Bill J. Ellis
February 19, 2003

Generating Patterns for Constant Path Analysis (CPA)

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 CS-note 34 a new technique for enhancing the capabilities of a proof planner is described. The technique will be referred to as *constant path analysis* (CPA) within this note. This note serves to explore CPA and focus on the particular step of generating a collection of patterns of *constant path descriptions* (CPDS) from the code.

1.1 Outline CPA

CPA is a bottom-up technique, gaining insights for proof from the code. It was originally intended for code which contains loops, however it may also be applicable for straight through code[1]. The process of CPA can broken down into several steps:

1. **Identify the problem -** Every VC that is not fully discharged by the SIMPLIFIER is a candidate problem.
2. **Elaborate the problem -** Perform some initial analysis of the problem to identify the key features and variables that are preventing proof.
3. **Generate CPDS -** Based on the elaborated problem the code is processed to generate CPDS. This involves tracing the actual values of variables as the program is executed. A different CPD will be generated for different paths through the code. The technique is presented in CS-note 34 §4 and is elaborated here in §5. The key feature of this operation is considering only a small finite number of iterations of any loops.
4. **Forward inference -** After the generation of CPDS forward inferences are performed, guided by the key features and variables identified while elaborating the problem. Note that in CS-note 34 this step was interleaved with step 3. It is helpful to consider these different processes separately.
5. **Find patterns in the CPDS -** The generated CPDS are analysed. The intention is to find patterns that give strong indications of the behaviour of the code.
6. **Propose candidate generalisations -** Using the patterns, candidate generalisations are generated. These should be transferable to the appropriate SPARK annotations.

1.2 CPA in context

CPA combines two previous themes of using actual values and information extracted from the source. Thus the following brief descriptions can be made:

- **Target boundary conditions -** Based on heuristics derived from the testing community, programming errors are more likely to occur at boundary conditions. This heuristic can be exploited in the search for non-theorems by targeting VCs with actual values that correspond to the boundary conditions.
- **Code characteristics -** Through analysing code characteristics small pieces of information can be induced that improve the chances of finding a proof. Such information includes knowing that a variable is always monotonically increasing or decreasing, the types of variables and the category of the algorithm under consideration.
- CPA **-** The program is analysed by tracing the actual values of variables. These traces form patterns which are analysed to gain insights about the characteristics of the code.

1.3 Scope of this note

Step 1 is already provided through the EXAMINER and SIMPLIFIER. However all of the remaining steps will require further development, potentially including some fairly sophisticated reasoning.

Within this note all emphasis will be placed on exploring step 3. This step is central to the technique and algorithmic, making it a good starting point. The elaboration of the problem performed in step 2 will be ignored for the moment. Instead every detail of analysed programs will be traced.

[1] The technique may aid in the generation of postconditions and preconditions. This is suitable for both looping and straight through code.

2 Formal program analysis

Programs can be analysed using weakest preconditions or strongest postconditions. It is common to prove programs using weakest preconditions as it is generally the case that postconditions are much stronger than the preconditons, aiding analysis in this direction. Additionally the assignment rule for weakest preconditons is slightly more straight forward than for strongest postconditions.

A result of this choice is that the program is analysed in reverse, against the flow of control. In CPA the program is analysed with the flow of control. To analysis the program in this direction weakest preconditions are not suitable and instead strongest postconditions must be used.

2.1 Formal definitions

Note that $P[E/V]$ is the result of changing all occurrences of V to E inside P. Further $\$a$ denotes a new unique history variable a.

2.1.1 Weakest precondition

The weakest precondition wp(**Statement**, **Postcondition**) is the precondition defining the largest set of legal input states such that the **Postcondition** is guaranteed to hold after executing **Statement**. The assignment rule for weakest preconditions is (1) and a small example using this rule is given in Figure 1.

$$\vdash \quad \underbrace{\{P[E/V]\}}_{\mathrm{wp}(\textbf{Statement},\, \textbf{Postcondition})} \quad \underbrace{V := E}_{\textbf{Statement}} \quad \underbrace{\{P\}}_{\textbf{Postcondition}} \tag{1}$$

Before	After

`{}`
`I:=I+1;`
`{integer(I)}`
	`{integer(I+1)}`
	`I:=I+1;`
	`{integer(I)}`

Figure 1: Application of weakest precondition assignment rule

2.1.2 Strongest postcondition

The strongest postcondition sp(**Precondition**, **Statement**) is the postcondition defining the smallest set of output states that are guaranteed to hold after executing **Statement** given **Precondition**. The assignment rule for strongest postcondition is (2) and a small example using this rule is given in Figure 2.

$$\vdash \quad \underbrace{\{P\}}_{\textbf{Precondition}} \quad \underbrace{V := E}_{\textbf{Statement}} \quad \underbrace{\{\$a.P[a/V] \wedge V = E[a/V]\}}_{\mathrm{sp}(\textbf{Precondition},\, \textbf{Statement})} \tag{2}$$

Before	After

`{integer(I)}`
`I:=I+1;`
`{}`
	`{integer(I)}`
	`I:=I+1;`
	`{integer(a) /\ I=a+1}`

Figure 2: Application of strongest postcondition assignment rule

This approach describes the theory of the strongest postconditions. However, it is possible to refine the implementation to record the history of variables. This is the approach taken for CPA. A small example illustrating history variables with propagation is given in Figure 3. Note that this approach is formalised in [3]. An alternative discussion with some helpful examples are in [5].

Before	After

`{integer(I_0)}`
`I:=I+1;`
`{}`
	`{integer(I_0)}`
	`I:=I+1;`
	`{integer(I_0) /\ I_1=I_0+1}`

Figure 3: Application of strongest postcondition assignment rule with history variables

3 Loop standard form

All loops in SPARK are ultimately treated as the same standard loop as shown in 4. In [1] the mapping from different kinds of loops to their standard form is given. This is repeated here in Figure 5.

```
loop
    Optionally_Some_Statements;
    exit when Condition;
    Optionally_More_Statements;
end loop;
```

Figure 4: Loop standard form in SPARK

4 Unravelling loops

Generating CPDS involves unravelling a few iterations from the first entry of a loop, allowing these iterations to be explored separately.

4.1 Unravelling loops in other fields

Unravelling loops has been pursued in other fields for reasons quite different to CPA.

- **Program slicing -** Program slicing was introduced by Weiser [8, 9]. The idea is to capture in a program slice the key details of a point of interest, or slicing criteria. Typically this slicing criteria will specify a line-number and variable. The original motivations for program slicing were to assist in debugging large programs. By applying program slicing on a troublesome variable a programmer can automatically generate information about this variable.

  A number of different kinds of program slices have been explored. Weiser defined a program slice S as reduced executable program obtained from a program P by removing statements such that S replicates part of the behaviour of P. There also exists forms of program slices that cannot be executed. Further, there is a distinction between static slices that make no assumptions on the the values of input variables and dynamic slices that rely on a specific set of input values.

- **Loop unrolling for efficiency -** Loops in high level programming languages can be compiled into more time efficient code by unravelling iterations. The disadvantage is that the unravelled code takes up more space. For example, some `for` loops can be completely replaced by a series of sequential statements. Some effort has also been placed toward unrolling unbounded loops [6].

- **Information flow analysis -** The information flow analysis adopted within SPARK based on [2] involves a kind of loop unravelling to determine variable associations.

4.2 Avoid code level transformations

Every loop can be modelled using an infinite number of `if` statements. Once this infinite structure is identified, a small finite segment from first entering the loop can be constructed. An apparently correct transformation to strip off a single iteration is described in Figure 6.

