Astrée: Proving the Absence of Runtime Errors

Daniel Kästner[1], Stephan Wilhelm[1], Stefana Nenova[1],
Patrick Cousot[†], Radhia Cousot[†], Jérôme Feret[†],
Laurent Mauborgne[†], Antoine Miné[†], Xavier Rival[†],

1: AbsInt Angewandte Informatik GmbH. Science Park 1, D-66123 Saarbrücken, Germany
2: École Normale Supérieure. 45, rue d'Ulm, F-75230 Paris, France

Abstract: Safety-critical embedded software has to satisfy stringent quality requirements. Testing and validation consumes a large – and growing – fraction of development cost. The last years have seen the emergence of semantics-based static analysis tools in various application areas, from runtime error analysis to worst-case execution time prediction. Their appeal is that they have the potential to reduce testing effort while providing 100% coverage, thus enhancing safety. Static runtime error analysis is applicable to large industry-scale projects and produces a list of definite runtime errors and of potential runtime errors which might be true errors or false alarms. In the past, often only the definite errors were fixed because manually inspecting each alarm was too time-consuming due to a large number of false alarms. Therefore no proof of the absence of runtime errors could be given. In this article the parameterizable static analyzer Astrée is presented. By specialization and parameterization Astrée can be adapted to the software under analysis. This enables Astrée to efficiently compute precise results. Astrée has successfully been used to analyze large-scale safety-critical avionics software with zero false alarms.

Keywords: Proof of absence of runtime errors, abstract interpretation, static C code analysis, DO-178B, ISO-26262

1. Introduction

Safety-critical embedded software has to satisfy stringent quality requirements. A system failure or malfunction can have severe consequences and cause high costs. Testing and validation consumes a large – and growing – fraction of development cost. Thus, developers face the challenge of ensuring the correct functioning of the software, but this has to be done with reasonable effort.

In the avionics, automotive, and healthcare industries static analyzers based on abstract interpretation have increasingly been used to validate program properties of safety-critical software. The results are only computed from the software structure without actually running the software under analysis. The results thus obtained hold for any possible input scenario and any possible program execution. Examples are tools for computing the worst-case execution time [28, 13] or the maximal stack usage of tasks [12], the accuracy of floating-point computations [20], and the absence of run-time errors [2, 6, 8]. Modern semantics-based static analyzers scale well and support the analysis of large industrial software projects.

This article focuses on a certain class of errors, the so-called runtime errors. Examples for runtime errors are floating-point overflows, array bound violations, or invalid pointer accesses. Runtime errors lead to undefined program behavior; the consequences range from erroneous program behavior to wholesale crashes. A well-known example for the possible effects of runtime errors is the explosion of the Ariane 5 rocket on its maiden flight in 1996 [19].

An important goal when developing critical software is to prove that no such errors can occur at runtime. Software testing can be used to detect errors, but since usually no complete test coverage can be achieved, it cannot provide guarantees. Semantics-based static analysis allows to derive such guarantees even for large software projects. The success of static analysis is based on the fact that safe overapproximations of program semantics can be computed. This means that the results of such analyses will be either "(i) statement x will not cause an error", or "(ii) statement x may cause an error". In the first case, the user can rely on the absence of errors, in the second case either an error has been found, or there is a false alarm. This imprecision allows sound static analyzers to compute results in acceptable time, even for large software projects. Nevertheless the results are reliable, i.e., the analysis will only err on the safe side: if the analyzer does not detect any error, the absence of errors has been proven - the coverage is 100%.

Each alarm has to be manually investigated to determine whether there is an error that has to be

corrected, or whether it was just a false alarm. If all the alarms raised by an analysis have been proven to be false, then the proof of absence of runtime errors is completed. This could be checked manually, but the problem is that such a human analysis is error-prone and time consuming, especially since there might be interdependencies between the false alarms. If the analyzer does not report any alarm the absence of runtime errors is automatically proven by the analyzer run. Therefore the ideal solution is to enable the analyzer to finish the analysis with zero alarms.

