or
Statement at line 9..|     statements E; <--| outside the loop
....................|     end loop;        |
Statement at line 8..|     statements D; <--|
Statement at line 9..|     statements E; <--|
....................|
Finish..............|
....................| end name;
```

- The start always indicates the beginning of a subprogram. From the subprogram specifier it is known that this is a procedure, of name `SumArray`.
- Every statement that modifies the state will produce an exception freedom VC. Thus no VCs indicates no code statements.

- The invariant (assertion) indicates a loop of some form. The loop may be a for loop, while loop or general `exit when condition;` loop.
- The two VCs generated from the invariant indicate a branch inside the loop. This might be implemented through an `if` statement an `if..then..else` statement or `case` statement.
- The statements at line 8 and 9 are reached from the invariant. It is not possible to tell if these exist inside or outside the loop from the comments alone.
- There is no path from the start to the finish. The invariant within the loop is always entered.

2.4 Using the VCs

The VCs are considered in turn in an attempt to reconstruct the source code.

2.4.1 procedure_sumarray_1.

```
For path(s) from start to run-time check associated with statement of line 6:

H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
        ->
C1:    0 >= integer__first .
C2:    0 <= integer__last .
```

A recurring problem is that VCs combine annotations and properties of the code without recording which are which. It is assumed that it is known that the example under analysis is minimally annotated. As our initial emphasis is on exception freedom proofs this assumption need not exclude the process[1]. The hypothesis `H2` indicates that there exists an array called `data_set` that is indexed through `index` with values of range `integer`.

```
type data_set_type is array (index) of integer;
data_set: data_set_type;
```

The conclusions are checking that `0` lies in the range of `integer`. Assuming the code is minimally annotated this must be a runtime check added as the result of assigning `0` to a variable of type `integer`. Importantly, the name of the variable being assigned to is not known.

```
something':=0;
```

2.4.2 procedure_sumarray_2.

```
For path(s) from start to default assertion of line 7:

H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H3:    0 >= integer__first .
H4:    0 <= integer__last .
        ->
C1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
C2:    index__first >= index__first .
C3:    index__first <= index__last .
```

[1]Additionally, SPARK does tend to float specification led hypotheses and conclusions to the top of VCs, providing a possible hook to establishing which parts of the VC are describing the code.

This is describing entering the invariant. The repeated hypothesis `H2` also seen in `procedure_sumarray_1` is a pattern of SPARK default invariants, indicating that this is indeed a minimally annotated example. Knowing that this is a default SPARK invariant and assuming the example is minimally annotated, the extra conclusions `C2` and `C3` must have come from definite type on a loop counter. Definite type is only available for `for` loops. Further, the conclusions indicate that the type of this counter is `index`. Further, as this is a default loop invariant, the invariant will occur just after the `for` loop is entered.

```
for Counter in direction index loop
--#assert default invariant;
...
end loop;
```

2.4.3 procedure_sumarray_3 and 4

```
For path(s) from default assertion of line 7 to default assertion of line 7:

procedure_sumarray_3.
H1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H2:    loop__1__i >= index__first .
H3:    loop__1__i <= index__last .
H4:    loop__1__i >= index__first .
H5:    loop__1__i <= index__last .
H6:    loop__1__i >= index__first .
H7:    loop__1__i <= index__last .
H8:    element(data_set, [loop__1__i]) >= 0 .
H9:    element(data_set, [loop__1__i]) <= 100 .
H10:   result + element(data_set, [loop__1__i]) >= integer__first .
H11:   result + element(data_set, [loop__1__i]) <= integer__last .
H12:   loop__1__i >= index__first .
H13:   loop__1__i <= index__last .
H14:   not (loop__1__i = index__last) .
        ->
C1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
C2:    loop__1__i + 1 >= index__first .
C3:    loop__1__i + 1 <= index__last .

