Astrée: from Research to Industry

David Delmas and Jean Souyris

Airbus France S.A.S.
316, route de Bayonne
31060 TOULOUSE Cedex 9, France

{David.Delmas, Jean.Souyris}@Airbus.com

Abstract. Airbus has started introducing abstract interpretation based static analysers into the verification process of some of its avionics software products. Industrial constraints require any such tool to be extremely precise, which can only be achieved after a twofold specialisation process: first, it must be designed to verify a class of properties for a family of programs efficiently; second, it must be parametric enough for the user to be able to fine tune the analysis of any particular program of the family. This implies a close cooperation between the tool-providers and the end-users. Astrée is such a static analyser: it produces only a small number of false alarms when attempting to prove the absence of run-time errors in control/command programs written in C, and provides the user with enough options and directives to help reduce this number down to zero. Its specialisation process has been reported in several scientific papers, such as [1] and [2]. Through the description of analyses performed with Astrée on industrial programs, we give an overview of the false alarm reduction process from an engineering point of view, and sketch a possible customer-supplier relationship model for the emerging market for static analysers.

Keywords: avionics software, verification, abstract interpretation, static analysis, run-time errors, Astrée.

Introduction

Verification activities are responsible for a large part of the overall costs of avionics software developments. Considering the steady increase of the size and complexity of this kind of software, classical Validation and Verification processes, based on massive testing campaigns and complementary intellectual analyses, hardly scale up within reasonable costs. Therefore, Airbus has decided to introduce formal proof techniques providing product-based assurance into its own verification processes.

Available formal methods include model checking, theorem proving and abstract interpretation based static analysis ([3], [4], [5]). The items to be verified being final products, i.e. source or binary code, model checking is not considered relevant. Some theorem proving techniques have been successfully introduced to verify limited software subsets. However, to prove properties of complete real-size programs with these techniques does not seem to be within the reach of software engineers yet.

On the other hand, abstract interpretation based static analysers aiming at proving specific properties on complete programs have been shown to scale to industrial safety-critical programs. Among these properties, one is to quote worst-case execution time assessment ([7]), stack analysis, accuracy of floating-point computations ([8]), absence of run-time errors, etc. Moreover, these analysers are automatic, which is obviously a requirement for industrial use.

Yet, however precise any such tool may be, a first run of it on a real-size industrial software will typically produce at least a few false alarms. For safety and industrial reasons, this number of false alarms should be as small as possible, and an engineering user should be able to be reduce it down to zero by a new fine tuned analysis. Indeed, the fewer false alarms are produced, the fewer costly, time-consuming and error-prone complementary intellectual analyses are necessary. Hence the need for both specialised and parametric tools outputting comprehensible diagnoses, which can only be achieved through a close cooperation between the tool-providers and the end-users.

The Astrée static analyser has proved to meet these requirements. In this paper, we first give an overview of this tool and its specialisation process. Then, we describe the analysis process: how we run the tool, how we analyse the resulting alarms, and how we tune the parameters of the tool to reduce the number of false alarms. Next, we illustrate the alarm reduction process with a report on the analysis of a real-size industrial control/command program, and an example of alarm analysis for another avionics program. Finally, we assess the analysis results, and state Airbus's position on the perspectives for a tool such as Astrée within a possible new kind of customer-supplier relationship.

The Astrée analyser

Astrée is a parametric Abstract Interpretation based static analyser that aims at proving the absence of RTE (Run-Time Errors) in programs written in C.

The underlying notion has been defined in several papers, such as [2, §2]: "*The absence of runtime errors is the implicit specification that there is no violation of the C norm (e.g., array index of bounds), no implementation-specific undefined behaviours (e.g., floating-point division by zero), no violation of the programming guidelines (e.g., arithmetic operators on short variables should not overflow the range `[-32768,32767]` although, on the specific platform, the result can be well-defined through modular arithmetics).*"

As explained in [1, §3.1], this tool results from the refinement of a more general-purpose analyser. It has been specialised in order to analyse synchronous control/command programs very precisely, thanks to specific iterator and abstract domains described in [1]. This is the result of the "per program family specialisation process". Furthermore, the parametric nature of Astrée makes it possible for the user to specialise it for any particular program within the family.

