# True Error or False Alarm? Refining Astrée's Abstract Interpretation Results by Embedded Tester's Automatic Model-Based Testing

Sayali Salvi[1], Daniel Kästner[1], Tom Bienmüller[2], and Christian Ferdinand[1]

[1] AbsInt GmbH, Science Park 1, 66123 Saarbrücken, Germany
[2] BTC Embedded Systems AG, Gerhard-Stalling-Str. 19, D-26135 Oldenburg, Germany

**Abstract.** A failure of safety-critical software may cause high costs or even endanger human beings. Contemporary safety standards require to identify potential functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Typically for ensuring functional program properties model-based testing is used while non-functional properties like occurrence of runtime errors are addressed by abstract interpretation-based static analysis. Hence the verification process is split into two distinct parts – currently without any synergy between them being exploited. In this article we present an approach to couple model-based testing with static analysis based on a tool coupling between Astrée and BTC Embedded*Tester*®. Astrée reports all potential runtime errors in C programs. This makes it possible to prove the absence of runtime errors, but typically users have to deal with false alarms, i.e. spurious notifications about potential runtime errors. Investigating alarms to find out whether they are true errors which have to be fixed, or whether they are false alarms can cause significant effort. The key idea of this work is to apply model-based testing to automatically find test cases for alarms reported by the static analyzer. When a test case reproducing the error has been found, it has been proven that it is a true error; when no error has been found with full test coverage, it has been proven to be a false alarm. This can significantly reduce the alarm analysis effort and reduces the level of expertise needed to perform the code-level software verification. We describe the underlying concept and report on experimental results and future work.

## 1 Introduction

Safety-related software has to satisfy stringent quality requirements. The complexity of software-implemented functionality grows at a fast pace. Development teams have to meet tight budget constraints and face increasing pressure to reduce time-to-market. To meet these conflicting goals the development process has to be sound and efficient.

A leap in development efficiency can be reached by a holistic model-centric approach to software development, testing and verification. In model-based development the software is graphically developed at a high abstraction level, typically by hierarchical finite state machines and data flow diagrams which represent specification and model at the same time. From this high-level model the implementation is automatically generated by configurable code generators, often in the form of C code. Model-based testing aims

A. Bondavalli et al. (Eds.): SAFECOMP 2014 Workshops, LNCS 8696, pp. 84–96, 2014.

at automating testing activities and integrating the development of both design artifacts and test artifacts in a unified framework. This makes it possible to automatically create test architectures, and generate and execute the test cases.

All contemporary safety standards require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage. This holds not only for DO-178B/DO-178C, but also for related norms like ISO 26262, and IEC-61508. Functional properties can be efficiently addressed by automatic model-based testing. The critical non-functional safety-relevant software characteristics are essentially implementation-level properties, e.g., whether real-time requirements can be met, whether stack overflows can occur, and whether there can be runtime errors like invalid pointer accesses, or divisions by zero. Because of the high abstraction level of model-based development these properties are largely hidden from the developers. Moreover they are very hard to check experimentally, i.e., by testing and measurements. Identifying safe end-of-test criteria for program properties like timing, stack size, and runtime errors is an unsolved problem. In consequence the required test effort is high, the tests require access to the physical hardware and the results are not complete.

Formal verification methods provide an alternative, in particular for safety-critical applications. One such method is abstract interpretation, which allows properties to be proven for all program runs with all inputs [8]. Nowadays, abstract interpretation-based static analyzers that can detect stack overflows [14] and violations of timing constraints [16], and that can prove the absence of runtime errors, are widely used in industry [15] (cf. Sec. 2). The advantage of abstract interpretation is that it enables full control and data coverage, but can be easily automatized and can reduce the testing effort.