Original form	Normal form

`while E loop`
`    S;`
`end loop;`
	`loop`
	`    exit when not E;`
	`    S;`
	`end loop;`

`for I in T range L..U loop`
`    S;`
`end loop;`
	`if L<= U then`
	`    I:=L;`
	`    loop`
	`        S;`
	`        exit when I=U;`
	`        I:=T'Succ(I);`
	`    end loop;`
	`end if;`

`for I in T loop`
`    S;`
`end loop;`
	`I:=T'First;`
	`loop`
	`    S;`
	`    exit when I=T'Last;`
	`    I:=T'Succ(I);`
	`end loop;`

`for I in reverse T range L..U loop`
`    S;`
`end loop;`
	`if L<=U then`
	`    I:=U;`
	`    loop`
	`        S;`
	`        exit when I=L;`
	`        I:=T'Pred(I);`
	`    end loop;`
	`end if;`

`for I in reverse T loop`
`    S;`
`end loop;`
	`I:=T'Last;`
	`loop`
	`    S;`
	`    exit when I=T'First;`
	`    I:=T'Pred(I);`
	`end loop;`

Figure 5: Mapping into SPARK normal form loops

Original form	Strip off iteration

`loop`
`    Some_Statements;`
`    exit when Condition;`
`    More_Statements;`
`end loop;`
	`Some_Statements;`
	`if not (Condition) then`
	`    More_Statements;`
	`    loop`
	`        Some_Statements;`
	`        exit when Condition;`
	`        More_Statements;`
	`    end loop;`
	`end if;`

Figure 6: Strip off one loop iteration (flawed)

However this approach is not applicable in general as it is possible and reasonable for loops to have more than one exit. An example of such a loop is in Figure 7, taken from [7]. This loop has two exits, one for where the target has been found and another for where the last element has been reached.

```
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;
```

Figure 7: For loop with two exits

In [4] methods are presented for eliminating all goto statements in programs to aid the design of optimising and parallelling compilers. It accepts that good practise has eliminated many goto statements, but that these still remain for special cases such as error handling or as sanitised gotos for exiting directly out of loops. A method is presented for dealing with `exit`[2] statements within loops, operating by adding additional variables and testing these accordingly. These methods do replace all goto statements with structured control flow. However, the resulting code bears increasingly little resemblance to the original. This is not suitable for CPA as the CPDS must reflect the original code for the patterns generated and their subsequent generalisations to be useful.

It is accepted that the unravelling of programs can not effectively occur at the code level in the general case. Instead, the code will be loaded into a separate structure and suitably processed. Note that the transformation of code loops into a normal form remains an important transformation. However this may also be performed through means other than transforming the source code.

5 About CPDS

In CS-note 34 §4 there is a notion of generating proofs $P(0), P(1), \ldots, P(n-1)$ where $P(X)$ is the proof at iteration X. Each proof is a conjunction of statements describing what would be true if the iteration were reached. Showing that these reduce to false is sufficient to show that the iteration is never reached. There are a number of subtleties to this that remain to be considered.

5.1 Extent of program unit

In CS-note 34 a loop is targeted, taking account of some preceding code. It is not clear how far back the analysis should progress. It is expected that it will rarely be desired to explore beyond a loops subprogram in generating the CPDS. Further subprograms are well defined units in SPARK. It is reasonable to make subprograms the units that are analysed by CPA. Note that it will remain possible for CPA to lead to requests for CPA on further subprograms. The restriction is that any single application of CPA will be restricted to a single subprogram.

5.2 Static or dynamic

Similar to program slicing, it is possible to envisage both static and dynamic implementations of CPA. In CS-note 34 a static approach is taken, discussing the input variables by name rather than assigning them particular values.

It might be possible to pursue a dynamic approach, constructing test case scenarios which result in the required paths being taken. However, the static approach is more general and more fitting with the intended purpose of pattern identification and generalisation. In avoiding the dynamic approach little valuable information can be gained from actually compiling and executing the subprogram being analysed[3].

[2] The paper is targeted at C and so actually considers `break` statements.

[3] However, as considered in §5.4, information may be gained by executing subprograms called from within the subprogram being

5.3 Selecting variables

The CPA presented in CS-note 34 records history values for the identified key variable `Result`. Where no key features are identified every variable might be traced.

It was initially thought that only local variables should be considered. Heuristically, invariants tend to include the local variables, which deal with low level implementation details of the computation being performed. However, although the local variables are expected to play important roles in any invariants, all variables are potentially significant.

5.4 Calling subprograms

Their are two different kinds of subprograms in SPARK:

- **Function -** Functions return a value. Where already supporting statements functions do not require any additional consideration.
- **Procedure -** In **??** §2.1.1 it is described how procedures are treated in generating SPARK VCs. Essentially a procedure hides a block of calculations. The procedure takes a collection of variables and assigns some of these new values when the procedure finishes. Thus a procedure can be modelled as a series of assignment statements to new values.

At the point at which a procedure is called there will be a number of assignments. For example in Figure 8 the call `Swap(A,B);` can be thought of as `Swap(A_Old -> A_New, B_Old -> B_New);`. This is a convenient way to express procedure calls within proof statements, replacing `New` and `Old` with the suitable history variable subscripts.

```
procedure Swap(X, Y: in out Integer)
is
    Temp: Integer;
begin
    Temp:=X;
    X:=Y;
    Y:=Temp;
end Swap;
```

Figure 8: Swap procedure

Note that this notation does not give any details on the `New` variables. Further, this is not directly suitable as part of CPD as it is not a Boolean expression. All occurrences of procedure calls must be interpreted in some manner to generate a genuine CPD. A few options are available:

- **Execute the code -** Where all of the input arguments to the procedure are known the procedure can be executed to gain the actual values of the new variables.
- CPA **again -** Where some of the values are unknown the behaviour of the procedure can not be determined. It might be possible to run CPA again specifying all of the known values for the initial state. However, there will potentially be many paths through the procedure to consider. Further, if the procedure contains a loop CPA may be poor at finding the final values. In principle running CPA again is viable but the results would likely be too unconstrained to be useful.
- **Give up -** Accept that the values are too difficult to calculate after calling the procedure. Some variables will become unknown after calling the procedure.

5.5 Working with arrays

Arrays are treated as single units within SPARK information flow analysis probably as it is too computationally expensive to trace every index of the array. Such an approach would result in the loss of valuable information in CPA. Instead the values assigned to each array index will be considered. Two approaches are considered:

analysed.

- **Treat as variables -** Treat each element of an array as a single variable. This is unlikely to be practical where arrays are very large.
- **Quantified initialisation -** CPA traces variables, keeping track of what changes via assignment statements. Assuming a small number of iterations are explored[4] and the subprogram being considered is sensibly sized only a small number of variables will be actually be referenced. For example, consider a bubble sort that takes as input a 100000 element array. Each iteration will result in a comparison of two elements and a possible swap of these elements. If unwrapping 4 iterations this results in only $4 * 2 = 8$ array index elements being referenced. The only point where a large amount of the array is referenced is when the entire array is initialised. By supporting some means to say (3) rather than (4) the number of stored entries can be drastically reduced. In the short term it may be possible to experiment by naively treating each array index as a variable. However, the quantified initialisation approach is preferable.

$$\forall i(i \geq 1 \wedge i \leq 100000 \rightarrow element(array, i) = undefined) \quad (3)$$

$$\begin{array}{ll} element(array, 1) = undefined & \wedge \\ element(array, 2) = undefined & \wedge \\ \ldots & \\ element(array, 100000) = undefined & \end{array} \quad (4)$$

Figure 9: Alternative presentations of array initialisation

5.6 Many paths through subsection

The proofs listed in CS-note 34 assume that the `if` statement, shown again in Figure 10, within the loop is always entered. As the values of `Data_Set(I)` are external to the subprogram they can not be known. It might be the case that the `if` statement is not entered on any number of the unwrapped iterations. In general their will be many paths through a subprogram, which may or may not enter loops.

```
if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
    Result:= Result+Data_Set(I);
end if;
```

Figure 10: Conditional statement on input parameters

5.7 Constant paths on iterations

There can exist a number of possible paths through a subprogram. In particular where loops are unravelled it is possible to take a different path through the loop on each iteration. The number of combinations of possible paths increases exponentially with each loop iteration.

As the intention is to identify patterns between different iterations it is considered most useful to compare iterations where the same path is taken through the loop. Thus for the example in CS-note 34 there is a CPD corresponding to always entering the `if` statement and another for always not entering `if` statement. This notion of a constant path for iterations greatly reduces the number of candidate CPDS that can be generated.

Note that identifying constant paths through loops requires knowing where the loop starts. This is not necessarily the first occurrence of a loop invariant. An approach will need to be taken that ensures that the path on entering the loop is consistent with the path taking going around the loop.

To exit the loop a different path will have to be taken than that around the loop. However, it remains possible to make this path as close to the path around the loop as possible by changing the minimum necessary number of choices at conditional statements to exit the loop. This is likely to be especially straight forward to calculate given the restricted form of `exit` statements allowed in loops as prescribed by the SPARK language, as descried in [1] §10.2 and §6.3.

