CS-Note* 58 Bill J. Ellis
August 18, 2003

Visiting Praxis: Into the deep

1 Introduction

Following the plan presented in CS-note 57 a number of tasks are performed, exploiting the availability of high integrity software at Praxis Critical Systems.

2 A superficial appraisal

The first task is an initial appraisal of the nature of high integrity software.

2.1 General observations

2.1.1 Loops -

- **Frequency -** Are fairly common, however, straight through code is most frequent.
- **Nature -** Kinds of loop operations:
  - **Math based -** A few key math routines are implemented in SPARK, such as cosine or square root. Common operations, such as multiplication and division, are left to the compiler and the resulting proof hole is filled with a proof function and associated proof rule. Such math operations are infrequent. These are built using general loops. Note that look up tables are sometimes employed rather than implement a subprogram.
  - **Initialisation -** A few loops are used for initialisation, apparently in favour of aggregates. These are for loops, with generally simple code bodies (no conditional statements, constant assignments). However, the nature of the structures may be complex, such as arrays of records. This is probably why loops have been chosen over aggregates.
  - **Search -** Search operations, a conditional assignment, within for statements. Where searching in a known type a for loop is used, exiting early if the value is found.
  - **Calculate -** Loops are also used for iterating over arrays of values. Where connecting with the outside world these arrays may be outside the system and accessed indirectly via a subprogram call. These calculation loops take various different forms.
- **Counters -** Where a for loop is used the constraint, inside type or specified range, is the bound on the counter. For non for loops, counter like variables are often present, but not always.
- **For loop ranges -** The range is specified to be constrained within a type or between two expressions, possibly constraining variables and explicit constants (1,2...).

2.1.2 Data structures -

- **Based types -** At low level code, based types are used.
- **Fixed point -** Uses fixed point over floating point, for proof reasons. (Fixed point arithmetic is ultimately tied to an integer type)
- **Records -** Are very common.
- **Array -** Arrays and arrays of arrays of a type, including record are common. No two or higher dimensional arrays.

*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).

2.1.3 Conditions -

- **Nature -** Kinds of conditions:
  - **Case when -** Nearly always the when condition of case statements is restricted to simple values of some type. Where an expression, the expression is not complex, for example just addition or inside a range. The switch itself may however be on a complex expression.
  - **With others -** Others is used occasionally. Cases are always complete.
  - **If common -** If statements about 5 times more common than case, and are very frequent.
  - **Complexity -** Complexity ranges from a trivial test to a many line evaluation incorporating function calls and large data structures.

2.1.4 Functions -

- **Postconditions -** Are supplied for some functions. May be essentially a repetition of the logic of the function, in straight through code cases.
- **Big return statements -** Functions sometimes have quite large return statements. These could probably be implemented a little cleaner with a result variable and some conditional statements. However, the large return may be easier to prove.

2.1.5 About the code -

- **Is interconnected -** The code is distributed into several packages in a clean manner. The result is that to correctly interpret any particular file may require interrogating other files.
- **Pragma is used -** The following pragmas are seen. Note that none of these pragmas directly change the code semantics.
  - **Inline -** These are very common. They force the compiler to compile sections of code inline.
  - **Pack -** Encourage the compiler to favour storage minimisation over other factors such as speed.
  - **Interface -** Establish interface to other languages.
  - **Interface_Name -** Same as the above, kept for compatibility with Ada 83 compilers.
  - **no_image -** Instructs the compiler to avoid the construction of strings for aggregates of the related component.
- **Many asserts -** It is fairly common to see more than the minimum necessary assert statements. These cut the path generating more VCs. However, in cutting the path the VCs tend to be smaller and more susceptible to human and computer analysis.

  A side effect of this additional cutting is the need to propagate information that would naturally be present if the minimum number of assertions were made. For example, in cutting a loop twice, the VC for the path between these assertions will not include any external information.

  While this is not logically needed for proof, it may be practically necessary to partition the problem into manageable sections.
- **Type -** Type is as constrained as possible, with many types and subtypes for the various different concepts. The bounds on the types are static, by definition, but may be a fairly complex expression.

  Note that delta is used. This specifies the discrete change for fixed point arithmetic.

  The types naturally leak out from the specification. A logical specification might impose various states and properties which are difficult to translate into the code. However, type is generally directly transferable from the specification to the code.
- **Hide -** Hide is used for interface and low level tasks. It is present.