In consequence, from a workflow perspective the verification process is split into two parts: model-based testing is used for showing functional program properties, and static analysis to prove the absence of non-functional program errors. In the course of the MBAT project[1] a concept for integrating model-based testing and static analysis has been developed which enables both aspects to be addressed seamlessly [13]. This concept has been realized by a tool coupling between BTC Embedded*Tester*® [7], and the static analysis tools aiT WCET Analyzer [1], StackAnalyzer [2], and Astrée [3] from AbsInt. Model-level information like execution model or environment specifications are automatically taken into account, reducing setup for test and analysis efforts and improving analysis precision. Tests and analyses can be launched seamlessly and produce unified result reports. While significantly improving the efficiency of the V&V process the tool coupling as described in [13] is mostly limited to workflow aspects and does not yet exploit the full potential of a combination of the two verification technologies, static analysis and model-based testing.

In this article we describe a deep technical integration between static runtime error analysis and model-based testing, implemented as a tool coupling between Astrée and BTC Embedded*Tester*®. Astrée is a sound static analyzer which can find all potential runtime errors in C programs. It works with an abstract semantics of the program which makes it possible to compute results even for big applications – the largest

[1] `http://www.mbat-artemis.eu/`

application investigated so far contains more than two million LOC. The downside of the abstraction mechanism is that there can be false alarms, i.e. spurious notifications about potential runtime errors, which are not actual bugs. Therefore, all alarms have to be investigated by the developers to determine whether they correspond to true errors which have to be fixed, or whether they are false alarms. This can cause significant effort. The key idea of this work is to apply model-based testing to automatically find test cases for alarms reported by Astrée. When BTC Embedded*Tester*® finds a test case reproducing the error, it has not only been proven that it is a true error, but users can directly investigate the situation in a debugger. When no test case reproducing the error could be found, the interpretation depends on the test model generation: when full test coverage can be achieved, the absence of the error has been proven. Situations where no full test coverage was possible, or where the error could not be reproduced in the given amount of time, have to be manually investigated – but even here the test coverage obtained is a valuable feedback for the user. With this coupling the effort for alarm analysis can be significantly reduced. Preliminary experimental results demonstrate the viability of our approach.

The article is structured as follows: we introduce the key concepts of static runtime error analysis in Sec. 2. Sec. 3 gives an overview of model-based testing with a focus on the concept of C observers which provide the basis for our methodological integration. The concept to automatically generate test cases for Astrée alarms is explained in Sec. 4. Experimental results are presented in Sec. 5 and Sec. 6 concludes.

## 2 Static Runtime Error Analysis

Over the last few years static analyzers based on abstract interpretation have evolved to be the state of the art for verifying non-functional software properties. A static analyzer is a software tool which computes information about the software under analysis without executing it. Abstract interpretation is a semantics-based method for program analysis which belongs to the formal verification methods. Its results are sound, i.e., they are valid for all program runs with all inputs and achieve full data and control coverage. The soundness of the analysis can be formally proven. Examples of abstract interpretation based static analyzers are tools to compute safe upper bounds on the worst-case execution time and the maximal stack usage of tasks [11] and to prove the absence of runtime errors [15]. Runtime errors like arithmetic overflows, array bound violations, divisions by zero, and invalid pointer accesses are critical programming errors. They can destroy the data integrity of a program, causing the program to behave erroneously, or to crash altogether. A well-known example for the possible effects of runtime errors is the explosion of the Ariane 5 rocket in 1996. The analyzer Astrée [3] is an abstract interpretation based static runtime error analyzer which finds all potential runtime errors in C programs, thereby enabling users to prove the absence of runtime errors [6]. Astrée analyzes structured C programs with the sole restrictions that no dynamic memory allocation and no recursion should be used, which is typically the case for safety-critical applications. The class of errors reported includes out-of-bound array accesses, erroneous pointer manipulations and dereferencing, integer and floating-point division by zero, integer and floating point overflows and invalid operations. Astrée

also detects read accesses to uninitialized variables, detects shared variables accessed by asynchronous threads and performs a sound value analysis for them, and enables users to prove user-defined static assertions. The static assertions can be applied to arbitrary C expressions so that functional program properties can be addressed. When Astrée does not report an assertion failure alarm, the correctness of the asserted expression has been formally proven. The core of Astrée is a sophisticated analysis engine which allows to fine-tune the analysis precision to the software under analysis. This makes very low false alarm rates possible: safety-critical avionics software of several 100,000 lines of C code could be analyzed successfully with Astrée without any false alarm [4,15].