procedure_sumarray_4.
H1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H2:    loop__1__i >= index__first .
H3:    loop__1__i <= index__last .
H4:    loop__1__i >= index__first .
H5:    loop__1__i <= index__last .
H6:    loop__1__i >= index__first .
H7:    loop__1__i <= index__last .
H8:    not ((element(data_set, [loop__1__i]) >= 0) and (element(
           data_set, [loop__1__i]) <= 100)) .
H9:    not (loop__1__i = index__last) .
        ->
C1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
C2:    loop__1__i + 1 >= index__first .
C3:    loop__1__i + 1 <= index__last .
```

The two VCs for the invariant to invariant case indicates that a single branch occurs within the loop. By comparing the two VCs it can be found that H8 and H9 in procedure_sumarray_3 are the negation of H8 in procedure_sumarray_4. Thus the tests associated with the two cases in the loop are:

```
procedure_sumarray_3.
data_set(loop__1__i)>=0 and data_set(loop__1__i)<=100
```

```
procedure_sumarray_4.
data_set(loop__1__i)<0 or data_set(loop__1__i)>100
```

Although not necessarily implemented as an `if` statement, these two complementary paths can be presented as an `if` statement:

```
if data_set(loop__1__i)>=0 and data_set(loop__1__i)<=100 Then
...
else
...
end if;
```

Aside from the different tests, the only difference between these VCs is the additional H10 and H11 in procedure_sumarray_3. These are checking that `result + element(data_set, [loop__1__i])` is within the type `integer`. Assuming minimal annotations, this must be an exception freedom check, indicating an assignment to the expression `result + data_set(loop__1__i)` to a variable of type `integer`. That this occurs in the procedure_sumarray_3 indicates that this occurs in the `then` branch of the `if` statements.

```
if data_set(loop__1__i)>=0 and data_set(loop__1__i)<=100 then
    something'':=result+data_set(loop__1__i);
Else
...
end if;
```

The lack of extra hypotheses or conclusions in procedure_sumarray_4 branch indicates no changes to the program state. The `else` branch is empty:

```
if data_set(loop__1__i)>=0 and data_set(loop__1__i)<=100 Then
    something'':=result+data_set(loop__1__i);
end if;
```

Both procedure_sumarray_3 and procedure_sumarray_4 have conclusions C1 and C2, checking that `loop__1__i + 1` is inside the type `index`. It is known that the loop counter is off this type, and that no other check exists in both procedure_sumarray_3 and procedure_sumarray_4 to act as the loop variable. Further, H14 in procedure_sumarray_3 and H9 in procedure_sumarray_4 is a likely loop exit condition. Thus `loop__1__i` is selected as the loop counter and its direction is increasing, giving the `for` loop.

```
for loop__1__i in Index loop
--#assert default invariant;
...
end loop;
```

2.4.4 procedure_sumarray_5

```
For path(s) from default assertion of line 7 to run-time check associated with statement of line 8:

H1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H2:    loop__1__i >= index__first .
H3:    loop__1__i <= index__last .
        ->
C1:    loop__1__i >= index__first .
C2:    loop__1__i <= index__last .
C3:    loop__1__i >= index__first .
C4:    loop__1__i <= index__last .
```

It is being checked, twice, that `loop__1__i` lies inside the type `index`. The invariant VCs procedure_sumarray_3 and procedure_sumarray_4 have already gathered details on the statements seen in the loop. The only place that `loop__1__i` is checked twice is in the condition of the `if` statement.

2.4.5 procedure_sumarray_6

```
For path(s) from default assertion of line 7 to run-time check associated with statement of line 9:

H1:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H2:    loop__1__i >= index__first .
H3:    loop__1__i <= index__last .
H4:    loop__1__i >= index__first .
H5:    loop__1__i <= index__last .
H6:    loop__1__i >= index__first .
H7:    loop__1__i <= index__last .
H8:    element(data_set, [loop__1__i]) >= 0 .
H9:    element(data_set, [loop__1__i]) <= 100 .
        ->
C1:    result + element(data_set, [loop__1__i]) >= integer__first .
C2:    result + element(data_set, [loop__1__i]) <= integer__last .
C3:    loop__1__i >= index__first .
C4:    loop__1__i <= index__last .
```

Similar to procedure_sumarray_5, this ties with the invariant VCs, and the assignment something'':=result+data_set(loop__1__i).

2.4.6 procedure_sumarray_7 and 8