2.2 Overview

High integrity software is just well written regular software. For a large system most aspects of SPARK can be expected to be seen, used appropriately. Thus the problems of proving high integrity software are as expected:

- **Assertion discovery -** Discovering properties that assist in the proof.
- **Rule discovery -** Discovering rules to assist in proofs. This is quite rare, but may be required on occasion, especially where performing partial correctness proofs.
- **Proof discovery -** Discovering proofs, exploiting any inserted properties.

With a general problem of:

- **Scale -** The software is separated into separate isolated subprograms. However, at higher levels in the code these subprograms are called from different parts of the code, creating an ever more complex structure.

Here the first two problems will be considered then examples will be developed to explore the practical problems presented by proof in the large.

3 Property discovery

3.1 Some rough metrics

The total package body code contains 74888 lines, of which 2901 are asserts, and 15871 are derives. The total code produces 9361 VCs.

It appears that the assertions inserted are often fiddly but rarely sophisticated. There are very few assertions that involve multiplication and no explicit assertions involve quantification[1].

Although property discovery is required, and is key for proving the VCs, a single property may be sufficient for proving a class of VCs. Thus users will spend the majority of their time tackling VCs rather than discovering assertions. This is because it is common for a subprogram to require a single piece of assertion discovery, often no more complicated than asserting type information, followed by several VC proofs, each requiring a separate proof session inside the SPADE-PC.

4 Property discovery examples

4.1 Dual counter

Short - This shows a loop terminating when one of two counters reaches a value.

Proof - Requires the discovery that the two counters are within their types.

4.1.1 Specification

```
package Example1
is
    subtype A_Type is Integer range 0 .. 2;
    subtype B_Type is Integer range 1 .. 1000;
    type C_Type is array (B_Type) of Integer;

    procedure EX1(A: out A_Type ;
                  B: out B_Type ;
                  C: in C_Type);
    --#derives A from C &
    --#        B from C;

    function LittleLookUp(Value: in Integer) return Boolean;

end Example1;
```

[1]Early versions of SPARK did not support quantified assertions directly within the source. Instead these would be introduced indirectly through declared proof functions. Declared proof functions are quite rare and tend to appear only where partial correctness proofs are being performed.

4.1.2 Body

```
package body Example1 is

    function LittleLookUp(Value: in Integer) return Boolean
    is
        Result: Boolean;
    begin
        case Value is
           when 1 => Result:=True;
           when 20 => Result:=True;
           when 30 => Result:=True;
           when 40 => Result:=True;
           when others => Result:=False;
        end case;
        return Result;
    end LittleLookUp;

    procedure EX1(A: out A_Type ;
                  B: out B_Type ;
                  C: in C_Type)
    is
        I: A_Type;
        J: B_Type;
    begin
        I:=0;
        J:=B_Type'First;

        loop
            if LittleLookUp(C(J)) then
                I:=I+1;
                B:=J;
            end if;

            exit when (J=B_Type'Last or
                       I=A_Type'Last);
            J:=J+1;
        end loop;
        A:=I;
    end EX1;
end Example1;
```

4.2 Nested loop

Short - This shows a loop within a loop to write values to an essentially two dimensional data structure.

Proof - Requires the discovery that the two loop counters are within their types. Further, the additional counter across the two loops must be constrained in relation to the loop counters.

This is perhaps the most complicated proof discovery expression, yet is also quite straight forward. It seems that a quite unsophisticated implementation of the recurrence relation analysis would be able to capture these properties. The complexities appear to lie in scale and in the eventual proof.

4.2.1 Specification

```
package Example2
--#own GlobalX, GlobalY;
is
    subtype C_Type is Integer range 0 .. 8192;
    subtype Outer_Type is Integer range 0 .. 7;
    subtype Inner_Type is Integer range 1 .. 1024;
    type GlobalY_Type is array (Inner_Type) of Integer;

    GlobalX: Integer;
    GlobalY: GlobalY_Type;

    procedure WriteToGlobalX(Value: in Integer);
    --#global out GlobalX;
    --#derives GlobalX from Value;

    procedure EX2;
    --#global in out GlobalY;
    --#       out GlobalX;
```

```
    --#derives GlobalY from GlobalY &
    --#        GlobalX from ;

end Example2;
```

4.2.2 Body

```
package body Example2 is

    procedure WriteToGlobalX(Value: in Integer)
    is
    begin
        GlobalX:=Value;
    end WriteToGlobalX;

    procedure EX2
    is
        C: C_Type;
    begin
        C:=0;
        WriteToGlobalX(Outer_Type'First);
        for I in Outer_Type loop
            WriteToGlobalX(I);
            for J in reverse Inner_Type loop
                GlobalY(J):=C;
                C:=C+1;
            end loop;
        end loop;
    end EX2;

end Example2;
```

4.3 Phil Pattern

Short - This shows the Phil loop pattern encountered in another system.

