Experimental Assessment of Astrée on Safety-Critical Avionics Software

Jean Souyris and David Delmas

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

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

Abstract. Astrée is a parametric Abstract Interpretation based static analyser that aims at proving the absence of RTE (Run-Time Errors) in control programs written in C. Such properties are clearly safety properties since the behaviour of a C program is undefined after a RTE. When it analyses a program of the class for which it is specialised, Astrée is far more precise than general purpose static analysers. Nevertheless, for safety and industrial reasons, the small number of false alarms first produced by the tool must be reduced down to zero by a new fine tuned analysis. Through the description of experiments made on real programs, the paper shows how Abstract Interpretation based static analysis will contribute to the safety of avionics programs and how a user from industry can achieve the false alarm reduction process via a dedicated method.

Keywords: avionics software, safety, verification, Abstract Interpretation, static analysis, run-time errors, Astrée.

Introduction

Safety-Critical avionics systems are composed of sensors, actuators, hydraulic pipes, electrical cables and computers. All these components must contribute to system safety. The contribution of hardware components to the safety objectives is being assessed using well-known techniques, that unfortunately do not apply to software components. Avionics software is considered "good" provided its development process is DO178B compliant, "good" meaning that the program implements its specification safely.

A way to view software verification from a safety-oriented perspective could be to get some safety properties as inputs of the software development process and to demonstrate that these properties hold. How do we perform such a demonstration?

Basically, software verification involves three kinds of techniques: testing, intellectual analyses and formal verification. The first one, i.e., testing, is the most popular. It has the advantage of being based on actual executions of the program under test.

Moreover, tests can be performed in an environment very close to the operational one. Nevertheless, testing is not sound. Indeed, even during the heaviest possible test campaign, it is impossible to verify every possible execution of the program under test. The second verification technique, i.e., intellectual analyses, is often used as a complement to testing. But this not really a proof that a property holds. The last technique, i.e., formal verification, aims at formalising the proof that a program satisfies some properties. But "formal" is not enough, for obvious industrial reasons it must also be automatic.

Ideally, safety properties of software should be demonstrated formally by automatic tools. Indeed, when any sound formal technique allows to prove a property, this means there exists no execution of the program that falsifies the property. We recall that, by principle, testing is unable to do so.

Existing formal proof techniques are Model-Checking, Theorem Proving and Abstract Interpretation ([3], [4], [5]) based Static Analysis. The items to be verified being final products, i.e. source or binary code, Model-Checking is not considered relevant. For different reasons, the use of Theorem Proving to verify safety properties on complete real-size programs seems far beyond software engineers capabilities.

Currently, to prove safety properties on real-life safety-critical programs, good candidates are static analysers aiming at proving a specific class of properties, i.e., WCET (Worst-Case Execution Time), Stack analysis, Floating-point calculus, Run-Time Errors, Memory usage properties. Indeed, Abstract Interpretation based static analysers now do scale, i.e., they are able to analyse complete safety-critical programs. They are a first step towards the proof of almost all safety properties of software. A second big step will be the proof of so-called "user-defined" safety properties, as opposed to the above analysers, in which the properties are "hard-coded. For instance, Fluctuat C (CEA, French nuclear research centre) does not analyse C programs in order to prove properties submitted by the user, but to compute a safe approximation of the rounding errors in floating-point calculus and stability properties.

The rest of the paper is focused on the proof of absence of Run-Time Errors in safety-critical avionics programs. Such a property is clearly a safety objective for software since after most RTE, the behaviour of a program is undefined. Indeed, although some RTE raise an interrupt, e.g., floating-point overflow, other errors like *out-of-bounds array access* or *dereferencing an invalid pointer*, might let the erroneous program do very bad things before any failure is detected.

This paper is essentially an industrial experience report on the use of the Astrée Abstract Interpretation based static analyser.

Astrée

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

The absence of RTE has been defined in [2, §2]: "*The absence of run-time 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).*"

Notion of false alarm. To prove the absence of Run-Time Errors, Astrée computes an abstract invariant that safely represents (thanks to the Abstract Interpretation theoretical framework) a superset of all possible executions of the analysed program. Actually, the first abstraction Astrée implements consists in computing an over-approximation of the reachable states of the analysed program, without storing the execution traces, i.e., the sequences of states. Intuitively, a program state is a pair (program location, memory state). The abstract invariant is the set of "per program location abstract invariants", otherwise called local invariants. A local invariant is defined for a particular location in the program and it is an over-approximation of all possible memory states at that point.