[4] 3 to 4 would be expected normally.

5.8 Backward CPA

It might be desired to unwrap the final last few iterations of a loop as well as its first few iterations. Unfortunately, this is not possible in general as it requires predicting the iteration at which the loop will terminate, and choosing to generate proofs for a few iterations back from this. However, it is possible to make some statements about the possible last iterations of the loop. Knowing how the loop can terminate is an important piece of additional information showing where the patterns developed in the first few iterations eventually converge, as highlighted in Figure 11[5].

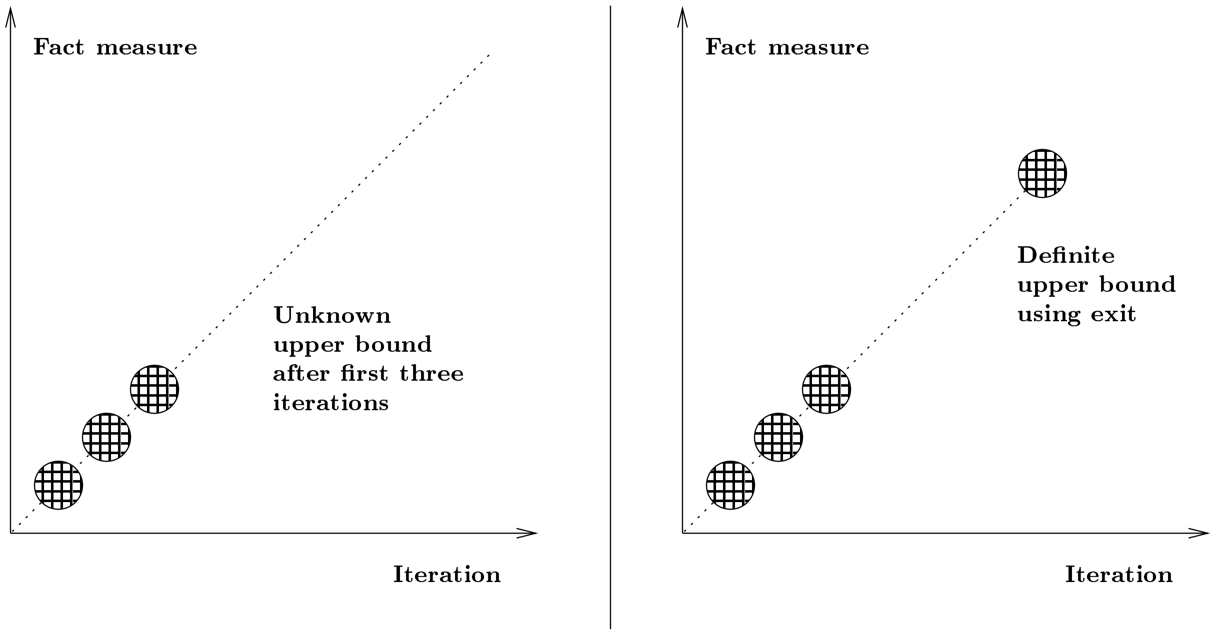

Figure 11: Using exit to close proof patterns

5.9 Forcing the paths

It may be the case that the path requested through the subprogram is not possible under all possible inputs. There exists programs where loops will always exit at a particular Iteration. Or where loops must take different paths though the loop in different iterations, for example consider the loop in Figure 12.

```
X:=1;
loop
    if even(X) then
        Statements1;
    else
        Statements2;
    end if;
    exit when X=Index'Last;
    X:=Index'Succ(I);
end loop;
```

Figure 12: Original flowchart

For such situations the proofs in CPD will contain expressions that are false. However, as CPA is static many values will be unknown. It will not be possible in general to reduce these expressions to true or false. For example,

[5]Consider the polish flag example of CS-note 10. One path will reveal that `I` increases while another will reveal that `J` decreases. However, only by considering the exit condition `I=J` is it then also known that `I<J` and that the convergence of these two variables marks the termination of the loop.

in Figure 12 it would need to be known that `even(1)` is false to determine that entering the `then` branch on the first iteration is not possible. This information can only be extracted by examining the details of the `even` function. Such an analysis might be performed within a proof planner (NuSPADE) however in general this will be quite difficult as many values may be unknown.

CPA exploits a heuristic that loops have a constant, repeated and predictable behaviour. Where this is not the case the approach will fail. However, it is expected that the number of loops in this category are small, especially in well designed high integrity software.

Thus, in all cases the paths selected will be forced to be executed. If the user requests that a loop iterates four times, even though it can only ever iterate three, then four proofs will still be generated. Some proofs describing the last iteration will be reducible to false given sufficient program knowledge.

5.10 Iteration boundary

Passing the iteration boundary presents the division between proofs. In CS-note 34 the point of the iteration boundary can be either the invariant or loop entry[6]. It is considered more appropriate to use the invariant as the iteration boundary, as this is the genuine boundary of the property being generated.

The positioning of the invariant is flexible. If an invariant already exists then its existing position should be used, otherwise it can be placed where desired. As a SPARK convention it is common to place the invariant just after the loop is entered to eliminate the path and VC for going around the loop. Note that if working with the loop normal form the location will need to be appropriately adjusted for the actual code.

5.11 Exiting loops

As subprograms are being considered, there may be statements after the first loop. To support complete subprogram analyses, variables will be traced from the beginning to the end of the subprogram. Three possible scenarios are possible:

- **Finite iterations then N more iterations then exit -** A finite number of iterations of the loop has been explored. An unknown number of iterations N may occur before the loop exiting, thus some history variables may become unknown where exiting the loop.
- **Finite iterations then exit -** A finite number of iterations of the loop has been explored and the loop is exited. In this case the history values can be carried through.
- **Direct exit -** Zero iterations of the loop have been explored. The loop has exited directly. In this case the history values can be carried through.

5.12 How to treat leaping

Where leaping[7] over N loop iterations to the exit of a loop two options are available:

- **Do not force paths -** The paths taken on these N iterations is not forced. The result is that any statement inside the loop may be performed.
- **Do force paths -** The path taken around the loop is used again as the path taken on these N iterations. Significantly, only those statements encountered on the N iterations can be executed.

Where forcing paths it is possible to leap over iterations that can not occur. As no proofs are generated for these iterations these false paths will never be identified. For example, in Figure 13 consider selecting the fixed paths as bypassing the `if` statement and exiting via the `exit when`. The false condition $not(((0+1)+1)+1=3)$ for trying to traverse the third iteration indicates the paths are impossible. However, if only considering two iterations this condition will be skipped over during the leaping and that the paths are impossible will no longer be detectable.

It is expected that this is not a particular problem. Undetectable impossible paths are a feature of CPA. To eliminate false paths directly would require tracing of every possible path. To make false paths detectable would require the ability to gain total program information by descending into further subprograms. Finally leaping would need to be suitability refined to avoid hitting the halting problem.

An advantage of forcing paths at leaping is that the number of reachable statements is reduced to those on the particular path. This allows for stronger information to be gathered on the status of variables after leaping. Not forcing paths at leaping brings little advantage where the first iteration paths are forced.

[6] These occurs at the same point in CS-note 34.

[7] A finite number of iterations of a loop will be explored to generate patterns. Between the last generated pattern and the end of the loop N loop iterations are skipped over. This skipping over loop iterations is called *leaping*.

```
Result:=0;
I:=0;
loop
    if I=3 then
        exit;
    end if;
    exit when I=TypeOfI'Last;
    I:=TypeOfI'Succ(I);
end loop;
```

Figure 13: Loop with impossible exit

5.13 Variable status after leaping (Tip of the iceberg)

Every variable must have one of the following status. These are arranged in descending order of information:

- **DefinedValue -** It is known that the variable is defined with a known particular value ($x = 7718$).
- **DefinedUnknown -** It is known that the variable is defined but the actual value of the variable is not known ($x = some\_legal\_value$).
- **Undefined -** It is known that the variable is undefined ($x = undefined$).
- **Unknown -** No information on the variable is known. It may or may not be undefined ($x = completely\_unknown$).