The alarm reduction process

Even for a program belonging to the family of synchronous control/command programs, the first run of Astrée will usually produce false alarms to be further investigated by the industrial user. The user must tune the tool parameters to improve the precision of the analysis for a particular program. This final “per program specialisation process” matches the adaptation by parameterisation described in [1, §3.2].

The need for full alarm investigation

We do not use Astrée to search for possible run-time errors; we use it in order to prove that no run-time error can ever occur. As a consequence, every single alarm has to be investigated.

Besides, every time Astrée signals an alarm, it assumes the execution of the analysed program to stop whenever the precondition of the alarm is satisfied, because the program behaviour is undefined in case of error, e.g. an out-of-bounds array assignment might destroy the code[1]. Thus, any satisfiable alarm condition may “hide” more alarms.

Let us give a simple example with variable `i` of type `int` in interval `[0,10]`:

```
1       int t[4];
2       int x = 1;
3       int y;
4       t[i] = 0;
5       y = 1/x;
```

Astrée reports a warning on line 4 (invalid dereference), but not on line 5. However, executing instruction 4 with `i>3` will typically overwrite the stack, e.g. set variable `x` to `0`, so that instruction 7 may produce a division by zero. Since the execution is assumed to stop whenever `i>3` on line 4, Astrée assumes `i` to be in interval `[0,3]` from line 4.

That is the reason why exhaustive alarm analysis is required: every false alarm should disappear by means of a more precise automated analysis, or, failing that, be proved by the user to be impossible in the real environment of the program.

[1] For a complete explanation, refer to [6, §4.1].

How to read an alarm message

The following alarm will be discussed later in this paper:

```
P3_A_3.c:781.212-228::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]
```

Let us explain how this message reads. Astrée warns that some simple precision floating-point computation may yield a result that cannot be represented in the type `float`. It points precisely to the operation that may cause such a run-time error: line 781 of the (pre-processed) file P3_A_3.c, between columns 212 and 228[2]:

```
_R1= PADN10 - X3A3Z15 ;
```

where variables `_R1`, `PADN10` and `X3A3Z15` have type `float`.

All other information describe the context of the alarm. The analysis entry point[3] is the `APPLICATION_ENTRY` function, defined on line 449 of some file. This function contains a loop on line 466. From the fourth loop iteration, at least in the abstract semantics computed by the tool, there exists an execution trace such that:

- function `SEQ_C4_10_P` is called on line 673;
- `SEQ_C4_10_P`, defined in some file, calls function P_3_A_3_P on line 118 of this file;
- the result of the floating-point subtraction of the operands `PADN10` and `X3A3Z15` does not range in `[-3.40282e+38, 3.40282e+38]`.

Of course, this does not necessarily mean there exists such an erroneous execution in the concrete semantics of the program: one is now to address this issue via a dedicated method.

Dealing with alarm investigation

As explained above, every alarm message refers to a program location in the pre-processed code. It is usually useful to get back to the corresponding source code, to obtain readable context information.

When Astrée fails to prove an operation free from run-time errors, it outputs an alarm message, together with a brief explanation of the reason why the alarm was raised. Most such alarm conditions are expressed in terms of intervals. To investigate them, one makes use of the global invariant of the most external loop of the program, which is available in the Astrée log file (provided the `--dump-invariants` analysis option is set). Considering every global variable processed by the operation

[2] Line numbers start from 1, whereas column numbers start from 0.

[3] The user provides Astrée with an entry point for the analysis, by means of the `--exec-fn` option. Usually, this is the entry point of the program.

pointed to by an alarm, one may extract the corresponding interval, which is a sound over-approximation of the range of this variable[4]. The user may also use the `__ASTREE_log_vars((`V_1`,...,`V_n`));` directive when the ranges of local variables are needed.