By definition of Run-Time Errors, Astrée knows at which program locations an RTE might occur. Therefore, at each such location in the program, the diagnosis part of Astrée checks if the RTE condition might be satisfied by the local invariant.

Unfortunately, as the abstract invariant is an over-approximation of all possible reachable states and, consequently, of all possible executions of the analysed program, any RTE condition, like "division by zero", might be satisfied by "false" executions only. By "false executions" we mean: executions that cannot occur in reality. When it is the case, the alarm is called a "false alarm".

Zero false alarm is mandatory, for industrial reasons. The main result of a Astrée is a list of alarms raised from the abstract invariant it computes. For each alarm, the user is informed about the program location at which the corresponding error could happen during real execution. Each alarm also comes with additional contextual information. Unfortunately, there is no simple and direct way to state whether an alarm is true or false. For each alarm, the user must look at the analysed program and analyse it backwards from the location at which the Run-Time Error might occur in a real execution. This user backward analysis aims at deciding whether it exists a run-time scenario causing the error or not.

By principle of Abstract Interpretation, if all the alarms raised by an analysis, i.e., one run of an analyser, are false, then the proof of absence of Run-Time Errors is completed. The problem here is that such a human analysis is error-prone and time-consuming.

Therefore, the fewer false alarms an Abstract Interpretation based static analyser like Astrée produces, the better for the safety of the demonstration that the analysed program cannot produce Run-Time Errors.

An analyser is said more precise than another one if it produces less false alarms for the same analysed program. The maximum precision is reached when all alarms, if any, are true.

Specialisation of Astrée

To achieve the above mentioned "zero false alarm" objective, an Abstract Interpretation based static analyser needs to be "specialised". Here, specialisation deals with the

ability to capture[1] precise enough abstract invariants by taking into account the kind of computations performed by the analysed program. The "per program family specialisation" is performed at design time by the developers of the analyser. As explained in [1, §3.1], it is obtained by refinement of a previous more general-purpose analyser, making the new analyser more precise for any program of a targeted class of programs.

One example of the specialisation of Astrée has consisted in turning it able to precisely – and safely, of course – over-approximate the output stream of floating-point values produced by digital filters.

Specialisation to synchronous avionics programs. Such programs perform control/command operations and are those on which Astrée is the most precise, i.e., on which it produces the smallest number of false alarms. In avionics, these programs are automatically produced from SCADE™ (formerly SAO) detailed specifications and have very peculiar computational characteristics. The following four paragraphs describe the most important specialisation issues.

Synchronous programs. First, their scheduling is fully statically defined. It means that although they are made of parallel tasks, their serialisation is performed at design time, i.e., the execution of any piece of code only depends on the date at which it must execute. Such a date is given by the program's clock. Mainly for that reason, these programs are said to be "synchronous". There is no event driven task, no pre-emption, etc. For automatic analysis, the first important consequence in terms of precision, is that it is bounded by the maximum value of the clock, i.e., the maximum duration of an "aircraft mission". Consequently, the "clock abstract domain" was the first due to the specialisation.

Linear control flow – Intensive use of Booleans. Another characteristic of these automatically produced programs stands in their extremely linear code structure. As any code instruction belongs to a small library component, conditionals, i.e., "`if(){}else{};`", or loops, e.g., "`while() {};`" are necessarily very local and contain a very limited number of instructions. For the loops, the consequence is that they are quite easy to analyse precisely. With respect to the conditionals, the consequence of their extreme locality is that the program contains a lot of Booleans. Typical scheme is the computation of a Boolean value as the result of a condition on floating-point values, and then the test of this Boolean value elsewhere in the code for performing appropriate actions. In "classical" programming, this is done by testing the condition in an "`if(){}else{};`" statement and performing the actions in the scope of the first or second pair of "`{}`". For a static analyser, the two situations are not similar. Indeed, without dedicated "abstract domain", there is a severe loss of precision when the control flow is "encoded in Booleans", like in the first of the two situations described above. To avoid this kind of loss of precision, Astrée contains an Abstract domain called the "Boolean tree domain".

[1] In the Abstract Interpretation framework, this capture is performed via so-called "abstract domains".

Floating-point calculus. Synchronous control/command avionics programs use a large number of floating-point variables (over 10,000): inputs, state variables and outputs. The operations performed for computing the state variables from inputs and the output variables from the state variables at each clock tick (every 10ms, for instance) represent thousands of lines of C code and are subject to accumulating rounding errors. To be sound, Astrée abstracts each single floating-point operation in such a way that the computed set of values includes all possible rounding errors. It is a real challenge, also because the proper tool, i.e., Astrée, makes its own computations by using floating-point arithmetic.