To that end, it is important that the analyzer is precise, i.e., produces only few false alarms without particular user interaction. This can only be achieved by a tool that can be specialized to a class of properties for a family of programs. Additionally the analyzer must be parametric enough for the user to be able to fine tune the analysis of any particular program of the family. General software tools not amenable to specialization and parametrization usually report a large number of false alarms. That is the reason why in industry such tools are only used to detect runtime errors, and not to prove their absence. The analyzer should also provide flexible annotation mechanisms for users to communicate external knowledge to the analyzer. Only by a combination of high analyzer precision and support for semantic annotations the goal of zero false alarms can be achieved.

In our article we focus on the static analyzer Astrée (*Analyseur statique de logiciels temps-réel embarqués*) [1], which originates from the École Normale Supérieure [10]. Since February 2009 Astrée is commercially available and is now developed and distributed by AbsInt under license of CNRS/ENS. Astrée has been specifically designed to meet the above mentioned requirements: it produces only a small number of false alarms for control/command programs written in C, and provides the user with enough options and directives to help reduce this number down to zero. Astrée has been successfully used to analyze industrial Airbus avionics software [27]. We give an overview of the structure of Astrée and describe how developers can use it to achieve the goal of zero false alarms and thus efficiently validate the absence of run-time errors.

2. Static Analyzers

Static analyzers compute their results only from the program structure by inspecting the source code or binary code, but without actually executing it.

Static analyzers are sometimes understood as including *style checkers* looking for deviations from coding style rules, like the MISRA guidelines prescribed by the Motor Industry Software Reliability Association [24]. Such style checkers are usually not "semantics-based", and thus cannot check for correct runtime behavior.

Furthermore static analyzers can be categorized in sound vs. unsound analyzers. A program analyzer is *unsound* when it can omit to signal an error that may appear at runtime in some execution environment. Unsound analyzers are *bug hunters* or *bug finders* aiming at finding some of the bugs in a well-defined class. Their main defect is unreliability, being subject to *false negatives* thus claiming that they can no longer find any bug while many may be left in the considered class. Unsoundness can be caused e.g., by skipping program parts which are hard to analyze, ignoring some types of errors, disregarding some runtime executions, or adopting a simplified program semantics. Example tools from this class are ESC Java [18], Coverity CMC [11], Klocwork K7 [16], PRE-fast [25], or Splint [17]. A more comprehensive overview is found in [6].

Such unsound approaches are all excluded in Astrée. Astrée is a *bug eradicator* in that sense that all bugs from a well-defined class, i.e., runtime errors, are found. Another tool from this class is Polyspace Verifier [8]. More precisely, Astrée is a sound semantics-based static analyzer based on Abstract Interpretation.

2.1 Abstract Interpretation

Static analyzers compute invariants for all program points by fixed point iteration over the program structure or the control flow graph. The theory of abstract interpretation [5] offers a semantics-based methodology for static program analysis. The concrete semantics is mapped to an abstract semantics by abstraction functions. While most interesting program properties are undecidable in the concrete semantics, the abstract semantics can be chosen to be computable. The static analysis is computed with respect to that abstract semantics. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. By skilful definition of the abstract domains a suitable trade-off between precision and efficiency can be obtained.

Abstract interpretation supports formal correctness proofs: it can be proven that an analysis will terminate and that it computes an overapproximation of the concrete semantics, i.e., that the analysis results are sound. A static runtime error analysis is

called *sound* if it never omits to signal an error that can appear in some execution environment. If no potential error is signalled, definitely no runtime error can occur. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm. Thus, imprecision can occur, but only on the *safe* side; it can never happen that there is an error from the error class under analysis which is not reported.

