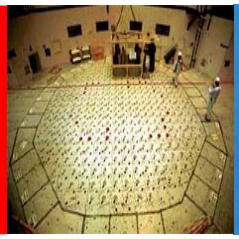
MALPAS

Advanced Software Analysis and Verification

Example Analysis



This page is left intentionally blank

Issue 1

Atkins
The Barbican, East Street
Farnham, Surrey, UK

Visit our web-site at www.atkinsglobal.com

This page is left intentionally blank

Contents

This page is left intentionally blank

MALPAS Example

Introduction

The following example is more instructive than a theoretical description of each of the analysis techniques. The example is a short program that implements a simplified reactor global trip function. Its specification is given below:

Specification

A reactor has a set of 5 sensors, each with its own range of values in which the reactor operates safely. A protection system examines these sensors to determine whether they are within their safety limits. If they are not, the system triggers a partial trip signal corresponding to the sensor that is out of range. The system then decides on the basis of these trip signals and sensor data whether to signal a full reactor – or *global* trip. The conditions for a global trip are as follows:

- If 3 or more partial trips out of the possible 5 are signalled, then the reactor will trip automatically. This is known as a type 1, automatic trip.
- If and only if 2 partial trips out of the possible 5 are signalled, then the reactor will signal a global trip if the signals are in a particular combination. This is known as a type 2, **combination trip**. The combinations of partial trips which set a global trip signal (type 2) are sensors 1 & 2, 1 & 4, 1 & 5, 2 & 4, 2 & 5, 3 & 4, 4 & 5.
- If and only if 1 partial trip out of the possible 5 is signalled, then depending on the percentage variance it shows from its permitted range, the reactor will trip. This is known as a type 3, **percentage trip**. In this example, the global trip is signalled if:

  [Amount that sensor is outside the range] / [range of safe values] >= 0.5

Normal operation (i.e. absence of any global trip condition) is signalled as type 0.

The program has been implemented in C and although valid, it is not intended as an example of good coding practice. The basis of the implementation is that the function `globtrip` is called to update the trip status and return the trip type. The logic to determine whether a particular set of partial trips represents either a combination trip or a percentage trip is implemented as the separate functions `combtrip` and `perctrip` respectively.

C Listing

```
int combtrip (char a1, char a2, char a3, char
a4, char a5) {

        if      ((a1==1 && a2==1) ||
                 (a1==1 && a4==1) ||
                 (a1==1 && a5==1) ||
                 (a2==1 && a4==1) ||
                 (a2==1 && a5==1) ||
                 (a3==1 && a4==1) ||
                 (a4==1 && a5==1))

                 return (1);
        else
                 return (0);

}

int perctrip (int val, int h, int l) {

int temp1, temp2;

/* this procedure implements partial trip rule
3, see below */

        if (val > h) {

            temp1=((val-h)/(h-l));
            temp2=((val-h) % (h-l));
            }
        else
            {
             if (val < l) {
                 temp1=((l-val)/(h-l));
                 temp2=((l-val) % (h-l));
                 }
            }
        if ((temp1<=0) && (temp2>=5))
            return (1);
        else
            return (0);
}

char globtrip (char sensors[5]) {

struct {char low; char hi;} sensor_range[5];
char i,j,k;
char a,b,c;
char p_trip [5];
char reactor_trip;

/* checks the sensors to see if they are outside
the bounds of their safe levels. Set k to one
such instance. */

        for (i=0;i<5;i++) {

            if ((sensors[i] <=
sensor_range[i].hi) &&
                (sensors[i] >=
```

```
sensor_range[i].low))

                p_trip[i] = 0;
            else {
                p_trip[i] = 1;
                k = i;
                 }

        } /* while */

/* we now have an array of flags called p_trip
indicating partial trips from certain sensors */

/* we now need to determine which combinations
of partial trips trigger a full reactor trip */

/* rule #1, the reactor will trip if there are 3
or more partial trips */

        j = 0;
        for (i=0; i<=5; i++)
                j=j+p_trip[i];

        if (j>=3)
            reactor_trip = 1;

        else if (j=2)

/* rule #2, the reactor will trip if there are
partial trips in the following combinations,
(0,1), (0,3), (0,4),(1,4), (1,3),(2,3),(3,4) */
        {
             if (combtrip(p_trip[0],
                          p_trip[1],
                          p_trip[2],
                          p_trip[3],
                          p_trip[4]))
                  reactor_trip = 2;
        }

        else if (j==1)

/* rule #3, if only one sensor is outside its
safety range then only
trip if the percentage it has deviated is
greater than 50% */

        {
                    a = sensors[k];
                    b = sensor_range[k].hi;
                    c = sensor_range[k].low;

                    if (perctrip(a,b,c))
                       reactor_trip = 3;
        }
        else reactor_trip = 0;
        return (reactor_trip);
}
```