The more variables with higher information status the greater the application of CPA. Where leaping out of a loop the status of some variables may need to change to lower information status. However, in general, not all of the variables will be affected. For example, consider Figure 14. The `for` loop can not change the value of `Result`. After leaping out of this loop `Result` can remain **DefinedValue**. This observation can be generalised to ensure that the minimum number of variable status downgrades are performed after leaping.

```
Result:=0;
For I in Index
loop
    Data_Set(I):=0;
end loop;
```

Figure 14: Initialisation code snippet

The aim is identify all of the variables which will definitely be unchanged where following the constant path through the loop during leaping. At the point of leaping all of the other variables will have their status downgraded to **DefinedUnknown**. It may be helpful to flag these as leaped to indicate explicitly why the variable changes. Note that it might be thought that **Undefined** variables will also need downgraded. However, if they are **Undefined** they can not have been assigned to and thus, since exploring constant paths, can never be assigned to.

Unfortunately, this presents a few practical difficulties. Finding which variables will be unchanged involves finding which variables every variable is dependent on. This might be directly via an assignment or conditional statement as shown in Figure 15. This is collecting essentially the same properties as SPARK information flow analysis, only for a restricted subset of a program.

`X:=Y;`	`X` depends on [`Y`], Y depends on []
`if X=Y then Z:=10; end if;`	`X` depends on [], Y depends on [], Z depends on [`X,Y`]

Figure 15: Variable dependencies

To correctly deal with `if` statements it might be appropriate to annotate the flowcharts with labels to describe where the `if` statement ends. Arrays will require appropriate treatment, probably adopting the same array as one unit policy seen in SPARK. Subprograms should already have SPARK derives annotation, allowing direct access to the associations between the variables inside the subprogram. It should be possible to implement reasonable detection of variable status after leaping by dealing with these technical issues and copying large parts of the algorithms presented in [2]. The details may be explored later where greater knowledge is known on the entire application of CPA[8].

5.14 About patterns

In CS-note 34 as patterns are generated they are also subjected to forward inference. This forward inference has been separated out into another step to consider later. The result is that the patterns discussed in this note are more detailed and less focused than those in CS-note 34.

5.15 Pattern matching

Each alternative path through the subprogram will potentially generate different patterns. It is considered that a number of these can be selected using some heuristic and then analysed together. The combined analysis is expected to lead to stronger insights.

[8]The process is involved rather than difficult. It would be good to have a better idea on what is required before exploring this further. Their are potential advantages of more clever detection of leaped variables. Their may also alternative applications for the collection of variables found. For example, knowing that variable X is always constant in a loop means that X can be safely treated as a constant in the invariant.

6 The algorithm

6.1 Identify all variables

Every variable accessible within the subprogram is identified. The source of the variable is marked as one of the following:

- **Import -** The variable is imported to the subprogram via its parameters or as a global variable.
- **Local -** The variable is locally declared within the subprogram.

The access permissions of **Import** variables can be gained from reading the modes of the subprograms parameters and SPARK global annotations. Where the variable is **Import** the access permissions of the variables are marked as one of:

- **Read only -** The variable is only ever read[9].
- **Write only -** The variable is only ever written to.
- **Read and write -** The variable is both read and written to.

Similar to the access permissions of **Import** variables, **Local** variables have associated information:

- **Constant(Value) -** The variable is a constant with **Value**.
- **Assigned(Value) -** The variable is initialised to **Value**[10].
- **Normal -** None of the above.

A special additional variable is added for functions. The variable will be called `Function_Return`. This describes the result of the function. Thus the last line of a function `return Y;` can be interpreted as `Function_Return:=Y;`. This variable will be of source **Local** and associated information **Normal**.

6.2 Establishing the initial state

The state lists all of the variables which may be modified by the subprogram. The patterns generated for a CPD are traces of the modifications to this state. Every variable that can be written to is part of the state. This includes **Import** with access permissions **Write only** or **Read and write**, and **Local** without the associated information **Constant(Value)**. Depending on the variables source, access permissions and initialisation different initialisations are used:

- **Import -**
  - **Write only -** The variable is imported but does not provide an initial value. The variable is initially **Undefined**.
  - **Read and write -** The variable is imported and does provide an initial value. The variable is initially **DefinedUnknown**.
- **Local -**
  - **Assigned(Value) -** The variable is initialised to a known value. The variable is initially **Value**.
  - **Normal -** The variable is internally defined within the targeted subprogram. The variable is initially **Undefined**.

Note that it may also be desirable to support special initialisations of values where the actual values are known by some external means. For example, to use CPA to analyse a subprogram call within a subprogram already subjected to CPA. Further, note that all of the information gathered in this step is directly available through the SPARK syntax of the subprogram.

[9] Read is the same as in and write is the same as out.

[10] Note that a similar entity is not applicable for **Import** values as the SPARK language excludes default initialisation of parameters.

6.3 Construct the flowchart

The structure of the code is translated into a flowchart in the normal manner. Every condition is given a unique label. Every cut (`--# assert X;`) is added as special nodes. If there is a loop without a cut then one is automatically added. The flow chart is divided into a number of potentially nested sub-flowcharts:

- **Flat -** The entire flowchart is placed inside a sub-flowchart named **Flat**. There will be **Start** and **Stop** labels for entering and exiting the **Flat** sub-flowchart. Note that these labels correspond exactly to the beginning and end of the subprogram.

- **Loop_N -** All loops in the code are enclosed within `loop ...end loop;`. This block of code translates to a sub-flowchart **Loop_N**. There will be **Enter_Loop_N** and **Exit_Loop_N** labels for entering and exiting the **Loop_N** sub-flowchart, for example, **Enter_Loop_N** `loop ...end loop;` **Exit_Loop_N**. In general there need be no association between these labels and the cut points in the code. The labels are desired to describe the true scope of the loop to allow for constant pattern generation.

The nodes of every group associate themselves to their enclosing sub-flowchart. Subprograms with more than one loop will result in in increased number of sub-flowcharts. Subprograms with nested loops will result in nested sub-flowcharts. It should be possible to represent all varieties of SPARK code using this approach[11].

Note that it is possible to add multiple cutpoints to SPARK programs. This might be performed in practise to divide a long path and hence one large VC into a few smaller paths and a few smaller VCs. Such additional cutpoints outside loops should not present a problem as they can be effectively ignored in CPA. However, where there is more than one cut in a loop some action will need to be taken. The analysis is based around a single loop invariant. Treating two invariants simultaneously is not supported. Thus where are multiple loop cutpoints, each will will need to be targeted as the single invariant on successive applications of the algorithm. It is likely that the occurrence of loops with multiple cut points is very rare and thus there is little motivation to consider exploring such repeated analysis.

6.4 Finding routes

A route exists between labels where there is a least one path between these labels. For the moment logically impossible paths, such as down the `else` branch of `if true then A else B;` are still considered valid. Such paths are likely to be uncommon. Further, their identification will require proof effort which is arguably not appropriate for this stage. However, the successful pruning of even a single route would bring some large efficiency savings and may be worth pursuing in the future. Within each **Loop_N** sub-flowchart there is a maximum of two routes:

- **Around loop -** From **Enter_Loop_N** to **Enter_Loop_N**.

- **Exit loop -** From **Enter_Loop_N** to **Exit_Loop_N**.

Note that most loops will be expected to have both routes. However, loops that lack any `exit;` statement will not have a route for **Exit loop** and similarly loops that always traverse an unconditional `exit;` will not have a route for **Around loop**. Within the **Flat** sub-flowchart there is two possible routes:

- **Termination -** From **Start** to **Stop**.

- **Nontermination -** From **Start** to **InfiniteLoop** a loop which does not terminate. This is a special route in that its end is not a label. It is expected that this route will only be of use where dealing with guaranteed infinite loops as they do not have an `exit;`. These can be detected as loops which do not have an **Exit loop** route. Thus this route will only be considered where a guaranteed infinite loop has been detected.

6.5 Finding paths

For every route every possible path is found. Each path will be described as ordered truth values to describe the path to take as each condition is reached. As sub-flowcharts (loops) are crossed their name is added and their internal details are skipped over. The result is that only the paths for the inner most nested sub-flowcharts are described solely by truth values. All other paths will make reference to a sub-flowchart.

[11] It is worth noting that their will certainly be limitations in the implementation. More complicated features of SPARK such as multi dimensional array aggregates are a low priority. However, the algorithm should in theory support all SPARK features.