3. Astrée— Design and Overview

Astrée [2] is a parametric static analyzer based on abstract interpretation that aims at proving the absence of run-time errors of programs written in C, according to "ISO/IEC 9899:1999 (E)" (C99 standard) [3]. Astrée analyzes structured C programs, with complex memory usage, but without dynamic memory allocation and without recursion. This encompasses many embedded programs as found in earth transportation, nuclear energy, medical instrumentation, aeronautic, and aerospace applications, in particular synchronous control/command programs such as electronic flight control. The errors that are currently reported are: out-of-bound array accesses, integer division by zero, floating point overflows and invalid operations (resulting in IEEE floating values Inf and NaN), integer arithmetics wrap around behavior (occurring mainly in overflows), and casts that result in wrap around operations (when the target type is too small to contain a value), and violations of arbitrary user defined assertions on the software. In addition, Astrée points out unanalyzed (unreachable) code and warns about possibly non-terminating code.

Providing a rigorous formal semantics of C programs as a basis for static analyzers is extremely difficult since there is considerable leeway for implementations. As an example the source semantics is undefined after a runtime error. A write access via an invalid pointer or an out-of-bounds array index can corrupt memory. The result of such a program execution cannot be statically determined. Therefore, Astrée distinguishes between two different types of runtime errors [6]: runtime errors corresponding to *undefined* behaviors, and runtime errors corresponding to *unspecified* but predictable behaviors. They differ in the consequences of an actually occurring error, but in both cases Astrée will go on with an over-approximation of the considered executions and it will definitely discover all errors after a false alarm.

3.1 Handling Undefined Behavior

For runtime errors corresponding to *undefined behaviors* Astrée produces an alarm and continues the analysis only for concrete program executions where the error does not occur. Examples for this class of runtime errors are integer division by zero, floating-point overflow, and invalid array or pointer accesses. E.g., the following program:

```
#include <stdio.h>
int main() {
  int n, T[0];
  n = 2147483647;
  printf("n=%i, T[n]=%i\n", n, T[n]);
}
```

produces different results on different machines:

```
n=2147483647, T[n]=2147483647 (PPC Mac)
n=2147483647, T[n]=-1208492044 (Intel Mac)
n=2147483647, T[n]=-135294988 (32-bit PC Intel)
Bus error (64-bit PC Intel)
```

Since it is not predictable what will happen after such an error, Astrée does not attempt to make any prediction. Instead, the analyzer assumes that program execution stops after the error and subsequently only considers scenarios where the error did not occur. In cases where an error will definitely occur in some execution context, Astrée reports a *definite alarm* and terminates the analysis for this context.

3.2 Handling Unspecified Behavior

For runtime errors corresponding to *unspecified* but predictable behavior Astrée emits an alarm and considers all possible outcomes during the rest of the analysis. An example for this are integer overflows for which the actual computations are different from the intended mathematical meaning. Let us consider the following example[1]:

```
1: void main() {
2:   int i;
3:   if (i<0) {
4:     i = -i;
5:   }
6:   __ASTREE_assert((i!=-1));
7: }
```

Astrée reports an alarm because of the potential overflow in line 4, but can verify the assertion in line 6. In fact, Astrée is able to represent the precise result of the computation for a 32-bit 2's complement architecture which is in the non convex set $[0, 2147483647] \cup \{-2147483648\}$ (cf. Sec.4).

3.3 The Zero-Alarm Goal

[1] We assume that two's complement hardware has been configured in the ABI settings of Astrée (cf. Sec. 5).

For industrial use an important goal is to produce the fewest possible number of false alarms. An automatic proof of the absence of runtime errors is only possible if the analysis terminates without any alarm – in the terminology of Polyspace Verifier [8] the entire code must be green. Any alarm has to be manually checked by the developers – and this manual effort should be as low as possible. If there is a true error, it has to be fixed and the analysis has to be restarted. A false alarm can possibly be eliminated by a suitable parameterization of Astrée (cf. Sec. 5). If the error cannot occur due to certain preconditions which are not known to Astrée, they can be made available to Astrée via dedicated annotations. These annotations make the side conditions explicit which have to be satisfied for a correct program execution.

Thus it is highly important that an analyzer supplies enough information for users to understand the cause of an alarm and to provide explicit formal means for suppressing false alarms. Of course for keeping the initial number of false alarms low, a high analysis precision is mandatory.