Intermediate Language Translation

C programs can be automatically translated into MALPAS Intermediate Language (IL) by using the C translator. There are two parts to the translation of each procedure, a specification and a body. The procedure specification is used to define all the significant characteristics of the procedure, such as the inputs, outputs and their types. The basic operation of MALPAS is to analyse the procedure body and compare the results against its specification.

The specification generated for the procedure `perctrip` is:

```
PROCSPEC perctrip(
   IN    val : int
   IN    h : int
   IN    l : int
   OUT   result_perctrip : int);
```

The corresponding body section is:

```
PROC perctrip;
   VAR temp1 : int;
   VAR temp2 : int;

   [ this procedure implements partial trip rule
3, see below ]
   IF (val > h)
   THEN
      $malpas_check(h-l NE 0);
      temp1 := (val-h) / (h-l);
      $malpas_check(h-l NE 0);
      temp2 := (val-h) MOD (h-l)
   ELSE
      IF (val < l)
      THEN
         $malpas_check(h-l NE 0);
         temp1 := (l-val) / (h-l);
         $malpas_check(h-l NE 0);
         temp2 := (l-val) MOD (h-l)
      ENDIF [if]
   ENDIF [else];
   IF ((temp1 <= 0) AND (temp2 >= 5))
   THEN
      result_perctrip := 1;
      STOP
   ELSE
      result_perctrip := 0;
      STOP
   ENDIF [else]
ENDPROC [perctrip]
```

IL has many of the characteristics of structured programming languages in common use, and is easily learnt by any moderately experienced software engineer.

MALPAS analyses each procedure individually. This allows the analysis process to be broken down into separate procedures, just as programs are. The analysis is amenable to team working, since the analysis of procedures can be performed in parallel. Although there is no predefined order in which analysis must be done, it is often convenient to proceed bottom-up, such that the analysis starts with those procedures that call no others, and continues with those procedures that only call analysed procedures. Using this approach, a complete picture of the behaviour of a program can be built up.

Structural Properties

To aid in setting up the analysis sequence, one of the first things that MALPAS generates is a procedure call graph. The call graph for the example program is shown below[1].

[1] The calls to procedure `$malpas_check` are semantically neutral and have been introduced automatically during the translation process. There is no corresponding C source, or need to analyse this extra procedure, it is simply one of the mechanisms available in MALPAS to check code integrity (in this case it is used to demonstrate that division by zero does not occur).

```
Call Graph
----------

Section:     Calls:
--------     ------

combtrip     -

perctrip     $malpas_check

globtrip     $malpas_check     combtrip     perctrip
```

Complexity Information

MALPAS can provide some simple complexity information about the IL model submitted for analysis. For our example, this information is as follows:

```
   Statistics for procedure combtrip:

   Number of nodes:                          8
   Number of arcs:                           8
   Number of identifiers:                   27
   Number of conditional nodes:              1


            Statistics for procedure perctrip:

   Number of nodes:                         20
   Number of arcs:                          22
   Number of identifiers:                   27
   Number of conditional nodes:              3


            Statistics for procedure globtrip:

   Number of nodes:                         56
   Number of arcs:                          63
   Number of identifiers:                   37
   Number of conditional nodes:              8
```

From this follows the McCabe Cyclomatic Complexities for the three procedures. They are 2, 4, and 9 respectively (simply one more than the number of conditional nodes). Given that 10 is often regarded as the highest desirable complexity for a single procedure, we note that all are “satisfactory” but that procedure `globtrip` is approaching the limit.

Next MALPAS considers the structure of each procedure and identifies all entry and exit points. Ideally in high integrity software, all procedures and loops should be structured with single entry and exit points. MALPAS will therefore report problems such as unreachable statements, non-terminating loops and loops with multiple entries. This form of analysis is of most benefit when applied to assembly code because assembly code lacks the high level control constructs such as IF and WHILE. Also, it is common for assembly language programmers to minimise the space needed for their code, which can easily lead to poor structuring.