Digital filters. As mentioned above, the control/command program computations are those induced by the control theory. Beyond the arithmetic operations, typical basic operators are delays, derivation, integration, first and second order digital filters. The very first versions of Astrée did not take into account the last two specificities. Consequently, the abstract domains existing at that time, e.g., intervals, clock, octagons, were not able to compute tight shapes of values for the outputs of the filters. As there are cascades of such filters, a lot of false overflows of floating-point values were raised by Astrée. The implementation of the Filter domain led to the suppression of all false alarms consecutive to the excessive over-approximation of the filter outputs.

Last step in specialisation: fine tuning by the user. In spite of its dedicated domains, Astrée might produce a few false alarms, which it is better to suppress by launching the tool again with a new set of "parameters", if possible.

In fact, after the user is convinced that a given alarm is false, next step is to understand why the analyser could not compute an invariant precise enough to avoid this false alarm. Since abstraction consists in not taking into account some characteristics (or properties) of the real executions of the program, it might be the case that none of the Astrée abstract domains is able to capture the program properties that would have avoided the false alarm. In this case, there is no other way than asking the Astrée team to improve the tool.

On the other hand, it might be that the domains able to capture the missing information are there, but their complexity is too high to be applied a priori uniformly to the whole program. In this case, some domains like the octagons or the domain of partitions can be locally used by means of Astrée directives to be inserted in the analysed program. This final "per program specialisation process" matches the adaptation by parameterisation described in [1, §3.2]. It is sound, provided the user only inputs facts about the environment of the program, and hints (to be checked by the tool) on how to improve the precision.

An engineering method to handle Astrée outputs

The first analysis of a complete real-world application usually floods the user with many alarms, so one hardly knows how to get started. Let us try to define a methodical approach to deal with these alarms.

The need for full alarm investigation

Exhaustive alarm analysis is absolutely necessary: every single alarm must either disappear by means of a more precise automated analysis, or be proved by the user to be impossible in the real environment of the program. Indeed, every time Astrée signals an alarm, it assumes the execution of the analysed program to stop whenever the precondition of the alarm is true. As a consequence, any true alarm may "hide" more alarms. Let us give a simple example with variable `X` of type `int` in interval `[0,10]`:

```
Y = 1/X;

Z = 1/X;
```

Astrée will report a warning on line 1 (possible division by zero), but not on line 2: since the execution is assumed to stop whenever `X` happens to equal zero on line 1, line 2 cannot be executed unless $X \neq 0$. As a consequence, Astrée assumes `X` to be in interval `[1,10]` from line 2.

How to read an alarm message

Here is an example of alarm reported by Astrée:

```
bnrvec.c:127.2-10::

    [call#APPLICATION_ENTRY@449:loop@466>=4:

     call#SEQ_C3_4_P@543:

     call#P_9_1_6_P@247:

     call#BNRVEC_P@442:]:

WARN: float->signed int conversion range [-6999258112,
6991659520] not included in [-2147483648, 2147483647]
```

Let us explain how this message is to be understood. Astrée warns that some `float->signed int` conversion may cause an integer overflow. It points precisely to the operation that may cause such a RTE: line 127 of the (pre-processed) file `bnrvec.c`, between columns 2 and 10[2]:

```
R3=E1*A3;
```

where R3 has type `int`, while `E1` and `A3` have type `float`.

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

All other information describe the stack. The analysis entry point[3] is the `APPLICATION_ENTRY` function, defined line 449 of file `appl_task.c`. This function contains a loop at 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_C3_4_P` is called on line 543 of file `appl_task.c`;
- `SEQ_C3_4_P` calls function `P_9_1_6_P` on line 247 of file `seq_c3_4.c`;
- `P_9_1_6_P` calls function `BNRVEC_P on` line 442 of file `P9_1_6.c`;
- the floating-point product of the operands `E1` and `A3` ranges from `-6999258112` to `6991659520`.

However, this interval is not a subset of `[-2147483648, 2147483647]`, hence contains values that cannot fit into a 32 bit signed integer.

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.

How to deal with alarm investigation

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

To investigate an alarm, 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 pointed to by an alarm, one may extract the corresponding interval, which is a sound over-approximation of the range of this variable.

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 stream language such as SAO, SCADE™ or Simulink™. The engineering user can indeed label every arrow representing a global variable with an interval, going backwards from the alarm point. The origin of the problem is usually found when some abrupt unforeseen increase in variable ranges is detected.

At this point, we know whether the alarm originated in some “random code” or in some definite specialized 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.

Once we have probed into 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;

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

- 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.