3.4 Alarm Analysis

Consider the following C program:

```
 1:  #define BASE 0x80000000
 2:  #define OFFSET 0x38343031
 3:  volatile int SwitchPosition;
 4:
 5:  int main()
 6:  {
 7:    /*...*/
 8:    int MODULE1 = BASE + OFFSET;
 9:    /*...*/
10:    char sp = SwitchPosition;
11:  }
```

Astrée emits two alarms for potential runtime errors, an arithmetic overflow in line 8 and in line 10, respectively. The code producing the alarm is marked in red (cf. Fig.1).

For each alarm the user has to check whether it can occur in a real program execution. The alarm from line 8 is caused by a true runtime error due to the handling of hexadecimal constants according to the C99 standard. The type of such a constant is assumed to be `int` if it fits into the `signed int` range, otherwise `unsigned int`, etc. In that case the type `unsigned int` is assumed since `0x80000000`$= 2^{31}$ which does not fit into a 32-bit `signed int`, but into a 32-bit `unsigned int`. Since `0x38343031>0` the result of the addition is of type `unsigned int` and is outside the `signed int` range. A possible fix is to declare `MODULE1` as `unsigned int`.

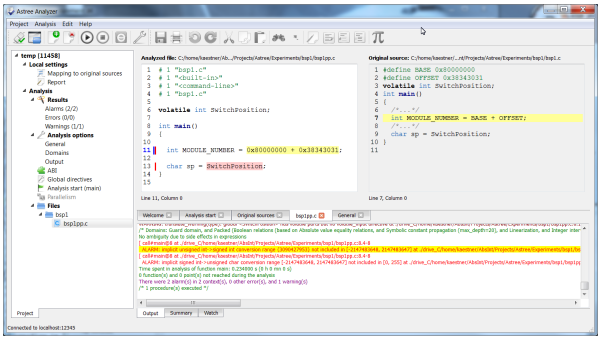

Figure 1: Astrée GUI with highlighted error location.

If the code has been implemented under the assumption that the modelled switch can take 8 different positions, the second alarm is a false alarm. The value of `SwitchPosition` is indeed volatile, but can only take values from `{0,...,7}`. This information can be made available to Astrée by the directive `__ASTREE_volatile_input((SwitchPosition, [0,7]))`. If the input program changes over time, the validity of such annotations always has to be explicitly checked. When running Astrée on the modified program, no alarms are reported.

Astrée explicitly supports investigating alarms in order to understand the reasons for them to occur. When clicking at an alarm message the corresponding code location is highlighted in the original and preprocessed source code. Alarms can be grouped by source code locations, and all contexts in which an alarm occurs are listed. Alarm contexts can be interactively explored: all parents in the call stack, relevant loop iterations or conditional statements can be visited per mouse click, and the computed value ranges of variables can be displayed for all abstract domains (cf. Fig. 2). Inversely, clicking on the

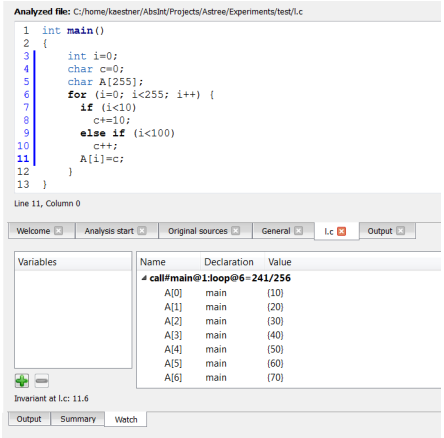

Figure 2: Astrée variable ranges display.

source code location for which an alarm has been produced repositions the focus of the output window to show the corresponding alarm message. In the output window alarm locations are collected in the order in which they are reached by the analyzer. This is very helpful for alarm investigation since fixing one alarm usually causes several subsequent alarms to disappear. The call graph of the software under analysis is visualized taking function pointer calls into account; an example call graph is shown in Fig. 3.