Proof - This requires the discovery that the loop counter is within its type and that the general counter is within specific bounds. The additional processing following the loop and multitude of types make this an interesting and flexible example.

4.3.1 Specification

```
package Example3
--#own GlobalX, GlobalY;
is
    subtype Inner_Type is Integer range 1 .. 1024;
    subtype Divide_Type is Integer range 0 .. 10;
    subtype Test_Type is Integer range 0 .. 100;
    subtype GlobalXContents_Type is Integer range 0 .. 100;
    subtype GlobalYContents_Type is Integer range 50 .. 110;

    type GlobalX_Type is array (Inner_Type) of GlobalXContents_Type;
    type GlobalY_Type is array (Inner_Type) of GlobalYContents_Type;

    GlobalX: GlobalX_Type;
    GlobalY: GlobalX_Type;

    function EX3 return Test_Type;
    --#global in GlobalX, GlobalY;

end Example3;
```

4.3.2 Body

```
package body Example3 is

    function EX3 return Test_Type
    is
        P: Integer;
        Q: Integer;
        R: Test_Type;
    begin
```

```
        P:=0;
        --Count up.
        for I in Inner_Type loop
            P:=P+(GlobalX(I)*GlobalY(I));
        end loop;

        --Adjust.
        Q:=P/Divide_Type'Last;

        --Clip.
        if Q>Test_Type'Last then
            R:=Test_Type'Last;
        elsif Q<Test_Type'First then
            R:=Test_Type'First;
        else
            R:=Test_Type(Q);
        end if;

        --Result.
        return R;
    end EX3;

end Example3;
```

4.4 Some Structure

Short - This shows a small example introducing some of the structural complexities of realistic high integrity code. Note how key operations are packaged as separate functions. Further note that it is now quite difficult to work out what a section of code is doing.

Proof - This requires the discovery of a suitable constraint on the total counter and index counter. This is quite hard, as the key details are hidden in other functions. One trick would be to recursively establish post conditions for the functions and propagate this information to the calling point. While this would probably be required for partial correctness, it may be sufficient for exception freedom to just use the type of returned values. This would appear to be especially applicable for high integrity software where types are highly constrained.

4.4.1 Specification

```
package Example4
is

    subtype Sequence_Type is Integer range 0 .. (10*2 -1);
    subtype Number_Type is Integer range 0 .. (10*2);
    subtype Value_Type is Integer range -1000 .. 1000;
    subtype EX4Result_Type is Integer range -1000 .. 1000;
    type Inside_Type is array (Sequence_Type) of Value_Type;

    type EX4_Input_Type is
        record
            Values: Inside_Type;
            Number: Number_Type;
            --Some extra content.
            A: Sequence_Type;
            B: Sequence_Type;
            C: Sequence_Type;
            D: Sequence_Type;
            E: Sequence_Type;
        end record;

    function SkipAround(Now: Sequence_Type) return Sequence_Type;
    function Adjust(Value: Integer) return Value_Type;
    function Subtract(Left, Right: Value_Type) return Value_Type;
    function Add(Left, Right: Value_Type) return Value_Type;
    function EX4(Input: EX4_Input_Type) return EX4Result_Type;

end Example4;
```

4.4.2 Body

```
package body Example4 is

    function SkipAround(Now: Sequence_Type) return Sequence_Type
    is
    begin
        return (Now+1) mod (Sequence_Type'Last+1);
    end SkipAround;

    function Adjust(Value: Integer) return Value_Type
    is
        Result: Value_Type;
    begin
        if (Value > Value_Type'Last) then
            Result:=Value_Type(Value-3600);
        else
            if (Value < Value_Type'First) then
                Result:=Value_Type(Value+3600);
            else
                Result:=Value_Type(Value);
            end if;
        end if;
        return Result;
    end Adjust;

    function Subtract(Left, Right: Value_Type) return Value_Type
    is
    begin
        return Adjust(Left-Right);
    end Subtract;

    function Add(Left, Right: Value_Type) return Value_Type
    is
    begin
        return Left+Right;
    end Add;


    function EX4(Input: EX4_Input_Type) return EX4Result_Type
    is
        I: Sequence_Type;
        S: Integer;
        WD0: Value_Type;
        R: Value_Type;
        Total: Integer;
    begin
        I:=Input.A;
        S:=Input.Number;
        Total:=0;

        WD0:=Input.Values(I);
        loop
            Total:=Total+Subtract(Input.Values(I), WD0);
            exit when I=Input.B;
            I:=SkipAround(I);
        end loop;

        --Result.
        R:=Add(WD0, (Total / S));
        return R;
    end EX4;

end Example4;
```