```
For path(s) from default assertion of line 7 to finish:

procedure_sumarray_7.
*** true .          /* trivially true VC removed by Examiner */

procedure_sumarray_8.
*** true .          /* trivially true VC removed by Examiner */
```

On exiting the loop no further exception freedom checks are performed, indicating that the loop is the last statement in the code.

2.5 Bringing together

Thus far the code as described by the VCs takes the form.

```
type data_set_type is array (index) of integer;
data_set: data_set_type;

Start...............|
....................| subprogram name(some arguments)
....................|     something':=0;
....................|     for loop__1__i Counter in Index loop
Assertion at line 7..|         --#assert default invariant;
Statement at line 8..|         if data_set(loop__1__i)>=0 and data_set(loop__1__i)<=100 Then
Statement at line 9..|             something'':=result+data_set(loop__1__i)
....................|         end if;
....................|     end loop;
Finish..............| end name;
```

This is correct aside from the missing names of the assignments which can at best be described `something`. However, the type of each `something` is known. A collection of `something` referring to the same type may be the same variable. Further heuristics may be employed, for example if a variable is initialised it is a counter, so it will appear on both sides of an assignment somewhere. Thus the `something` in this example may be replaced by `result`, completing the reverse engineer of the code:

```
type data_set_type is array (index) of integer;
data_set: data_set_type;

Start...............|
....................| subprogram name(some arguments)
....................|     result:=0;
....................|     for loop__1__i in Index loop
Assertion at line 7..|         --#assert default invariant;
Statement at line 8..|         if data_set(loop__1__i)>=0 and data_set(loop__1__i)<=100 Then
Statement at line 9..|             Result:=result+data_set(loop__1__i)
....................|         end if;
....................|     end loop;
Finish..............| end name;
```

2.6 Results

The above illustrates that the approach of performing program analysis on the VCs, rather than the code, is not necessarily impossible. However, the approach is built on various assumptions and rather shaky heuristics. Further, the example considered is far short from the richer algorithms expected to see in general SPARK programs. Although just about feasible to perform SPARK VC reverse engineering manually, it is certainly not obvious how to perform this automatically.

Unsurprising their is a great deal of source code information captured within the VCs. Unexpectedly, genuine progress can be made at reconstructing the code from the VCs. This indicates that VCs should be every bit as secure as the source code.

3 Exploiting the EXAMINER

VC reverse engineering is too complex to be a practical route to program analysis. It will be necessary to have the real source code available. However, it may yet be possible to minimise the scope of tools accessing this source code by exploiting the existing facilities of the EXAMINER.

3.1 Augmented VC reverse engineering

The assumptions made during VC reverse engineering can be quickly checked against the source when available. This is only appropriate if it is possible to check these assumptions using very light weight parsing of the source and that making this extra information available significantly reduces the complexity of VC reverse engineering. Neither of these two conditions are met for the small example above, and hence certainly not for full SPARK.

3.2 Specialised augmented VC reverse engineering

Additional information can be found to aid the VC reverse engineering process by adding an invariant to trace the values of each variable through the loop. This can be achieved by adding dummy proof functions:

```
--#function ID_I(I: Index) return Boolean;
--#function ID_Result(Result: Integer) return Boolean;
```

And then tracing each variable using these functions in the invariant:

```
--#assert ID_I(I) and ID_Result(Result);
```

Applied to the Phil Thorley example this extends the invariant to invariant VCs with:

```
procedure_sumarray_3.
H1:    id_i(loop__1__i) .
H2:    id_result(result) .
...
        ->
C1:    id_i(loop__1__i + 1) .
C2:    id_result(result + element(data_set, [loop__1__i])) .
...