6.6 Finding flows

A flow is a list of truth values and special commands that can be executed to generate a pattern. An infinite number of different flows can be generated, by repeatedly increasing the number of iterations around the loop. To generate a small collection of flows the iteration depth is fixed at some value.

Flows are generated by starting with a **Termination** or **Nontermination** route. Within the list assigning truth values to conditionals there may be named loops. These loops are replaced with one of:

- **N around, leap, and exit -** Go around the loop N times, the loop continues untraced for an unknown number of iterations and then the loop exits. This involves taking N occurrences of an **Around loop** path followed by a command **Leap** and then an **Exit loop** path.
- **N around, and exit -** Go around the loop N times and then exit the loop. This involves taking N occurrences of an **Around loop** path followed by the **Exit loop** path.
- **Exit -** Do not go around the loop, and exit straight away. This involves taking an **Exit loop** path.

Note that there may be more than one **Around loop** and **Exit loop** paths. Rather than covering all possibilities some heuristics may be employed in narrowing the choices. Further note that there may be loops within loops or loops preceding loops. With each additional loop there is an exponential increase in the number of available flows. However, so long as the number of iterations N is fixed the number of flows will always be finite and for most subprograms generally small.

6.7 Generating patterns from flows

The input to the pattern generation algorithm is the initialised state, the flow to follow and the flowchart. The result of the pattern generation algorithm is the construction of proofs for each journey traversed between cutpoints.

A counter is associated with each loop and initialised to 1. A set of modifiable variable names is associated with each sub-flowchart and initialised to empty. The algorithm starts by departing from the cutpoint **Start**. For the purpose of the algorithm the cutpoint **Start** is never arrived at. This is where the algorithm begins.

6.7.1 Depart from cutpoint

At the departure from a cutpoint some actions may take place depending on the kind of cutpoint:

- **Invariant -** The departure from an invariant is recorded. As there may be multiple invariants the name of the owning sub-flowchart is recorded. Further, as the same invariant may be departed from several times the invariants counter is recorded. The counter associated with the loop that contains this invariant is increased by one. If the flow indicates **Leap** the proof state is set to new values based on the recorded collection of variable names. Otherwise the state is cleared for this journey.
- **Start -** The departure at Start is recorded. The proof state is set to the initialised state.
- **Stop -** This is not possible, as the arrival at Stop will terminate the algorithm.

6.7.2 Between cutpoints

Until the arrival at a cutpoint different actions take place for different kinds of nodes are reached in the flowchart:

- **Sequence of statements -** As each sequence of statements are encountered the proof state is amended accordingly. It may help to envisage a function attached to each sequence of statements which is provided the proof state before the sequence of statements and returns the proof state after these statements have been performed.
- **Procedure -** The procedure call is added to the proof. This will name the procedure, the appropriate history variable for each input and the appropriate history variable for each output.
- **Condition -** The next truth value is taken from the head of the flow to follow. If true the condition is added to the proof and the true branch is taken. Otherwise the negation of the condition is added to the proof and the false branch is taken.

6.7.3 Arrive at cutpoint

At the arrival at a cutpoint some actions may take place depending on the kind of cutpoint.

- **Invariant -** The arrival an invariant is recorded. As there may be multiple invariants the name of the owning sub-flowchart is recorded. Further, as the same invariant may be arrived at several times the invariants counter is recorded.

  The first time an invariant is arrived at having previously, but not necessarily just previously, departing from itself the variables with status to be downgraded are determined. This is not fully elaborated yet, however some details have been considered in §5.13.

- **Start** The arrival at Start is recorded.

- **Stop** The arrival at Stop indicates that the pattern generation is complete.

In all cases the current proof state is recorded under the unique pair (departure, arrival).

7 Phil Thornley Example

7.1 Code

Body

```
package body PhilThornleyExample
is
    procedure SumArray(Data_Set: in DataArray ; Result: out Integer)
    is
    begin
        Result:=0;
        For I in Index
        --#assert true;
        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;
```

Body in normal form

```
package body PhilThornleyExample
is
    procedure SumArray(Data_Set: in DataArray ; Result: out Integer)
    is
        I: Index;
    begin
        Result:=0;
        I:=Index'First;
        loop
            --#assert true;
            if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
                Result := Result + Data_Set(I);
            end if;
            exit when I=Index'Last;
            I:=Index'Succ(I);
        end loop;
    end SumArray;

end PhilThornleyExample;
```

7.2 Identify all variables

data_set:	**Import**	(**Read only**)
result:	**Import**	(**write only**)
i:	**Local**	

7.3 Establishing the initial state

$$result_0 = \textbf{Undefined} \quad \wedge$$
$$i_0 = \textbf{Undefined}$$

7.4 Construct the flowchart

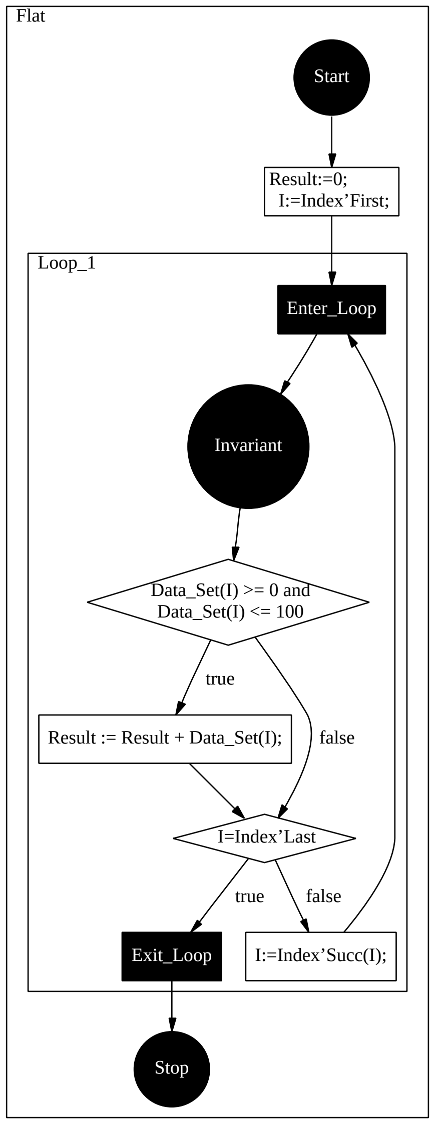

Figure 16: Flowchart

7.5 Finding routes

7.5.1 Flat

$$\textit{termination}: \quad \textbf{Start} \quad \Rightarrow \quad \textbf{Stop}$$

7.5.2 Loops

Loop_1

$$\begin{array}{llcl} \textit{around\_loop}: & \textbf{Loop\_1\_Enter} & \Rightarrow & \textbf{Loop\_1\_Enter} \\ \textit{exit\_loop}: & \textbf{Loop\_1\_Enter} & \Rightarrow & \textbf{Loop\_1\_Exit} \end{array}$$

7.6 Finding paths

7.6.1 Flat

$$\begin{array}{ll} \textit{to\_termination}: & \textbf{Start} \Rightarrow \textbf{Stop} \\ & \{[\textbf{Loop\_1}]\} \end{array}$$

7.6.2 Loops

Loop_1

$$\begin{array}{ll} \textit{around\_loop}: & \textbf{Loop\_1\_Enter} \Rightarrow \textbf{Loop\_1\_Enter} \\ & \{[true, false], \\ & [false, false]\} \\ \textit{exit\_loop}: & \textbf{Loop\_1\_Enter} \Rightarrow \textbf{Loop\_1\_Exit} \\ & \{[true, true], \\ & [false, true]\} \end{array}$$

7.7 Finding flows

Note that the number of iterations is set at three.

$$\begin{array}{ll} \textit{flow}: & \textbf{Start} \quad \Rightarrow \quad \textbf{Stop} \\ & [\textbf{Loop\_1}] \end{array}$$

7.7.1 Expanding Loop_1

N around, leap, and exit

$$\begin{array}{c} [true, false, true, false, true, false, \textbf{Leap}, true, true] \\ [true, false, true, false, true, false, \textbf{Leap}, false, true] \\ [false, false, false, false, false, false, \textbf{Leap}, true, true] \\ [false, false, false, false, false, false, \textbf{Leap}, false, true] \end{array}$$

N around, and exit

$$\begin{array}{c} [true, false, true, false, true, false, true, true] \\ [true, false, true, false, true, false, false, true] \\ [false, false, false, false, false, false, true, true] \\ [false, false, false, false, false, false, false, true] \end{array}$$

Exit

$$\begin{array}{c} [true, true] \\ [false, true] \end{array}$$

7.8 Generating patterns from flows

7.8.1 Inside the then

The flow that most closely repeats that taken in CS-note 34 is used here.