Then, we have to go backwards in the program data-flow, in order to get to the roots of the alarm: either a bug or insufficient precision of the automated analysis. This activity can be quite time-consuming. However, it can be made easier for a control/command program that has been specified in some graphical stream language such as SAO, SCADE™ or Simulink™, especially if most intermediate variables are declared global. The engineering user can indeed label every arrow representing a global variable with an interval, going backwards from the alarm location. The origin of the problem is usually found when some abrupt inexplicable increase in variable ranges is detected.

At this point, we know whether the alarm originated in some local code with limited effect or in some definite specialised operator (i.e., function or macro-function). Indeed, an efficient approach is first to concentrate on alarms in operators that are used frequently in the program, especially if several alarms with different stack contexts point to the same operators: such alarms will usually affect the analysis of the calling functions, thus raising more alarms. For control/command programs with a fairly linear call-graph, it can be also quite profitable to pick alarms originating early in the data-flow first. To get rid of such alarms may help eliminate other alarms originating later in the data-flow.

Once we have found the roots of the alarm, we will usually need to extract a reduced example to analyse it. Therefore, we:

- write a small program containing the code at stake;
- build a new configuration file for this example, where the input variables `V` are declared `volatile` by means of the `__ASTREE_volatile_input((V [min, max]));` directive. The variable bounds are extracted from the global invariant computed by Astrée on the complete program;
- run Astrée on the reduced example (which takes far less time than on a complete program).

Such a process is not necessarily conservative in terms of RTE detection. Indeed, as the abstract operators implemented in Astrée are not monotonic, an alarm raised when analysing the complete program may not be raised when analysing the reduced example. In this case, this suggests (though does not prove) that the alarm under investigation is probably false, or that the reduced example is not an actual slice of the complete program with respect to the program point pointed to by the alarm.

[4] If the main loop is unrolled N times (to improve precision), the N first values of variables are not included in the global invariant. The `__ASTREE_log_vars((`V_1`,...,`V_n`));` directive is needed to have Astrée output these values. However, the global invariant is enough to deal with alarms occurring after the Nth iteration.

However, this hardly ever happens in practice: every alarm raised on the complete program will usually be raised on the reduced example as well. Furthermore, it is much easier to experiment with the reduced example:

- adding directives in the source to help Astrée increase the precision of the analysis;
- tuning the list of analysis options;
- changing the parameters of the example itself to better understand the cause of the alarm.

Once a satisfactory solution has been found on reduced examples, it is re-injected into the analysis of the complete program: in most cases, the number of alarms decreases.

Verifying a control/command program with Astrée

Let us illustrate this alarm reduction process for a periodic synchronous control/command program developed at Airbus. Most of its C source code is generated automatically from a higher-level synchronous data-flow specification. Most generated C functions are essentially sequences of calls of macro-functions coded by hand. Like in [1, §4], it has the following overall form:

```
declare volatile input, state and output variables;
initialise state variables;
loop forever
    read volatile input variables,
    compute output and state variables,
    write to volatile output variables;
    wait for next clock tick;
end loop
```

This program is composed of about 200,000 lines of (pre-processed) C code processing over 10,000 global variables. Its control-flow depends on many state variables. It performs massive floating-point computations and contains digital filters.

Although an upper bound of the number of iterations of the main loop is provided by the user, all these features make precise automatic analysis (taking rounding errors into account) a grand challenge. A general-purpose analyser would not be suitable.
Fortunately, Astrée has been specialised in order to deal with this type of programs: only the last step in specialisation (fine tuning by the user) has to be carried out. The automated analyses are being run on a 2.6 GHz, 16 Gb RAM PC. Each analysis of the complete program takes about 6 hours.