Since Astrée is based on an abstract semantics of the program there can be false alarms, i.e. spurious notifications about potential runtime errors which are caused by the overapproximation inherent to the analysis. False alarms can also be caused by preconditions that have not been made known to Astrée. Therefore, all alarms have to be investigated by the developers to determine whether they correspond to true errors which have to be fixed, or whether they are false alarms. This can cause significant effort.

### 2.1 Runtime Errors and Alarms

Runtime errors can occur in situations where the behavior of the C program is undefined, or unspecified according to the C semantics [12]. A notification about a potential runtime error is termed as *Alarm*. Astrée distinguishes two main types of alarms:

**Type A:** alarm about a runtime error which has unpredictable results. The analyzer reports the alarm and continues the analysis for scenarios where the error does not occur. For contexts where the error definitely occurs, the analyzer reports a definite runtime error and stops the analysis as there is no feasible continuation. Examples are out-of-bound array accesses, or write accesses via dangling pointers.

**Type C:** alarm about a runtime error which has a predictable outcome. The analyzer continues the analysis by overapproximating all possible results, including the effect of the error. Examples are integer overflows, invalid shifts, invalid cast operations.

The alarm messages displayed in the Astrée GUI possess a well-defined format. It provides details such as execution context of the alarm scenario, alarm location, type of alarm and the actual alarm text message. In addition to this, users can also request for program invariants, i.e., access the value ranges of variables the analyzer has computed for each specific context. The alarm messages have the following syntax:

(*context* `at` *filename*`:`*line1*`.`*column1*`-`*line2*`.`*column2*)$^+$ `ALARM(`*class*`):` *alarm message*

The *context* information provides a forward sequence of unfinished function calls to reach the alarm location, loops encountered in these functions before reaching the alarm location and disambiguated conditional statements encountered before reaching the alarm location. The syntax is the following:

`call#`*f*`@`*line*
: function *f* is called at line *line*.

`loop@`*line*`=`*n*`/`*m* or `loop@`*line*`>=`*m+1*
: *n* is the rank of loop iteration and *m* is the unroll level. The first *n* iterations of the loop at line *line* are unrolled by the analyzer.

`if@`*line*`=true` or `if@`*line*`=false`
: if condition at line *line* is evaluated to true or false resp.

The location information provides start coordinates *line1*`.`*column1* and end coordinates *line2*`.`*column2* of the code fragment in the file *filename* for which the *alarm message* is issued. The *class* is one of the alarm types, i.e., A or C. An example of an alarm message is shown here:

```
[ call#analysis_wrapper@4 at astree.cfg:4.5-21
  loop@17=1/3 at astree.cfg:17.2-25.3
  call#fuelratecontroller@23 at astree.cfg:23.4-22
  call#IntakeAirflowEstimation@9921 at frc.c:9921.3-26
  call#Tab2DS17I2T4169_a@10221 at frc.c:10221.4-21
  loop@9554>=2 at frc.c:9554.3-9558.24
  ALARM (C): implicit signed int->unsigned char conversion range
  [-1, 254] not included in [0, 255] at frc.c:9555.6-16 ]
```

The alarm is reported for a potential overflow which occurs in the second or later iteration of the loop from line 9554 to line 9558 in file `frc.c`. The lines above describe the precise call stack traversed to reach the loop in the potential error scenario.

Astrée uses various abstract domains to compute program invariants providing detailed information about the values of variables and the relations between them in every possible execution context [6]. All computed invariants can be inspected by the users; depending on the option setting they can be observed either at each statement or in the beginning of each function, and they can be context-insensitive or context-sensitive. In the latter case the computed abstract values are shown separately for each execution context.

Note that Astrée computes an abstract semantics and provides local abstract invariants attached to program points. Thus, in case of a potential runtime error users can access the alarm message along with its context and the invariants. But it does not provide a concrete execution trace for the alarm, since each abstract trace represents a set of concrete execution traces.