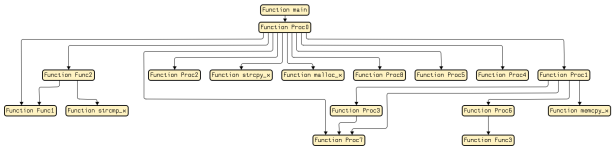

Figure 3: Astrée Call graph visualization.

4. Astrée Domains

Astrée offers a variety of predefined abstract domains. In this section the most important ones are shortly summarized and illustrated with examples[2]. The *memory abstract domain* is an abstraction of sets of concrete memory states whose elements, called abstract environments, map variables to abstract cells. An abstract cell can represent one or several scalar variables, an expanded array cell, a folded array cell (cf. Sec. 5.1), or a structure field. The memory domain empowers Astrée to exactly analyze pointer arithmetics, as well as struct and union manipulations. In the following example, Astrée can prove both assertions to be correct:

```
typedef struct _x {
  unsigned int a: 1;
  unsigned int b: 1;
} bit ;

void main( ){
  bit z ;
  z.b = 0 ;
  z.a = 1 ;
  __ASTREE_assert((z.a == 1));
}
```

Pointers are supported both for functions and for data; efficiency and precision are enhanced by several domains covering symbolic information. Thus, Astrée reports zero alarms on:

```
struct s { struct s* next; int data; } ;
struct s A[100];
void main()
{
  int i=0;
  struct s *ptr;
  for (i=0; i<199; i++) {
    if (i<99)
      A[i].next=&(A[i+1]);
    else
      A[i-99].data=i;
  }
  A[99].next=0; A[99].data=99;
  ptr = &(A[0]);
  while (ptr != 0) {
    ptr = ptr->next;
  }
}
```

The *interval domain* approximates variable values by intervals. The octagon domain [23] covers relations of the form $x \pm y \leq c$ for variables x and y and constants c. Compared to the full polyhedron domain [4] covering convex polyhedra of the form $\sum_{i=1}^{N} \alpha_i x_i \leq c$ it has the advantage that is significantly faster and supports floating-point arithmetics [2]. In the example program below the relation between X and Y is automatically discovered so that Astrée can show the absence of overflows and can prove the assertion that X<=Y.

```
void main()
{
  int X=100000,Y=1000000;
  while (X >= 0) {
    X--;Y--;
  }
  __ASTREE_assert((X<=Y));
}
```

The *modulo-interval domain* enables a precise analysis of overflows on two's complement hardware. This is especially useful for code automatically generated by dSPACE TargetLink [9] with the "compute-through-overflow" technique [15]. Consider the following example:

```
short x,y;
__ASTREE_volatile_input((x, [-1,1]));
__ASTREE_volatile_input((y, [-1,1]));
void main()
{
  short z;
  z = (short)((unsigned short)x +
              (unsigned short)y);
}
```

Astrée emits three alarms because of overflows on explicit typecasts but computes the correct value range for z, i.e., z∈[-2,2][3].

Floating-point computations are precisely modelled while keeping track of possible rounding errors

[2]The analysis time for all examples shown was below 1 sec on a 2.4 GHz Centrino 2 laptop.

[3]To support overflow-safe code generators, Astrée can be configured not to emit alarms for explicit typecasts.

[22]. Most static analyzers either do not handle floats or handle them incorrectly because they are based on mathematical properties of real numbers not valid for floats. For example $(x+a)-(x-a) = 2a$ is not valid for floats:

```
#include <stdio.h>
int main () {
  double x; float a,y,z,r1,r2;
  a = 1.0; x = 1125899973951488.0;
  y = x+a; z = x-a;
  r1 = y - z; r2 = 2*a;
  printf("(x+a)-(x-a) = %f\n", r1);
  printf("2a          = %f\n", r2);
}
```

The output produced is:

```
(x+a)-(x-a) = 134217728.0000
2a = 2.0000
```

The double value `x` is just in the middle of two consecutive floating-point numbers to which, respectively, `x-1` and `x+1` will be rounded in round-to-nearest mode. Astrée considers the worst-case of all rounding modes and will always safely overestimate rounding errors so that fatal losses of precision leading to overflows are detected.