First analysis

Program preparation

The program is being prepared in the following way:

- some assembly functions are recoded in C or removed;
- compiler built-in functions are redefined:

```
  double fabs(double x) {
      if (x>=0.) return x; else return (-x);
  }
  double sin(double x) {
      double y;
      __ASTREE_known_fact((y>=-1.0));
      __ASTREE_known_fact((y<=1.0));
      return y;
  }
  double cos(double x) {
      double y;
      __ASTREE_known_fact((y>=-1.0));
      __ASTREE_known_fact((y<=1.0));
      return y;
  }
  volatile void waitforinterrupt(void) {
      __ASTREE_wait_for_clock(());
}
```

In this way, we provide Astrée with a model of external functions. On the one hand, the `ASTREE_known_fact((...));` directive helps Astrée bound the values computed by trigonometric functions. On the other hand, the `__ASTREE_wait_for_clock(());` directive delimits the code executed on each iteration of the main loop. Its counterpart is the `__ASTREE_max_clock((3600000));` directive in the analysis configuration file, which provides Astrée with an upper bound of the number of iterations of this loop.

Finally, the analysis configuration file contains `__ASTREE_volatile_input((V [min, max]));` directives describing the ranges of all the volatile inputs of the program.

Analysis options

Table 1. List of options.

Option	Meaning
`--config-sem prog.config`	Analysis configuration file.
`--exec-fn APPLICATION_ENTRY`	Entry point of the program.
`--inner-unroll 15`	Inner loops[5] are unrolled at most 15 times (to improve precision).
`--dump-invariants`	Prints the invariant of the most external loop of the program, i.e. the ranges of all global variables.

Results

Under the above conditions, this first analysis produces 467 alarms.

Let us take a closer look at the three following messages, the first of which has been described earlier:

```
P3_A_3.c:781.212-228::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]

P3_A_3.c:781.355-362::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:if@781=true:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]

P3_A_3.c:781.409-416::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C4_10_
P@673:call#P_3_A_3_P@118:if@781=true:]:

WARN: float arithmetic range [-inf, inf] not included
in [-3.40282e+38, 3.40282e+38]
```

Floating-point overflow is being suspected. Let us show line 781 of the pre-processed P3_A_3.c file:

[5] The main loop is unrolled 3 times (default).

```
{static NUM _R1;static INT _R2;static BOO _R3; if (
BLBPO ) { _R1=0; _R2=0; if ( B3A3Z09 )  X3A3Z15 =
PADN10 ; else  X3A3Z15 = SYNC_11_E2 ; } else { if (
B3A3Z09  ^ _R3) { if ( B3A3Z09 ) { _R2= SYNC_11_E7 ;
_R1= PADN10 - X3A3Z15 ; } else { _R2= SYNC_11_E4 ; _R1=
SYNC_11_E2 - X3A3Z15 ; } } else { if (_R2>0) _R2=_R2-1;
if ( B3A3Z09 )  X3A3Z15 =( PADN10 -(_R1*_R2/ SYNC_11_E7
)); else  X3A3Z15 =( SYNC_11_E2 -(_R1*_R2/ SYNC_11_E4
)); } } _R3= B3A3Z09 ;}
```

We emphasize the three program locations using bold type. Looking up in the source file, we can see this is an expansion of macro-function `SYNC`:

```
SYNC(11,PADN10,SYNC_11_E2,B3A3Z09,SYNC_11_E4,BLBPO,SYNC
_11_E7,X3A3Z15)
```

From constant definitions and global variable ranges, we can find the values or intervals of every variable occurring in the computation.

No code is generated for “11”, a macro-function occurrence number. `PADN10` is an intermediate variable used in several contexts, so its global interval is of no use. Yet, looking at the source code, we easily notice that this variable is no more than a copy of global variable `X3A3Z01`, the range of which has been computed by Astrée :

```
X3A3Z01 in [-1e+06, 1.41851e+06]
```

The `SYNC_11_E2 float` constant has value `0`. All other inputs of the `SYNC` macro-function are integer constants (with value 17) or Booleans. Astrée has also output an interval for the result of the macro-function:

```
X3A3Z15 in [-3.40282e+38, 3.40282e+38]
```

Unsurprisingly, considering overflow is suspected, that is the largest possible range for a simple-precision floating-point number.