Path	$[true, false, true, false, true, false, \mathbf{Leap}, true, true]$
Depart cutpoint	**Start**
Code	Result:=0; I:=Index'First;
Fact[12]	$i_1 = Index'First \quad \wedge$ $result_1 = 0$
Arriving cutpoint	**Loop_1_Invariant(1)**
Pattern saved as	**Start ⇒ Loop_1_Invariant(1)**

Path	$[true, false, true, false, true, false, \mathbf{Leap}, true, true]$
Depart cutpoint	**Loop_1_Invariant(1)** (Increase count. Test for **Leap**. Is not found.)
Condition	Data_Set(I) >= 0 and Data_Set(I) <= 100
Pop Path	**true** $\Leftarrow [false, true, false, true, false, \mathbf{Leap}, true, true]$
Fact	$(element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100)$
Code	Result := Result + Data_Set(I);
Fact	$(element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100) \quad \wedge$ $result_2 = result_1 + (element(data\_set, i_1))$
Condition	I=Index'Last
Pop Path	**false** $\Leftarrow [true, false, true, false, \mathbf{Leap}, true, true]$
Fact	$(element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100) \quad \wedge$ $result_2 = result_1 + (element(data\_set, i_1)) \quad \wedge$ $not(i_1 = Index'last)$
Code	I:=Index'Succ(I);
Fact	$(element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100) \quad \wedge$ $result_2 = result_1 + (element(data\_set, i_1)) \quad \wedge$ $not(i_1 = Index'last) \quad \wedge$ $i_2 = i_1 + 1$
Arriving cutpoint	**Loop_1_Invariant(2)**
Pattern saved as	**Loop_1_Invariant(1) ⇒ Loop_1_Invariant(2)**
Loop_1 modifiable set	$\{result, i\}$

Path	$[true, false, true, false, \mathbf{Leap}, true, true]$
Depart cutpoint	**Loop_1_Invariant(2)** (Increase count. Test for **Leap**. Is not found.)
Skip details...	
Fact	$(element(data\_set, i_2) \geq 0) \wedge (element(data\_set, i_2) \leq 100) \quad \wedge$ $result_3 = result_2 + (element(data\_set, i_2)) \quad \wedge$ $not(i_2 = Index'last) \quad \wedge$ $i_3 = i_2 + 1$
Arriving cutpoint	**Loop_1_Invariant(3)**
Pattern saved as	**Loop_1_Invariant(2) ⇒ Loop_1_Invariant(3)**

[12]These correspond to the proofs of CS-note 34. However, they are called facts to avoid confusion with genuine proofs. Note that these are the facts of following a particular constant path. The truth of the expression of the fact, which may reduce to true or false, is a separate concern.

Path	$[true, false, \textbf{Leap}, true, true]$
Depart cutpoint	**Loop_1_Invariant(3)** (Increase count. Test for **Leap**. Is not found.)
Skip details...	
Fact	$(element(data\_set, i_3) \geq 0) \wedge (element(data\_set, i_3) \leq 100) \quad \wedge$ $result_4 = result_3 + (element(data\_set, i_3)) \quad \wedge$ $not(i_3 = Index'last) \quad \wedge$ $i_4 = i_3 + 1$
Arriving cutpoint	**Loop_1_Invariant(3)**
Pattern saved as	**Loop_1_Invariant(3)** $\Rightarrow$ **Loop_1_Invariant(4)**

Path	$[\textbf{Leap}, true, true]$
Depart cutpoint	**Loop_1_Invariant(4)** (Increase count. Test for **Leap**. Is found.)
Leap proof	$i_5 = \textbf{DefinedUnknown}_{leaped} \quad \wedge$ $result_5 = \textbf{DefinedUnknown}_{leaped}$
Skip details...	
Fact	$(element(data\_set, i_5) \geq 0) \wedge (element(data\_set, i_5) \leq 100) \quad \wedge$ $result_6 = result_5 + (element(data\_set, i_5)) \quad \wedge$ $i_5 = Index'last$
Arriving cutpoint	**Stop** (Cease algorithm)
Pattern saved as	**Loop_1_Invariant(4)** $\Rightarrow$ **Stop**

7.8.2 Outside the then

The following path is not considered in CS-note 34. Is is conjectured that by analysing this path the lower bound, `Result >= 0`, can be found.

Path	$[false, false, false, false, false, false, \textbf{Leap}, false, true]$
Depart cutpoint	**Start**
Code	Result:=0; I:=Index'First;
Fact	$i_1 = Index'First \quad \wedge$ $result_1 = 0$
Arriving cutpoint	**Loop_1_Invariant(1)**
Pattern saved as	**Start** $\Rightarrow$ **Loop_1_Invariant(1)**

Path	$[false, false, false, false, false, false, \textbf{Leap}, false, true]$
Depart cutpoint	**Loop_1_Invariant(1)** (Increase count. Test for **Leap**. Is not found.)
Condition	Data_Set(I) >= 0 and Data_Set(I) <= 100
Pop Path	**false** $\Leftarrow [false, false, false, false, false, \textbf{Leap}, true, true]$
Fact	$not((element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100))$
Condition	I=Index'Last
Pop Path	**false** $\Leftarrow [false, false, false, false, \textbf{Leap}, false, true]$
Fact	$not((element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100))$ $not(i_1 = Index'last)$
Code	I:=Index'Succ(I);
Fact	$not((element(data\_set, i_1) \geq 0) \wedge (element(data\_set, i_1) \leq 100))$ $not(i_1 = Index'last) \quad \wedge$ $i_2 = i_1 + 1$
Arriving cutpoint	**Loop_1_Invariant(2)**
Pattern saved as	**Loop_1_Invariant(1)** $\Rightarrow$ **Loop_1_Invariant(2)**
Loop_1 modifiable set	$\{i\}$

Path	$[false, false, false, false, \textbf{Leap}, false, true]$
Depart cutpoint	**Loop_1_Invariant(2)** (Increase count. Test for **Leap**. Is not found.)
Skip details...	
Fact	$not((element(data\_set, i_2) \geq 0) \wedge (element(data\_set, i_2) \leq 100)) \wedge$ $not(i_2 = Index'last)$ $i_3 = i_2 + 1$
Arriving cutpoint	**Loop_1_Invariant(3)**
Pattern saved as	**Loop_1_Invariant(2)** $\Rightarrow$ **Loop_1_Invariant(3)**

Path	$[false, false, \textbf{Leap}, false, true]$
Depart cutpoint	**Loop_1_Invariant(3)** (Increase count. Test for **Leap**. Is not found.)
Skip details...	
Fact	$not((element(data\_set, i_3) \geq 0) \wedge (element(data\_set, i_3) \leq 100)) \wedge$ $not(i_3 = Index'last)$ $i_4 = i_3 + 1$
Arriving cutpoint	**Loop_1_Invariant(3)**
Pattern saved as	**Loop_1_Invariant(3)** $\Rightarrow$ **Loop_1_Invariant(4)**

Path	$[\textbf{Leap}, false, true]$
Depart cutpoint	**Loop_1_Invariant(4)** (Increase count. Test for **Leap**. Is found.)
Leap proof	$i_5 = \textbf{DefinedUnknown}_{leaped}$
Skip details...	
Fact	$not((element(data\_set, i_5) \geq 0) \wedge (element(data\_set, i_5) \leq 100)) \wedge$ $i_5 = Index'last$
Arriving cutpoint	**Stop** (Cease algorithm)
Pattern saved as	**Loop_1_Invariant(4)** $\Rightarrow$ **Stop**

8 Search Boolean Matrix Example

8.0.3 Code

Specification

```
package Boolean2DMatrixOperations
is
    type Boolean2DMatrixIndex is range 1..4;
    type Boolean2DMatrix is array(Boolean2DMatrixIndex,
                                  Boolean2DMatrixIndex) of Boolean;
   function HasTrue(SearchMatrix: in Boolean2DMatrix) return Boolean;
end Boolean2DMatrixOperations;
```