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

- adding directives in the source, to help Astrée increase the precision of its analysis;
- tuning the list of analysis options;
- changing the parameters of the example 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 safety-critical control/command program with Astrée

We will now present experiments realised on a periodic synchronous control/command program developed at Airbus. Most of its C source code was 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 great challenge. A general-purpose analyser would not be suitable.

Fortunately, Astrée has been specialised 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.

The very first analysis produces 467 alarms. With this program, after options related to loop unrolling and widening parameters have been tuned, we get 327 remaining alarms to be further analysed by the industrial user, in order to decide whether they are false alarms or not.

Analyzing a false alarm

For instance, in this program, 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 are two examples:

```
g.c:191.8-55::

[call#APPLICATION_ENTRY@449:loop@466=1:call#SEQ_C1_P@49
8:call#P_2_7_1_P@360:call#G_P@967:if@119=false:if@124=f
alse:loop@132=2:if@152=false:if@156=false:loop@164=2:]:

WARN: float arithmetic range ([-inf, inf]

g.c:191.8-55::

[call#APPLICATION_ENTRY@449:loop@466=1:call#SEQ_C3_3_P@
522:call#P_2_7_4_P@202:call#G_P@694:if@119=false:if@124
=false:loop@132=3:if@152=false:if@156=false:loop@164=2:
]:

WARN: float arithmetic range ([-inf, inf]
```

To understand the problem and be able to tune the analysis parameters, one is to build a reduced example from one of these contexts, say P_2_7_1_P. The following code is being extracted from the original P_2_7_1_P 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, 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 it, 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. It has the same effect on the complete program. Besides, 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.

Analysing a true alarm

Let us describe one of the (few) remaining alarms:

```
P8_4_1.c:506.0-38::

[call#APPLICATION_ENTRY@449:loop@466=1:call#SEQ_C4_7_P@
609:call#P_8_4_1_P@152:]:

WARN: implicit unsigned int->signed int conversion
range {3090427953} not included in [-2147483648,
2147483647]
```

This alarm points to the following (pre-processed) instruction:

```
MODULE_NUMBER= 0x80000000 + 0x38343031 ;
```

where `MODULE_NUMBER` has type `int`.

This code is fairly straightforward, in the sense that its execution does not depend on any input. Such a context allows for a very precise analysis, which is why Astrée does not report an interval, but a single possible value for the result of the addition (`3090427953`).

The reason for this alarm is the following: the ISO/IEC 9899 international standard, describing the semantics of integer constants, specifies the type of an integer constant to be the first of the corresponding list in which its value can be represented. For a hexadecimal constant without a suffix, the list is:

1. int
2. unsigned int

3. long int
4. unsigned long int
5. unsigned long long int
6. long long int

As `0x80000000 = 2`^{31} does not fit into a 32 bit `int`, this constant must have type `unsigned int`. `0x38343031>0`, hence the result of the addition has type `unsigned int` and its value (`3090427953`) is outside the range of signed 32 bit integers, i.e. `[-2147483648, 2147483647]`.

Most compilers will know how to deal with such a conversion, however, Astrée soundly and genuinely warns that the semantics of this code is unspecified by ISO/IEC 9899. Indeed, its behaviour is implementation-defined.

One way to fix it (without using type `long long int`, which Astrée does not support) is to replace `0x80000000` by `(-2147483647-1)`, which is probably what the programmer had in mind in the first place.

Results

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

Conclusion and future work

Several theoretical papers, such as [1] and [2], already explained the solutions to the scientific and technological issues which the Astrée team had to deal with to demonstrate that an Abstract Interpretation based static analyser, Astrée, can analyse real-size industrial programs and be extremely precise.

But it was a point of view of scientists.

In our paper, we have tried to present an industrial point of view. The challenge for software engineers from industry was to be able to use the analyser Astrée on a real-size program without altering it before the analysis. We have succeeded, as shown in the previous sections, thanks to two kinds of skills. First, being software engineers, we were not afraid of reading code. Secondly, we have had the confirmation that, to use an Abstract Interpretation based static analyser, it is mandatory to know enough about its underlying principles, and that it was within the reach of software engineers.

A proof that a safety-critical avionics program is free from Run-Time Errors is obviously not a proof that this program is safe. Although we have not described it in this paper, we are also addressing complementary safety objectives such as WCET assessment [6], safe memory use, precision and stability of floating-point computations [7]. To meet all these objectives is a first step towards the safety assessment of a program.

Other future work will address the proof of absence of RTE on multi-threaded asynchronous programs.

References

1. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and 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 ASTRE'E 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 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.

7 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.