In order to analyse these alarms, we may wonder why the `X3A3Z01` input has so large a range. Looking a few lines backwards in the data-flow, we notice its interval depends upon the analysis by Astrée of another occurrence of the `SYNC` macro-function:

```
SYNC(14,X3A3Z09,SYNC_14_E2,BAPRO2U,SYNC_14_E4,BLBPO,SYN
C_14_E7,X3A3Z01)

INV(1,BLBPO,PADB12)

ET(1,PADB12,BIMPACC,PADB11)

MEM_N(19,X3A3Z01,PADB11,PADN10)

CONF1_I(11,BLSOL,CONF1_11_E2,CONF1_11_E3,BLBPO,PADB15)

INV(2,PADB15,B3A3Z09)

SYNC(11,PADN10,SYNC_11_E2,B3A3Z09,SYNC_11_E4,BLBPO,SYNC
_11_E7,X3A3Z15)
```

We can look up the range of the input of this first `SYNC` in the global invariant:

```
X3A3Z09 in [-4966.87, 6738.46]
```

The analysis of this first SYNC has multiplied the ranges between the X3A3Z09 input and the X3A3Z01 output by a factor of 200. The factor is even higher for the second SYNC. Building more SYNC-based reduced examples, we easily convince ourselves that the analysis of this macro-function causes variable ranges to blow up. The larger the input range, the larger the factor. As a consequence, several occurrences of it in the data-flow will eventually cause alarms, hence maximal simple-precision range, hence more alarms when the outputs are used elsewhere.

For once, Astrée does not implement a dedicated abstract domain to handle this type of code. There is no way the user can make the analysis more precise. This is where support from the tool-provider is needed.

From the reduced example extracted by Airbus, the Astrée development team found out that the frequency of widening steps was too high for this macro-function to be analysed precisely enough. They delivered a new version of the tool implementing new options, for the user to be able to tune widening parameters. In particular, the new `--fewer-widening-steps-in-intervals <k>` option makes Astrée widen unstable interval constraints `k` times less often. On the reduced examples, all alarms disappear with `k=2`.

Improving the precision of the analysis

The `--fewer-widening-steps-in-intervals 2` option is being added to the list of analysis options. All SYNC-related alarms disappear, and we get 327 remaining alarms.

We notice that many calling contexts of the widely used linear two-variable interpolation function G_P give rise to alarms within the source code of this function. Here is an example:

```
g.c:200.8-55::

[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C1_P@7
11:call#P_2_7_1_P@360:call#G_P@977:if@132=false:if@137=
true:if@165=false:if@169=false:loop@177=2:]:

WARN: float division by zero [0, 45]
```

To understand the problem and be able to tune the analysis parameters, one is to build a reduced example from function P_2_7_1_P. The following code is being extracted from the original function:

```
void P_2_7_1_P () {

    PADN13 = fabs(DQM);

    PADN12 = fabs(PHI1F);

    X271Z14 = G_P(PADN13, PADN12, G_50Z_C1, G_50Z_C2, &
    G_50Z_C3 [0][0], & G_50Z_C4 [0][0], ((sizeof(
    G_50Z_C1 )/sizeof(float))-1), (sizeof( G_50Z_C2
    )/sizeof(float))-1);

}
```

where:

- `fabs` returns the module of a floating-point number;
- `DQM`, `PHI1F`, `PADN13`, `PADN12` and `X271Z14` are floating-point numbers;
- `G_50Z_C1`, `G_50Z_C2`, `G_50Z_C3` and `G_50Z_C4` are constant interpolation tables.