For our example, acceptable structure is reported for all 3 procedures. Furthermore the analyst is told that the procedures `combtrip` and `perctrip` are loop-free, and that two loops were found in procedures `globtrip`. The output for procedure `globtrip` is shown below.

```
Control Flow Summary
====================

[CF1]  Structure acceptable

Each loop has one entry

Each loop has one exit

No unreachable code

No dynamic halts

The graph was fully reduced after the following
stages:
      KASAI, HECHT, HK, TOTAL

Loophead node numbers:      #6      #16
```

Procedure Interfaces

In order to proceed to the next stage of analysis, it is necessary to capture some additional information about the interfaces between the various procedures. A statement as to how each output depends on the inputs is required. Ideally the analyst states his expectations based on information contained in the program design documentation, so that MALPAS can check that the code itself is consistent. Failing this, MALPAS can generate the information from the code, and the analyst can inspect it for correctness. In our example[2], MALPAS generates the following for `combtrip`:

```
DERIVES
result_combtrip FROM [INs/INOUTs]
a1 & a2 & a3 & a4 & a5
```

and similarly for procedure `perctrip`, the generated information is:

```
DERIVES
result_perctrip FROM [INs/INOUTs]
h & l & val
```

For each procedure there is a statement (a “DERIVES list”) about each output and its dependence on inputs. Inputs and outputs in this respect can be either explicit parameters or global variables. Consider the above DERIVES list for procedure `perctrip`. This states that the effect of the procedure is to compute the output `result_perctrip` as a function of the inputs `h`, `l`, and `val`. In this instance the nature of the function is left abstract. It is however possible to introduce named functions for use in DERIVES lists and to express their semantics.

[2] MALPAS does not mandate similar information for the top level procedure `globtrip`, because it is not called anywhere. MALPAS will however check the information against the code if it is provided.

Data Integrity

MALPAS considers code integrity at a number of different levels. The first phase of checking looks for various forms of internal inconsistency. For example:

- variables should be initialised before they are first read
- outputs should be written on all paths
- outputs should be a function of inputs only (i.e. not of outputs or local variables)
- the information flow through a procedure should be consistent with any DERIVES lists provided

For our sample code, MALPAS finds no fault with the procedure `combtrip`, but anomalies are reported for the procedures `perctrip` and `globtrip`. Faults can be reported in a number of different ways according to the user options selected; the following is the report for `perctrip` using one of the standard options:

```
[DU19]   VARs that are sometimes read when undefined
         temp1   temp2

[IF2]    Information Flow dependencies not in DERIVES
         result perctrip  VARs/OUTs  :  temp1   temp2
```

These are two manifestations of the same problem, namely that there is at least one path through `perctrip` on which the variables `temp1` and `temp2` are read without having first been written.

Adding an else clause so that the first conditional in `perctrip` reads as follows can cure the problem:

```
if (val > h) {
   temp1=((val-h)/(h-l));
   temp2=((val-h) % (h-l));
   }
else if (val < l) {
   temp1=((l-val)/(h-l));
   temp2=((l-val) % (h-l));
   }
else return (0);
```

The other main aspect of data integrity considered by MALPAS concerns the use of partial functions. This is discussed more fully below under the heading “functional integrity”.

Functional Behaviour

Functional behaviour can be investigated in MALPAS through use of either the semantic or compliance analysers. With the compliance analyser, the analyst has to cast the specification into the formal specification language of IL. The techniques will be familiar to practitioners of formal methods (i.e. specification in mathematical notation supported by precise semantics and proof rules). For each procedure the analyst provides pre- and post- conditions. These may in turn be supported by additional types, functions and rules for algebraic term re-writing.

Due to the specialist skills needed for compliance analysis, semantic analysis is often a more attractive proposition. With this analyser, a substantially different and improved representation of the code is generated. With two distinct views of the code to check against the specification, the original source and the MALPAS representation, the analyst has a very much better chance to latch onto any problems that may exist.

MALPAS considers each loop free region of a procedure separately and represents the functional behaviour of that region as a series of possible execution paths through the code. For each path, the set of input conditions that leads to execution of that particular path is given together with the final value of each variable, expressed in terms of the program state on entry to the region under consideration. In essence a set of parallel operations which precisely captures the code behaviour is provided to supplement the sequential view given by the source code.