The *clock domain* has been specifically developed for synchronous control programs and supports relating variable values to the system clock [6].

With the *filter domain* [14] digital filters can be precisely approximated. In the following example the current output `P` is a function of the two previous outputs `S[0,1]`, the current input `X` and the two previous inputs `E[0,1]`. Astrée warns about the non-terminating loop but does not issue any alarm and thus can automatically prove the absence of run-time errors. The value range computed for `P` is $[-1418.3827, 1418.3827]$.

```
typedef enum {
    FALSE = 0,
    TRUE = 1
} BOOLEAN;

BOOLEAN INIT;
float P, X;

void filter ()
{
  static float E[2], S[2];
  if (INIT) {
    S[0] = X;
    P = X;
    E[0] = X;
  } else {
    P = (((((0.5*X)-(E[0]*0.7))+(E[1]*0.4))
         +(S[0]*1.5))-(S[1]*0.7));
  }
  E[1] = E[0];
  E[0] = X;
  S[1] = S[0];
  S[0] = P;
}

void main ()
{
  X = 5;
  INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35;
    filter ();
    INIT = FALSE;
  }
}
```

5. Parameterizing Astrée

The C99 standard does not fully specify data type sizes, endianness nor alignment. An integer can be represented either by a sign and an absolute value, by one's complement, or by two's complement. Additionally there are operating system dependencies, e.g., whether global or static variables are automatically initialized to zero, or not. Astrée is informed about these target settings by a dedicated configuration file and takes the specified properties into account.

Astrée can be adapted to specific program families in order to improve analysis precision. The key feature here is that Astrée is fully parametric with respect to the abstract domains: by selecting the set of active domains the analyzer can focus on the domains relevant to the software under analysis. Moreover Astrée can be extended by new abstract domains so that specific requirements of individual applications can be addressed[4].

In addition to the application domain awareness, there are two mechanisms for adapting Astrée to individual programs. First, abstract domains can be parameterized to tune the precision of the analysis for individual program constructs or program points [21]. Second, there are annotations for making external information available to Astrée. Both are presented in this section.

As current experience shows the parameterization of the programs under analysis rarely has to be changed when the analyzed software evolves over time. So in contrast e.g., to theorem provers the parameterization is very stable.

5.1 Parameterization of Abstract Domains

[4] Note that while, in general, the specialization of Astrée is under user control, incorporating new domains requires a new release of Astrée.

Let us illustrate the parameterization of abstract domains with two examples: *semantic loop unrolling* and *variable smashing*. They allow to tune the precision of the analyzer to the software under analysis, i.e., to analyze critical program parts with high precision, and improve speed by lowering the precision for uncritical program parts.

Semantic loop unrolling [21] enables the analyzer to distinguish different iterations of a loop to improve analysis precision. When a loop is unrolled n times, individual invariants are computed for the first n iterations and all subsequent iterations are summarized by a common invariant. In general, the analysis will become more precise with increasing unrolling and the analysis time will grow. Users can specify a default unrolling factor which can be overridden for individual loops by the `__ASTREE_unroll` directive. Astrée also offers a heuristic loop unrolling which automatically determines suitable unrolling factors. In the following example:

```
int main()
{
    int i=0;
    char c=0;
    char A[255];
    for (i=0; i<255; i++) {
      if (i<10)
        c+=10;
      else if (i<100)
        c++;
      A[i]=c;
    }
}
```

automatic unrolling enables Astrée to report no alarms and to precisely compute the values of `c` and of each cell of `A` at the program exit.