`DQM` and `PHI1F` are declared as volatile inputs in the analysis configuration file. Their ranges are extracted from the global invariant computed by Astrée on the full program:

```
DQM in [-37.5559, 37.5559]
PHI1F in [-199.22, 199.22]
```

On this reduced example, we get the same alarms as on the full program. All of them suspect an overflow or a division by zero in the last instruction of the `G_P` function:

```
return(Z2*(Y2-C2[R3])+Z1*(C2[G2]-Y2))/(C2[G2]-C2[R3]);
```

However, when reading the code of `G_P`, one notices that `G2=R3+1` always holds at this point. Moreover, in this reduced example, the interpolation table `G_50Z_C2` is such that `G_50Z_C2[i+1]-G_50Z_C2[i]>1` for any index `i`. Hence, these alarms are false alarms; we must now tune the analysis to get rid of them.

To do so, we have to make Astrée perform a separate analysis for every possible value of `R3`, so it can check no RTE can possibly happen on this code. The way to do so is to ask for a local partitioning on `R3` values between:

- the first program point after which `R3` is no longer written;
- the first program point after which `R3` is no longer read.

Let us implement this, using Astrée partitioning directives:

```
__ASTREE_partition_begin((R3));

G2=R3+1;

Z1=(X1-C1[R2])*(*(C4+(TAILLE_X)*R3+R2)) +
(*(C3+(TAILLE_X+1)*R3+R2));

Z2=(X1-C1[R2])*(*(C4+(TAILLE_X)*G2+R2)) +
(*(C3+(TAILLE_X+1)*G2+R2));

return(Z2*(Y2-C2[R3])+Z1*(C2[G2]-Y2))/(C2[G2]-C2[R3]);

__ASTREE_partition_merge(());
```

This hint makes the alarms disappear on the reduced example.

An even more precise analysis

The analysis of the whole program is being re-launched after the partitioning directives have been inserted in the `G_P` function. All alarms within the G_P function disappear, and many alarms depending directly or indirectly on variables written after a call of function G_P disappear as well: the overall number of alarms boils down to 11.

Here is one of them:

```
PB_9_6.c:610.214-254::
[call#APPLICATION_ENTRY@449:loop@466>=4:call#SEQ_C3_2_P
@501:call#P_B_9_6_P@304:]:
WARN: float division by zero [0, 131070]
```

This alarm occurs within the code of the EANCAL_ANI6_0 macro-function. We use bold type to emphasize the program location which is referred to:

```
#define EANCAL_ANI6_0(NN,_S1) {\
...
if ((_REG_ANI6_PM1 < 0x19) || (_REG_ANI6_PM2 < 0x19))\
  {\
     BOVFANI6BIS0 = TRUE;\
     _S1=9216.0;\
  }\
else\
  {\
     BOVFANI6BIS0 = FALSE;\
     _S1=460800.0/(_REG_ANI6_PM1 + _REG_ANI6_PM2);\
  }\
};
```

where variables _REG_ANI6_PM1 and _REG_ANI6_PM2 of type unsigned int are declared volatile inputs in the configuration file of the analysis. This is obviously not a false alarm.

Results

On this control/command program, it has been possible for a non-expert user from industry to reduce the number of alarms down to zero.

Verifying another kind of avionics programs with Astrée

We will now give an example of alarm analysis on another synchronous program, where the need for specialisation is obvious. This program is not quite a control/command program. It lies on the boundary of the family of programs for which

Astrée has been specialised. Nevertheless, the analyser is still precise on this program, raising few false alarms.

This avionics software product is meant to format data from input media to output media. It is composed of basic functions, and its control flow is defined by constant configuration tables. Unlike the previous program, it performs very limited floating-point computations, but processes many structured data types.