Taking our example to illustrate this concept, consider the procedure `globtrip`, in particular the final section of the procedure starting at the statement:

```
if (j>=3)
```

In MALPAS terminology this is a loop free region identified thus:

```
From node #19 to node #END   (3 semantically
impossible paths detected)
```

MALPAS reports that there are three paths, and tabulates their influence on the output `result_globtrip` as follows:

```
Paths Table
===========

Conditions           Paths
                   1   2   3
C1                 T   T   T
C2                 T   F   F
C3                     T   F
result_globtrip A11 A12 A13
```

The execution conditions for each of the three paths is expressed in terms of C1, C2 and C3. For example path three executes if the following is true of the program state on entry to the region:

```
C1 AND ¬C2 AND ¬C3
```

The meaning of each term is provided in a separate key as

follows:

```
Conditions
==========

  C1: i > 5
  C2: j > 2
  C3: _combtrip.result_combtrip(p_trip ! 0,
p_trip ! 1, p_trip ! 2, p_trip ! 3, p_trip ! 4) = 0
```

The outcome of executing each of the paths is expressed in the final row of the table, viz:

```
result_globtrip A11 A12 A13
```

where A11, A12 and A13 represent "net" assignments to `result_globtrip` that occur, expressed in terms of the program state on entry to the region. As before, MALPAS supplies the meaning in a separate key:

```
Actions
=======

Assignments to result_globtrip
  A11: 1
  A12: reactor_trip
  A13: 2
```

Clearly there is a problem with the code. Our expectation is that the result will be one of the values 0, 1, 2 or 3 and that all four of these outcomes are possible. MALPAS tells us that only the outcomes 1 and 2 are possible, and that there is a path where the initial state of the local variable `reactor_trip` is the final state of `result_globtrip`. Since other parts of the MALPAS output tell us that the initial state of `reactor_trip` is undefined[3] this is clearly unsatisfactory. On closer examination it turns out that there are two separate faults in the code.

Firstly the test:

```
if (j=2)
```

should be:

```
if (j==2)
```

Secondly, setting `reactor_trip` to 0 as a final "catch all" else clause of the cascaded condition in this program region is incorrect, as some of the other paths through the conditional do not set it.

[3] It is reported during data integrity analysis, and can also be found in the semantic analysis output for the preceding regions.

Once these problems are corrected, the output from MALPAS is as follows:

```
From node #19 to node #END   (0 semantically
impossible paths detected)

Conditions
==========

  C1: i > 5
  C2: j > 2
  C3: j = 2
  C4: _combtrip.result_combtrip(p_trip ! 0,
p_trip ! 1, p_trip ! 2, p_trip ! 3,p_trip ! 4) =
0
  C5: j > 0
  C6: j = 1
  C7: _perctrip.result_perctrip(int
(HI sensor_range ! k)), int(LOW (sensor_range !
k)),int(sensors_array ! (k + OFFSET(sensors))))
= 0

Actions
=======

Assignments to result_globtrip
  A11: 1
  A12: 0
  A13: 2
  A14: 3

Paths Table
===========

Conditions                  Paths
                  1   2   3   4   5   6
C1                T   T   T   T   T   T
C2                T
C3                    T   T
C4                    T   F
C5                            F
C6                                T   T
C7                                T   F
result_globtrip A11 A12 A13 A12 A12 A14
```

This is now much nearer to the required behaviour. There are four possible outcomes for output `result_globtrip` and they correspond to the possibilities described in the specification. Recalling that `j` is the number of partial trips, a type 1 trip occurs whenever `j>2` (path 1) and a type 0 trip whenever `j<=0` (path 4). When `j=2` (paths 2 and 3), the trip type is 2 for critical combinations and 0 otherwise. Finally when j=1 (paths 5 and 6), the trip type is 3 if the percentage test fails and 0 otherwise.

Careful consideration of these 8 possibilities reveals another possible problem. When there are 2 partial trips, a percentage trip (type 3) never occurs, yet there is the possibility that one or both partial trips can fail the percentage test.

As discussed above, a different approach to demonstrating functional correctness is to use the compliance analyser. We will explore in outline how this might proceed for the procedure `perctrip`. Compliance analysis attempts to prove that the code in the body of a procedure has the properties and functional behaviour stated in the PROCSPEC. Generally required behaviour is expressed in the DERIVES list and

properties are expressed as POST conditions. The PROCSPEC shown below for `perctrip` is set up to attempt a proof that the procedure correctly detects “too high” conditions.

```
FUNCTION toolow (integer, integer, integer):
boolean;
FUNCTION toohigh (integer, integer, integer):
boolean;

REPLACE(v, h, l:integer) toolow(v, h, l)
BY (l-v/h-l) > 0 OR ((2*(l-v MOD h-l)) - (h-l))
>= 0;

REPLACE(v, h, l:integer) toohigh(v, h, l)
BY (v-h/h-l) > 0 OR ((2*(v-h MOD h-l)) - (h-l))
>= 0;

PROCSPEC [perctrip] perctrip(
      IN    val : int
      IN    h : int
      IN    l : int
      OUT   result_perctrip : int)
PRE  h>l
POST  ('val>'h AND toohigh('val, 'h, 'l))
result_perctrip = 1;
```