Body

```
package body Boolean2DMatrixOperations
is
   function HasTrue(SearchMatrix: in Boolean2DMatrix) return Boolean
   is
       Result: Boolean;
   begin
       Result:=False;
       for I in Boolean2DMatrixIndex
           --#assert true;
       loop
           for J in Boolean2DMatrixIndex
           --#assert true and true;
           loop
               if SearchMatrix(I,J)=True then
                   Result:=True;
                   exit;
               end if;
           end loop;
           if Result=True then
               exit;
           end if;
       end loop;
      return Result;
   end HasTrue;
end Boolean2DMatrixOperations;
```

Body in normal form

```
package body Boolean2DMatrixOperations
is
   function HasTrue(SearchMatrix: in Boolean2DMatrix) return Boolean
   is
       Result: Boolean;
       I: Boolean2DMatrixIndex;
       J: Boolean2DMatrixIndex;
   begin
       Result:=False;
       I:=Boolean2DMatrixIndex'First;
       loop
           --#assert true;
           J:=Boolean2DMatrixIndex'First;
           loop
               --#assert true and true;
               if SearchMatrix(I,J)=True then
                   Result:=True;
```

```
                exit;
            end if;
            exit when J=Boolean2DMatrixIndex'Last;
            J:=Boolean2DMatrixIndex'Succ(J);
        end loop;
        if Result=True then
            exit;
        end if;
        exit when I=Boolean2DMatrixIndex'Last;
        I:=Boolean2DMatrixIndex'Succ(I);
    end loop;
    return Result;
  end HasTrue;
end Boolean2DMatrixOperations;
```

8.1 Identify all variables

searchmatrix:	**Import**	(**Read only**)
result:	**Local**	
i:	**Local**	
j:	**Local**	
function_return:	**Local**	

8.2 Establishing the initial state

$$\begin{aligned} result_0 &= \textbf{Undefined} && \wedge \\ i_0 &= \textbf{Undefined} && \wedge \\ j_0 &= \textbf{Undefined} && \wedge \\ function\_return_0 &= \textbf{Undefined} \end{aligned}$$

8.3 Construct the flowchart

8.4 Finding routes

8.4.1 Flat

$$termination: \quad \textbf{Start} \quad \Rightarrow \quad \textbf{Stop}$$

8.4.2 Loops

Loop_1

$$\begin{array}{llcl} around\_loop: & \textbf{Loop\_1\_Enter} & \Rightarrow & \textbf{Loop\_1\_Enter} \\ exit\_loop: & \textbf{Loop\_1\_Enter} & \Rightarrow & \textbf{Loop\_1\_Exit} \end{array}$$

Loop_2

$$\begin{array}{llcl} around\_loop: & \textbf{Loop\_2\_Enter} & \Rightarrow & \textbf{Loop\_2\_Enter} \\ exit\_loop: & \textbf{Loop\_2\_Enter} & \Rightarrow & \textbf{Loop\_2\_Exit} \end{array}$$

8.5 Finding paths

8.5.1 Flat

$$\begin{array}{l} to\_termination: \textbf{Start} \Rightarrow \textbf{Stop} \\ \qquad\qquad\qquad\quad \{[\textbf{Loop\_1}]\} \end{array}$$

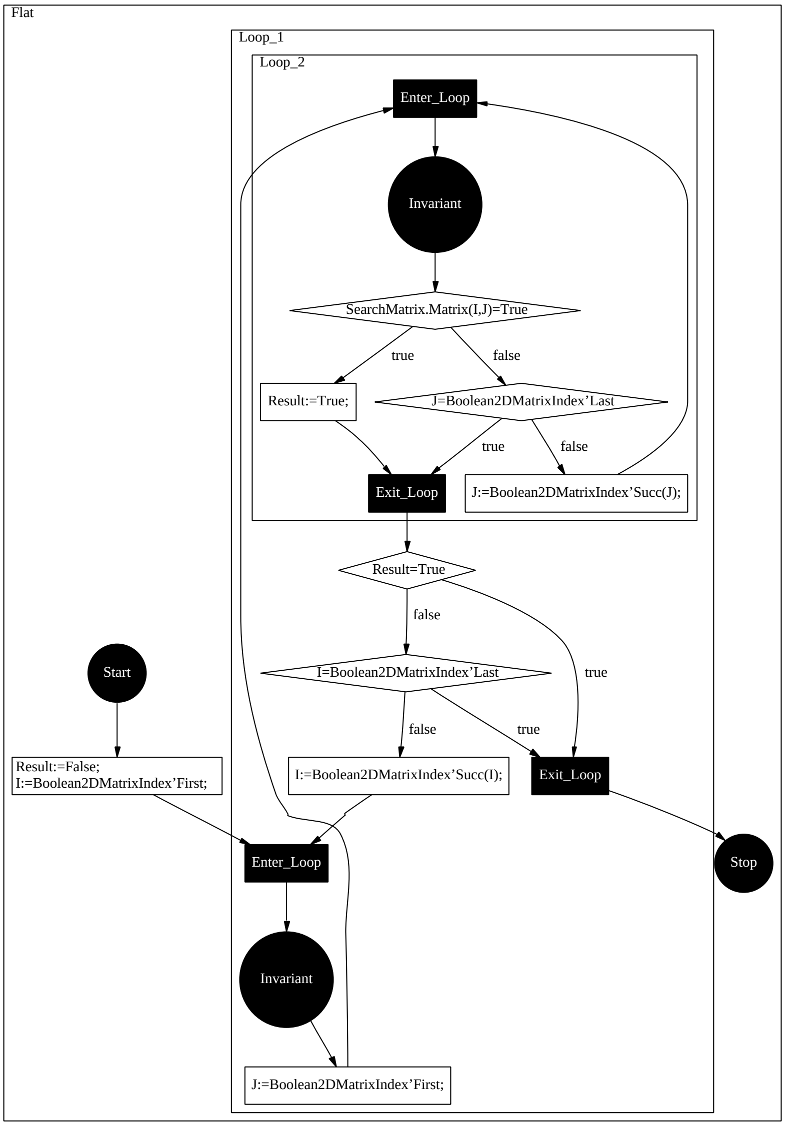

Figure 17: Flowchart

8.5.2 Loops

Loop_1

$$\begin{array}{ll} around\_loop: & \textbf{Loop\_1\_Enter} \Rightarrow \textbf{Loop\_1\_Enter} \\ & \{[\textbf{Loop\_2}, false, false]\} \\ exit\_loop: & \textbf{Loop\_1\_Enter} \Rightarrow \textbf{Loop\_1\_Exit} \\ & \{[\textbf{Loop\_2}, true], \\ & \ [\textbf{Loop\_2}, false, true]\} \end{array}$$

Loop_2

$$\begin{array}{ll} around\_loop: & \textbf{Loop\_2\_Enter} \Rightarrow \textbf{Loop\_2\_Enter} \\ & \{[false, false]\} \\ exit\_loop: & \textbf{Loop\_2\_Enter} \Rightarrow \textbf{Loop\_2\_Exit} \\ & \{[true], \\ & \ [false, true]\} \end{array}$$

8.6 Finding flows

The number of iterations is set at one. This is sufficient to demonstrate the process of generating patterns. However, a higher number of iterations is expected to be required to make any useful insights.

$$\begin{array}{ll} flow: & \textbf{Start} \quad \Rightarrow \quad \textbf{Stop} \\ & [\textbf{Loop\_1}] \end{array}$$

8.6.1 Expanding Loop_1

N around, leap, and exit

$$[\textbf{Loop\_2}, false, false, \textbf{Leap}, \textbf{Loop\_2}, true]$$
$$[\textbf{Loop\_2}, false, false, \textbf{Leap}, \textbf{Loop\_2}, false, true]$$

N around, and exit

$$[\textbf{Loop\_2}, false, false, \textbf{Loop\_2}, true]$$
$$[\textbf{Loop\_2}, false, false, \textbf{Loop\_2}, false, true]$$

Exit

$$[\textbf{Loop\_2}, true]$$
$$[\textbf{Loop\_2}, false, true]$$

8.6.2 Expanding Loop_2

Every occurrence of **Loop_2** in a journey can be replaced with one of the following. Note that there are 6 journeys above waiting for **Loop_2** details of which there are 6 options, giving a possible $6 * 6 = 36$ combinations.