The alarm

The alarm message to be further investigated is the following:

```
mess_conv.c:1058.29-85::

[call#main@8483:call#SQF_Se_Gateway@8501:loop@591=1:cal
l#XMM_Se_Message@612:call#XMC_Se_ReceiveUnrefreshMess@8
98:loop@1046>=2:]:

WARN: unsigned int->unnamed enum conversion range [0,
4294967295] not included in [0, 66]
```

The alarm occurs on line 1058 of the pre-processed `mess_conv.c` file, which we emphasize in bold type below:

```
1: if (IdFctConv < GST_Ct_T_STRUCT_GW_SIZE.NB_GW) {
2:   P_ID_Fct_Conv = (const XMT_Ts_Messages *)
                         &XMC_Ct_T_TABLE_GW[IdFctConv];
3:   Nb_Conv = P_ID_Fct_Conv->Conv_Number;
4:   Index_1ere_Conv = P_ID_Fct_Conv->Begin_List_Index;
5:   if ((Index_1ere_Conv < GST_Ct_T_STRUCT_GW_SIZE.NB_GW_LIST)
        && ((Index_1ere_Conv + (TCD_Td_uInt32) Nb_Conv)
           <= GST_Ct_T_STRUCT_GW_SIZE.NB_GW_LIST)) {
6:     for (cpt_nb_conv = 0;
       cpt_nb_conv < (TCD_Td_uInt32) Nb_Conv;
       cpt_nb_conv++) {
7:       Index_Conv = Index_1ere_Conv + cpt_nb_conv;
8:       P_List_Index_Conv =
            (const XMT_Ts_FunctionsListMessages *)
            &XMC_Ct_T_TABLE_GW_LIST[Index_Conv];
9:       Function_Id =(TED_Te_FunctionName)
                      P_List_Index_Conv->GW_Name_Function;
```

Analysis

Just before this piece of code, the abstract value of `IdFctConv` is `[0, 51]`. Consequently, the abstract value of the `P_ID_Fct_Conv` pointer after instruction 2 is an interval containing more than one value.

It follows that the abstract value of `Nb_Conv` is an interval: `[2, 342]`. Indeed, its lower bound is the minimum value of the `Conv_Number` field for elements of the `XMC_Ct_T_TABLE_GW[]`array with indexes ranging from 0 to 51, which is actually 2. Its upper bound is the maximum value of the same field in the same array slice, which is in fact 342.

Similarly, `Index_1ere_Conv` ranges in `[0, 1117]`.

Let us now consider instruction 6 (the for loop). The loop test expression is `cpt_nb_conv < (TCD_Td_uInt32) Nb_Conv`, and the initial value of the `cpt_nb_conv` loop counter is zero. Because of the abstraction and the interval computed for `Nb_Conv`, this abstract value computed by Astrée for `cpt_nb_conv` in the body of the loop is `[0, 341]`.

Then, using the range computed for `Index_1ere_Conv`, the abstract value for `Index_Conv` after instruction 7 is `[0, 1458]`.

In the concrete semantics of the program, `XMC_Ct_T_TABLE_GW_LIST[]` is a constant table of size 8192. It contains significant data up to index 1118, and all remaining locations have value 2^{32}-1. Instruction 9 uses the `P_List_Index_Conv` pointer computed by instruction 8 to read the `GW_Name_Function` field of the element at index `Index_Conv` in this array.

From the abstract value computed for `Index_Conv` after instruction 7, i.e. `[0, 1458]`, Astrée considers that accesses to the `XMC_Ct_T_TABLE_GW_LIST[]` array beyond index 1118 are possible.

However, we have checked that no real execution of the program computes indexes greater than 1118, thus, the `Function_Id` variable cannot be assigned the $2^{32}–1$ value. Hence, this is a false alarm.

The way to avoid this false alarm

Looking at the `XMC_Ct_T_TABLE_GW[]`array, we notice that all values of variables `Index_1ere_Conv` and `Nb_Conv` are bound by the following relation: `Index_1ere_Conv + Nb_Conv < 1118.` Such a relation is usually precisely caught by the octagon domain of Astrée. We have now to find out why the constraint computed in this case, i.e. `Index_1ere_Conv + Nb_Conv <= 1459`, is not precise enough.

This constraint is computed by Astrée after instruction 4. It is imprecise because the abstract value of `Nb_Conv` (resp. `Index_1ere_Conv`) results from the join of the values of the `Conv_Number` field (resp. `Begin_List_Index`) for all possible values of index `IdFctConv`, i.e. `[0, 51]`.