The post-condition here expresses what we are trying to prove. It also illustrates how the specification is typically built up by introducing additional functions with semantics provided for them through the use of term rewriting rules. Variables are primed where appropriate to denote that they are initial values.

If a pre-condition is specified, then the proof demonstration is performed for a restricted set of input conditions only. In our example we have assumed that the high limit `h`, and the low limit `l` for the sensor are such that `h>l`. If compliance analysis of the calling procedures were performed, we would get a proof that this assumption is correct.

The output of compliance analysis is a “threat”, i.e. the set of input conditions under which the code violates the specification. If the analyser reports `_threat:=false` then the code satisfies the specification. In this instance the code is not correct. Investigation of the threat will reveal that the code:

```
if ((temp1<=0) && (temp2>=5))
```

should be:

```
if ((temp1>0) || (((2*temp2)-(h-l))>=0))
```

Functional Integrity

There is more to the code integrity than the data integrity topic discussed above. For example, we need to check that array indexes are always in bounds, and that overflows do not occur.

To state the problem more generally, many operations are partial. They are not defined for all inputs, and therefore have associated with them an integrity pre-condition. If this pre-condition is not satisfied, then the code may or may not behave in accordance with expectations. A typical operation with an

integrity pre-condition is division; the divisor must not be zero.

MALPAS supports functional integrity checking in a number of ways. We will demonstrate below one form of checking supported by the compliance analyser.

The fragment of IL shown below is the translation of the second loop in procedure `globtrip`.

```
j := char(0);
   i := char(0);
   LOOP [for]
      ASSERT i>=0 AND i<=6;
      EXIT WHEN (i > 5);
      $malpas_check(i>=0 AND i<SIZE POINTER
$char_array_at_p_trip);
      j := char(j+p_trip!(i));
      i := char(i+1)
   ENDLOOP [for];
```

Compliance Analysis

The call to `$malpas_check` was added during the translation from C to IL. This procedure has no outputs and so the call has no effect, i.e. the semantics of the program remain unchanged. It does however have the following pre-condition, which corresponds to valid indexing of the array `p_trip`:

```
i>=0 AND i<5
```

The compliance analyser checks that the pre-conditions of all called procedures are satisfied, so it will report any possibility that `p_trip` is indexed out of range. The traditional approach to program proving is applied, whereby an invariant condition has to be identified for each loop to carry the proof through the loop. This is the purpose of the `ASSERT` statement in the IL text above.

The output from compliance analysis for this loop is shown below. The assignment to `_threat` tells us that the loop invariant is correct, and the assignment to `_false_pre_$malpas_check_#28` tells us that the array index i can have the value 5 in violation of an integrity requirement associated with statement #28.

```
From node :    #26
To node   :    #26

MAP

_false_pre_$malpas_check_#28 := i = 5

_threat := false

ENDMAP
```

The underlying problem is the indexing of array `p_trip`. The statement

```
for (i=0; i<=5; i++)
```

is incorrect and should read

```
for (i=0; i<5; i++)
```

Conclusions

The example analysis given above should give some idea of how MALPAS can be used in the verification of code. In this example, we have taken a small sample of C and applied a cross-section of MALPAS analysis techniques to it. The analysis has:

- confirmed a satisfactory level of complexity
- confirmed the use of good structured programming practice
- helped in classifying parameters into inputs and outputs
- found problems due to uninitialised data in the procedures `globtrip` and `perctrip`
- identified incorrect functionality in procedures `globtrip` and `perctrip`
- detected an ambiguity in the specification concerning the behaviour required when two partial trips are present
- found incorrect indexing of an array in `globtrip`

Despite our use of a small example, MALPAS analysis of large complex software is perfectly feasible. Large software assessments should be supported with configuration management and care taken when integrating separately analysed modules. To date, MALPAS has been applied successfully on a number of projects each with a requirement to verify programs in the order of 100,000 source lines long.