N around, leap, and exit

$$[false, false, \textbf{Leap}, true]$$
$$[false, false, \textbf{Leap}, false, true]$$

N around, and exit

$$[false, false, true]$$
$$[false, false, false, true]$$

Exit

$$[true]$$
$$[false, true]$$

8.7 Generating patterns from flows

8.7.1 Never finding true

The journey for never finding an element in the matrix that is true is traced. The **N around, leap, and exit** style will be adopted for both loops.

Path	$[false, false,$ **Leap**, $false, true,$ $false, false,$ **Leap**, $false, false,$ **Leap**, $false, true,$ $false, true]$
Depart cutpoint	**Start**
Code	Result:=False; I:=Boolean2DMatrixIndex'First;
Fact	$result_1 = False \quad \wedge$ $i_1 = Boolean2DMatrixIndex'First$
Arriving cutpoint	**Loop_1_Invariant(1)**
Pattern saved as	**Start** ⇒ **Loop_1_Invariant(1)**

Path	$[false, false,$ **Leap**, $false, true,$ $false, false,$ **Leap**, $false, false,$ **Leap**, $false, true,$ $false, true]$
Depart cutpoint	**Loop_1_Invariant(1)** (Increase count. Test for **Leap**. Is not found.)
Code	J:=Boolean2DMatrixIndex'First;
Fact	$j_1 = Boolean2DMatrixIndex'First$
Arriving cutpoint	**Loop_2_Invariant(1)**
Pattern saved as	**Loop_1_Invariant(1)** ⇒ **Loop_2_Invariant(1)**

Path	$[false, false,$ **Leap**, $false, true,$ $false, false,$ **Leap**, $false, false,$ **Leap**, $false, true,$ $false, true]$
Depart cutpoint	**Loop_2_Invariant(1)** (Increase count. Test for **Leap**. Is not found.)
Condition	SearchMatrix(I,J)=True
Pop Path	**false** ⇐ $[false,$ **Leap**, $false, true,$ $false, false,$ **Leap**, $false, false,$ **Leap**, $false, true,$ $false, true]$
Fact	$not(element(data\_set, i_1, j_1) = True)$
Condition	J=Boolean2DMatrixIndex'Last
Pop Path	**false** ⇐ [**Leap**, $false, true,$ $false, false,$ **Leap**, $false, false,$ **Leap**, $false, true,$ $false, true]$
Fact	$not(element(data\_set, i_1, j_1) = True) \quad \wedge$ $not(j_1 = Boolean2DMatrixIndex'Last)$
Code	J:=Boolean2DMatrixIndex'Succ(J);
Fact	$not(element(data\_set, i_1, j_1) = True) \quad \wedge$ $not(j_1 = Boolean2DMatrixIndex'Last) \quad \wedge$ $j_2 = j_1 + 1$
Arriving cutpoint	**Loop_2_Invariant(2)**
Pattern saved as	**Loop_2_Invariant(1)** ⇒ **Loop_2_Invariant(2)**
Loop_2 modifiable set	$\{j\}$

Path	[**Leap**, $false, true$, $false, false$, **Leap**, $false, false$, **Leap**, $false, true$, $false, true$]
Depart cutpoint	**Loop_2_Invariant(2)** (Increase count. Test for **Leap**. Is found.)
Leap proof	$j_3 = \textbf{DefinedUnknown}_{leaped}$
Condition	SearchMatrix(I,J)=True
Pop Path	**false** $\Leftarrow$ [$true$, $false, false$, **Leap**, $false, false$, **Leap**, $false, true$, $false, true$]
Fact	$not(element(data\_set, i_1, j_3) = True)$
Condition	J=Boolean2DMatrixIndex'Last
Pop Path	**true** $\Leftarrow$ [$false, false$, **Leap**, $false, false$, **Leap**, $false, true$, $false, true$]
Fact	$not(element(data\_set, i_1, j_3) = True) \quad \wedge$ $j_3 = Boolean2DMatrixIndex'Last$
Condition	Result=True
Pop Path	**false** $\Leftarrow$ [$false$, **Leap**, $false, false$, **Leap**, $false, true$ $, false, true$]
Fact	$not(element(data\_set, i_1, j_3) = True) \quad \wedge$ $j_3 = Boolean2DMatrixIndex'Last$ $not(result_1 = True)$
Condition	I=Boolean2DMatrixIndex'Last
Pop Path	**false** $\Leftarrow$ [**Leap**, $false, false$, **Leap**, $false, true$, $false, true$]
Fact	$not(element(data\_set, i_1, j_3) = True) \quad \wedge$ $j_3 = Boolean2DMatrixIndex'Last \quad \wedge$ $not(result_1 = True) \quad \wedge$ $not(i_1 = Boolean2DMatrixIndex'Last)$
Code	I:=Boolean2DMatrixIndex'Succ(I);
Fact	$not(element(data\_set, i_1, j_3) = True) \quad \wedge$ $j_3 = Boolean2DMatrixIndex'Last \quad \wedge$ $not(result_1 = True) \quad \wedge$ $not(i_1 = Boolean2DMatrixIndex'Last) \quad \wedge$ $i_2 = i_1 + 1$
Arriving cutpoint	**Loop_1_Invariant(2)**
Pattern saved as	**Loop_2_Invariant(2)** $\Rightarrow$ **Loop_1_Invariant(2)**
Loop_1 modifiable set	$\{i\}$

Path	[**Leap**, $false, false,$ **Leap**, $false, true,$ $false, true$]
Depart cutpoint	**Loop_1_Invariant(2)** (Increase count. Test for **Leap**. Is found.)
Leap proof	$i_3 = \textbf{DefinedUnknown}_{leaped}$
Code	J:=Boolean2DMatrixIndex'First;
Fact	$j_4 = Boolean2DMatrixIndex'First$
Arriving cutpoint	**Loop_2_Invariant(3)**
Pattern saved as	**Loop_1_Invariant(2)** $\Rightarrow$ **Loop_2_Invariant(3)**

Path	$false, false,$ **Leap**, $false, true,$ $false, true$]
Depart cutpoint	**Loop_2_Invariant(3)** (Increase count. Test for **Leap**. Is not found.)
Skip details...	
Fact	$not(element(data\_set, i_3, j_4) = True) \quad \wedge$ $not(j_4 = Boolean2DMatrixIndex'Last)$ $j_5 = j_4 + 1$
Arriving cutpoint	**Loop_2_Invariant(4)**
Pattern saved as	**Loop_2_Invariant(3)** $\Rightarrow$ **Loop_2_Invariant(4)**

Path	**Leap**, $false, true,$ $false, true$]
Depart cutpoint	**Loop_2_Invariant(4)** (Increase count. Test for **Leap**. Is found.)
Leap proof	$j_6 = \textbf{DefinedUnknown}_{leaped}$
Skip details...	
Fact	$not(element(data\_set, i_3, j_6) = True) \quad \wedge$ $j_6 = Boolean2DMatrixIndex'Last \quad \wedge$ $not(result_1 = True) \quad \wedge$ $i_3 = Boolean2DMatrixIndex'Last$
Code	return Result;
Fact	$not(element(data\_set, i_3, j_6) = True) \quad \wedge$ $j_6 = Boolean2DMatrixIndex'Last \quad \wedge$ $not(result_1 = True) \quad \wedge$ $i_3 = Boolean2DMatrixIndex'Last \quad \wedge$ $function\_return_1 = result_1$
Arriving cutpoint	**Stop** (Cease algorithm)
Pattern saved as	**Loop_2_Invariant(4)** $\Rightarrow$ **Stop**

9 Conclusions

A process outlined in CS-note 34 has been divided into several steps. The main algorithmic step of generating patterns is given a detailed description and two examples are presented. The next phase is to consider how these patterns are generalised into useful proof annotations. The results from this may lead to small refinements to the algorithm presented here.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.

[2] Jean-Francois Bergeretti and Bernard A. Carr. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985.

[3] E. W. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[4] Ana M. Erosa and Laurie J. Hendren. Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In *ICCL*, 1994.

[5] G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering,* (Toronto, Ontario; July 14-16, 1995), pages 188–197. IEEE Computer Society Press (Order Number PR07111), July 1995.

[6] Huang and Leng. Generalized loop-unrolling: a method for program speed-up.

[7] Praxis Critical Systems. Proof of the absence of run-time errors from spark programs a case study. 1996.

[8] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[9] Mark Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.