For large aggregate variables, it would be inefficient to represent each scalar component with a distinct object in Astrée. Variable smashing enables Astrée to use one single summary cell to represent the value of many cells at different memory locations. This results in a loss of precision but can improve memory consumption and analysis time. In general, Astrée supports partial variable folding, e.g., an array inside a structure can be folded without folding the rest of the structure. Arrays are automatically smashed when their size exceeds a certain global threshold that can be changed with the `smash-threshold` option. It is possible to locally override this setting for individual arrays by a dedicated directive. The directive `__ASTREE_smash_variable((V,n))` indicates that all arrays with `n` or more *elements* in the variable `V` should be folded. In the following example:

```
struct {
  int nb;
  int tab[10];
  struct { int x; int tab2[30]; } tab3[2];
} a;
__ASTREE_smash_variable((a,4));
```

the arrays `tab` and `tab2` will be folded, but not the array `tab3`.

5.2 Semantic Hypotheses

Astrée assumes that the value of `volatile` variables can change asynchronously at any program point. However, the `volatile` declaration sometimes is also used for non-volatile variables to prevent the compiler from performing certain optimizations. Therefore Astrée offers options for ignoring the volatile keyword for global, resp. local variables. With the directive `__ASTREE_volatile_input((V))` individual variables V can be declared as volatile even when the volatile keyword is ignored. It also supports taking into account some *hypotheses* on the possible values of the volatile variables; then the analysis assumes that their values can change asynchronously but will always stay within the specified bounds. The directive can target global, static and local variables and also supports structured variables. It is possible to mix volatile and non-volatile fields in the same structure. In the example:

```
typedef volatile int t;
struct {
  volatile int x;
  t y;
  int v;
  volatile int z[2];
  int *A;
} a;
```

the following parts of `a` are volatile: `a.x`, `a.y`, `a.z[0]`, and `a.z[1]`.

The directive `__ASTREE_assert((B))` tells Astrée to check whether the Boolean expression B is always true at this program point. If there is a context where B may evaluate to false, Astrée produces an alarm.

With the directive `__ASTREE_known_fact((B))` users can make additional knowledge available to Astrée, where `B` is a Boolean expression in C syntax without side effects. Astrée then assumes that at the program point of the directive the condition `B` is satisfied without checking this hypothesis. However, if Astrée can prove that `B` is always false, it issues a warning. A simple example is `__ASTREE_known_fact((i>0))`.

All specified hypotheses are summarized in the report file. This way, all conditions that have to be satisfied for the analysis result to be valid, are documented with the analysis result. When the execu-

tion conditions of the program change, it is enough to check whether these directives are still valid, and, if not, run a new analysis with updated hypotheses.

6. Qualification Support

Ideally, Astrée should be continually used during the software development process. This way, potential runtime errors are detected early which contributes to preventing late-stage design or integration problems. In the *validation stage* the goal is to verify that no runtime errors may occur. To be amenable for certification according to DO-178B, analysis tools have to be qualified [26]. The qualification process can be automated to a large degree by a *Qualification Support Kit*, which currently is under development. A qualification kit consists of a report package and a test package. The report package lists all functional requirements and contains a verification test plan describing one or more test cases to check each functional requirement. The test package contains an extensible set of test cases and a scripting system to automatically execute all test cases and evaluate the results. The generated reports can be submitted to the certification authority as part of the DO-178B certification package.

7. Practical Experience

Astrée has been used in several industrial avionics and space projects. One of the examined software projects from the avionics industry comprises 132,000 lines of C code including macros and contains approximately 10,000 global and static variables [2]. The first run of Astrée reported 1200 false alarms; after adapting Astrée the number of false alarms could be reduced to 11. The analysis duration was 1h 50 min on a PC with 2.4 GHz and 1GB RAM.

[7] gives a detailed overview of the analysis process for an Airbus avionics project. The software project consists of 200,000 lines of preprocessed C code, performs many floating-point computations and contains digital filters. The analysis duration for the entire program is approximately 6 hours on a 2.6 GHz PC with 16 GB RAM. At the beginning, the number of false alarms was 467 and could be reduced to zero in the end.

8. Conclusion and Outlook

Software errors in safety-critical embedded systems can cause severe damage. Development standards like DO-178B or ISO-26262 increasingly demand to demonstrate the absence of software errors. Software tools based on static program analysis offer a complete coverage and can contribute to significantly reducing testing effort. Here it is important to achieve a high analysis precision in order to keep the number of false alarms low. An analyzer should give detailed information about occurring alarms to help the user understand the reasons of the alarm. Furthermore the analyzer should be parameterizable so that users can tune the analyzer for the software and can eliminate false alarms.