5 Looking at VCs

5.1 Throw away the noise!

It is accepted that large software systems will naturally produce larger VCs. However, the size of the VCs in SPARK tends to explode as properties are introduced for all variables following all assertions. The aim is to throw as much information at the SIMPLIFIER as possible in the hope that it will automatically find a solution.

Proof planning offers the ability for a more focused search. However, at a quite unsophisticated level, it would be possible to throw away many hypotheses when looking at a particular conclusion. A possible suggestion is a recursive search for symbols in the conclusions in the hypotheses. As hypotheses are found, their symbols are added to the search. This may find more hypotheses than are needed, but should only throw away that which is truly redundant.

This may be computationally quite expensive, and might be better implemented alongside the smart VCs rather than within the proof planner[2].

5.2 Composites for interface

Composites are used implement interfaces to hardware. The details of the interface are released to the system via proof rules rather than the usual use of assertions. The SIMPLIFIER is weaker at working with proof rules, presenting tension between choosing quantified formulae or a large number of ground instances. The proof planners flexibility in using general quantified formulae may be of some use in this area.

6 Some proof discovery examples

6.1 Some arithmetic

Short - Arithmetic is common place, often generating VCs that are automatically proved by the SIMPLIFIER. However, if using div or mod the SIMPLIFIER tends to have a little more trouble. This small example gives the SIMPLIFIER quite some problems.

Proof - This appears to be classic proof planning. The isolate tactic and reasonable selection of rewrite rules should be sufficient to proves these.

6.1.1 Specification

```
package Example1
is
    type Angle is range 0 .. 359;
    function EX1(TheStart: Angle ; Wide: Angle) return Angle;

end Example1;
```

6.1.2 Body

```
package body Example1 is

    function EX1(TheStart: Angle ;
                 Wide: Angle) return Angle
    is
        R: Angle;
    begin
        if(((Integer(Wide)/2)+Integer(TheStart))>359) then
            R:=Angle(((Integer(Wide)/2)+Integer(TheStart))-360);
        else
            R:=TheStart+(Wide/2);
        end if;
        return R;
    end EX1;
end Example1;
```

6.2 More arithmetic

Short - Some more arithmetic. While pure arithmetic examples like this are not common, little snippets of arithmetic do occur across most subprograms, often of a similarly complex nature. Again the SIMPLIFIER has trouble with this example, probably due to the mod and abs.

Proof - This appears to be classic proof planning.

[2]Andrew compares these classifications to the discovery of wave front annotation. On reflection, this is a much cleaner analogy, keeping the planning components within the proof planner. Further, if planning were to fail on the constrained set of hypotheses, it would have still have access to the entire collection of hypotheses if a deeper search was desired.

6.2.1 Specification

```
package Example2
is
    type Integer32 is range -2**31 .. (2**31)-1;
    subtype Positive32 is Integer32 range 1 .. Integer32'Last;
    function EX2(N: Integer32 ;
                 Res: Positive32) return Integer32;
end Example2;
```

6.2.2 Body

```
package body Example2 is

    function EX2(N: Integer32 ;
                 Res: Positive32) return Integer32
    is
        R: Integer32;
    begin
        R:=((N/Res)*Res);
        if((abs(N) mod Res) >= (Res/2)) then
            if N>=0 then
                R:=R+Res;
            else
                R:=R-Res;
            end if;
        end if;
        return R;
    end EX2;
end Example2;
```

6.3 Even more arithmetic

Short - A tiny function including a single expression creating some more fiddly arithmetic VCs.

Proof - Involves rewriting to a common form and exploiting numeric rules.

6.3.1 Specification

```
package Example3
is
    subtype TwoK is Integer range 0 .. 2048;
    function EX3(S: TwoK) return TwoK;
end Example3;
```

6.3.2 Body

```
package body Example3 is

    function EX3(S: TwoK) return TwoK
    is
        R: TwoK;
    begin
        R:=(S / 2) + (S rem 2);
        return R;
    end EX3;
end Example3;
```