In order to force Astrée not to compute the above mentioned joins too early, we add partitioning directives into the code.

`__ASTREE_partition_begin((IdFctConv));` is inserted before instruction 2, while the related `__ASTREE_partition_merge(());` is inserted after instruction 4.

These directives make Astrée perform a separate analysis for each individual possible value of `IdFctConv`.

The consequence is that the precise `Index_1ere_Conv + Nb_Conv <= 1119` constraint is now computed by Astrée after instruction 4.

Nevertheless, the alarm does not disappear. At this point, there is no way left for the user to tune the analysis better. That is a typical case in which support from the tool-developers is needed. After a slight improvement by the Astrée team dealing with product reduction between the interval and the octagon abstract domains, this alarm is no longer raised.

Conclusion

The experiments described in this paper show that the Astrée static analyser can be used by engineers from industry to prove the absence of RTE on real avionics programs, and that such non-expert users can meet the zero false alarms objective. Among the reasons for this success, one is to quote the fact that the user does not have to provide Astrée with the invariant of the program to be analysed, only a few clues on how to find it are necessary. The next step for this tool could be its transfer to operational software development teams, which requires an industrial version of Astrée, guaranteeing perennial support.

Moreover, our experience with tools like Astrée gives us the opportunity to sketch a customer-supplier relationship model that could be appropriate for abstract interpretation based tools.

Indeed, the specialisation process of a precise abstract interpretation based analyser makes it necessary for the tool designers to receive accurate information on the targeted type of programs from the end-users. The customer must therefore reveal detailed information about the structure of the targeted programs, their execution model, their dimensions and the type of computations they perform, and provide representative examples.

Furthermore, any change in the analysed program may cause the analyser to become too imprecise for the false alarm reduction process to be industrially feasible. If the case arises, the tool-supplier has to adapt the analyser. As a consequence, the providers of such tools must be prepared to update their products, e.g. add or improve abstract domains, whenever the set of parameters is no longer sufficient to analyse some program of the family precisely, even after the tool specialisation has been performed. This kind of support comes on top of the usual list of services that any tool-provider has to offer.

In brief, a dedicated tool requires a one-to-one customer-supplier relationship.

Acknowledgements. We warmly thank every member of the Astrée team for their constant support.

References

1. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, & X. Rival. A static analyzer for large safety-critical software. In Proc. ACM SIGPLAN'2003 Conf. PLDI, pages 196–207, San Diego, CA, US, 7–14 June 2003. ACM Press.

2 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival. The ASTREE analyser. In ESOP 2005 -- The European Symposium on Programming, M. Sagiv (editor), Lecture Notes in Computer Science 3444, pp. 21--30, 2--10 April 2005, Edinburgh, (c) Springer.

3 Patrick Cousot. Interprétation abstraite. Technique et Science Informatique, Vol. 19, Nb 1-2-3. Janvier 2000, Hermès, Paris, France. pp. 155--164.

4 Patrick Cousot. Abstract Interpretation Based Formal Methods and Future Challenges. In Informatics, 10 Years Back - 10 Years Ahead, R. Wilhelm (Ed.), Lecture Notes in Computer Science 2000, pp. 138--156, 2001.

5 Patrick Cousot & Radhia Cousot. Basic Concepts of Abstract Interpretation. In Building the Information Society, R. Jacquard (Ed.), Kluwer Academic Publishers, pp. 359--366, 2004.

6 Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux & Xavier Rival. Varieties of Static Analyzers: A Comparison with ASTREE. 2007.

7 Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst-case execution time of an avionics program by abstract interpretation. In Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, pages 21-24, 2005.

8 Eric Goubault, Matthieu Martel, and Sylvie Putot, Static Analysis-Based Validation of Floating-Point Computations, Proceedings of Dagstuhl Seminar Numerical Software with Result Verification 2003, LNCS volume 2991, pp 306-313.