Astrée has been specifically designed to meet these requirements: the analysis time scales well even for industrial applications with several 100KLOC. Even with default settings it produces only a small number of false alarms for control/command programs written in C. Since human alarm investigation is a time consuming task, this is essential for keeping the analysis effort at a reasonable level. Additionally Astrée supplies developers with all required information to understand the reasons of alarms and provides them with enough options and directives to help reduce this number significantly. Thus, in contrast to many other static analyzers Astrée cannot only be used to detect runtime errors, but to actually prove their absence. Industrial synchronous real-time software from the avionics industry could be successfully analyzed by Astrée with zero false alarms.

9. References

[1] AbsInt GmbH. Astrée Website. http://www.astree.de.

[2] 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 *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[3] JTC1/SC22. Programming languages – C, 16 Dec. 1999.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[6] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of Static Analyzers: A Comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007*, pages 3–20. IEEE Computer Society, 2007.

[7] D. Delmas and J. Souyris. ASTRÉ: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, 2007.

[8] A. Deutsch. Static Verification of Dynamic Properties. *ACM SIGAda 2003 Conference*, 2003.

[9] dPACE GmbH. Targetlink. http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm.

[10] École Normale Supérieure. ASTRÉE Website. http://www.astree.ens.fr.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *18th ACM Symp. on Operating Systems Principles*, 2001.

[12] Christian Ferdinand, Reinhold Heckmann, and Daniel Kästner. Static Memory and Timing Analysis of Embedded Systems Code. In *Proceedings of the IET Conference on Embedded Systems at Embedded Systems Show (ESS) 2006, Birmingham*, 2006.

[13] Christian Ferdinand, Reinhold Heckmann, and Reinhard Wilhelm. Analyzing the Worst-Case Execution Time by Abstract Interpretation of Executable Code. In Manfred Broy, Ingolf H. Krüger, and Michael Meisinger, editors, *Automotive Software - Connected Services in Mobile Networks. First Automotive Software Workshop (ASWSD 2004), San Diego, California, USA, January 10-12, 2004, Revised Selected Papers*, volume 4147 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2006.

[14] Jérôme Feret. Static Analysis of Digital Filters. In *European Symposium on Programming (ESOP'04)*, number 2986 in LNCS. Springer-Verlag, 2004. Â© Springer-Verlag.

[15] H. L. Garner. Theory of Computer Addition and Overflows. *IEEE Trans. Comput.*, 27(4):297–301, 1978.

[16] Klocwork. Klocwork K7^{TM}. http://www.klocwork.com.

[17] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *2001 USENIX Security Symposium, Washington, D.C.*, 2001.

[18] K. Leino and G. Nelson. An Extended Static Checker for Modula-3. In *7th Int. Conf. on Compiler Construction, CC'98*, number 1383 in LNCS. Springer, 1998.

[19] J.L. Lions et al. ARIANE 5, Flight 501 Failure. *Report by the Inquiry*, 1996.

[20] Matthieu Martel. An Overview of Semantics for the Validation of Numerical Programs. *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI05)*, 2005.

[21] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP'05*, 2005.

[22] A. Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *Proc. of the European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, Barcelona, Spain 2004. http://www.di.ens.fr/~mine/publi/article-mine-esop04.pdf.

[23] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf.

[24] MISRA (The Motor Industry Software Reliability Association). Guidelines for the use of the C language in vehicle based systems. http://www.misra.org.uk, 1998.

[25] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proc. 27th ACM SIGSOFT Int. Conf. on Software Engineering*, pages 580–586. ACM Press, 2005.

[26] Radio Technical Commission for Aeronautics. RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification.

[27] Jean Souyris and David Delmas. Experimental Assessment of Astrée on Safety-Critical Avionics Software. In *SAFECOMP*, pages 479–490, 2007.

[28] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.