procedure_sumarray_4.
H1:    id_i(loop__1__i) .
H2:    id_result(result) .
...
        ->
C1:    id_i(loop__1__i + 1) .
C2:    id_result(result) .
...
```

Once again, this extends the applicability of the VC reverse engineering. However, their are still issues in tracing variables outside loops and implementing such an ad-hoc program extraction system.

4 Conceptual shift for NuSPADE

Separate to the availability and parsing of source code is how the source code is eventually used. As discussed in previous CS-notes, NuSPADE will access two different kinds of information:

- **The VCs -** The VCs are the problem to be solved. The proof planner creates goals from the VCs and these goals are proof planned.
- **Information in the world -** During proof planning the planner makes use of external information such as heuristics, rewrite rules, source code, off the shelf tools, to advance the proof of the VCs.

This long standing view is looking increasingly flawed. Invariant discovery is a key step of the proof planner to complete the proof. In the above view the proof planner would need to be executed at least twice if invariant discovery is required. The first phase would discover an invariant, revised VCs are generated with this new invariant in place, and then the proof of the revised VCs are completed.

These separate phases should be viewed as separate phases of proof planning, rather than executing the proof planner multiple times. This is an important distinction as it means that the top level proof planner does not operate on the VCs, but rather the code and specification from which the VCs are generated.

This leads to the stronger observation that NuSPADE does not just need access to the source to aid heuristics, NuSPADE is a program prover that takes solely SPARK source code, and its embedded specification, as input. NuSPADE controls the generation of VCs (using the EXAMINER) patching the code with discovered invariants and completing proofs. NuSPADE, intends to proof *programs*, unlike the SIMPLIFIER and SPADE-PC which intends to proof only *vcs*.

In practise the mechanical processes of adding discovered invariants to the source code and generating VCs will be still be done by hand in early versions of NuSPADE. However, now these steps occur during the planning process, rather than in between separate planning attempts.

5 Conclusions

To perform program analysis the source code is needed and to access this source code it is most appropriate to build a traditional program parser rather than try to build upon the facilities of the EXAMINER. To emphasis the progression of NuSPADE from a VC prover to a program prover its presentation has been revised, specifying that its input is source code rather than VCs. In practise this makes very little change to the mechanics of NuSPADE. However it does give a more accurate description of the intended behaviour of the system.

5.1 Recent progress

Since the initial early draft of this note, CS-note 40 was produced describing how the source code would be accessed. Following this note good progress has been made toward an initial working prototype.

Unfortunately, while demonstrated to be viable, the approach outlined in CS-note 40 lacks a certain rigour[2]. Following CS-note 40 it was found that the system it describes is essentially a *program transformation* system, in both behaviour and general design. Tools already exist for this purpose [2], providing an efficient and more importantly more structured approach. The rational reconstruction of CS-note 40 will be pursued in a further note.

References

[1] Praxis Critical Systems. *Generation of VCs for SPARK Programs.* 2001.

[2] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

A Appendix (Phil Thornley Example)

A.1 Specification

```
package PhilThornleyExample
is
        -- The range of the array used to hold flags.
        subtype Index is Integer range 0 .. 9;
        type DataArray is array (Index) of Integer;

        procedure SumArray(Data_Set: in DataArray ; Result: out Integer);
        --#derives Result from Data_Set;

end PhilThornleyExample;

```

A.2 Body

```
package body PhilThornleyExample
is
    procedure SumArray(Data_Set: in DataArray ; Result: out Integer)
    is
    begin
        Result:=0;
        For I in Index loop
            if Data_Set(I) >= 0 and Data_Set(I) <= 100 Then
                Result := Result + Data_Set(I);
            end if;
        end loop;
    end SumArray;

end PhilThornleyExample;
```

[2]The intermediate grammar and the rules applied to this grammar are somewhat ad-hoc.