## 3 Model-Based Testing

In this section we give an overview of model-based testing with the example of BTC Embedded*Tester*® (ET). ET provides an automated test and verification environment for Simulink/TargetLink and also for handwritten C code. It is capable of performing various tasks such as automatic test case generation and execution, back-to-back testing, automatic test analysis, automatic test and coverage reporting, debugging, and import/export of test cases.

### 3.1 Test Case Generation

ET can generate input stimuli vectors and test vectors. Input stimuli vectors represent a set of input values per execution step for a number of execution steps. They can be

either imported or generated using vector generation engines. ET uses two engines: the Automatic Test vector Generation engine (ATG) and the Code Verification analysis engine (CV). ATG is a random engine based on well-tuned heuristics which is fast, but not complete. If a test goal cannot be reached, it is unclear whether the goal is unreachable or the engine just could not reach it. In contrast, the CV engine is complete. It tries to generate vectors by claiming that a certain test goal is unreachable [5]. A counterexample, if there is one, is provided as a vector, otherwise the goal is proven unreachable because it uses an exhaustive technique based on symbolic model checking that checks the full search space. The input stimuli vectors can then be used to generate test vectors. Test vectors represent a set of input and output values per step. The output values in a test vector are the expected values. So, when test vectors are generated using simulation with input stimuli vectors, the output values in them need to be reviewed. ET also supports back-to-back testing in which case the output values are not required to be reviewed. Test vectors with expected values can also be imported in ET. ET supports various kinds of simulations like MIL (model-in-the-loop), SIL (software-in-the-loop) and PIL (processor-in-the-loop).

### 3.2 Observer

ET provides the possibility to define evaluation functions (C code) which can be used to decide if the behaviour of the system under test is correct or invalid. These evaluation functions are called *observers*. Observers are integrated automatically into the test execution/simulation environment of a system under test by ET. ET generates tests that cover observers and performs automatic test execution of the system under test in co-execution/simulation with observers. Observers are used in ET, e.g., for requirement verification, standard analysis, or additional test (e.g., equivalence class test cases) generation.

A C-observer (more precisely a *commitment observer*) is a small C function that evaluates some property of the system under test. It returns a boolean value indicating whether the observed property is valid or invalid. C-observer code can check conditions on interface objects of the system under test, i.e., inputs, outputs, calibration variables and display variables. An example of an observer is shown here:

```
/* Observer Evaluation Function */
unsigned char eval_OBS_2_RV_fuel_rate() {
  unsigned char evalResult = 1;
  if (!(fuel_rate >= 0 && fuel_rate <= 32358))
    evalResult = 0;
  return evalResult;
}
```

### 3.3 Property Location Language – PLL

ET uses the Property Location Language (PLL) for uniquely identifying code properties (i.e., test cases). For example, to generate input stimuli vectors for a code property the corresponding PLL expression of the code property has to be provided.

In general, a PLL expression has three parts, identifying code properties relevant for testing: 1) one or several property classes like Statement (STM), Relation Operator (RO), Decision (D), Division by 0 (DZ) etc. ; 2) a unique ID of a specific entity of this property class in the code; 3) a specific property value of this entity. One example is **D:3:1**, where "D" denotes the decisions in the code, "3" is the unique ID of a decision in the code (IDs are automatically generated by ET) and "1" represents the case that the decision evaluates to true. In case of observers, the PLL expression has two additional parts in the beginning i.e., in total five parts: it starts with "O:[OBSID]:...", where [OBSID] is a unique identifier of an observer. e.g., **O:OBS_1:D:3:1**.

### 3.4 Integration of C-Observer in ET

**Step1.** An observer is defined by a user or it is generated automatically from some source, e.g., requirement specification.

**Step2.** Stimuli vectors are generated for the observer. Using the right PLL formulation to indicate the desired observer test properties is important.

**Step3.** The generated stimuli vectors are used to generate the corresponding test vectors.

**Step4.** The ET debugging feature can be used to perform step-by-step debugging and analysis of the behaviour of the generated vector. Alternatively, in some use cases like requirement verification, the test vectors are automatically executed. During test execution, the observer is run in parallel with the system under test and the return value of the observer is used to check whether the system execution is valid. The execution is valid when the return value is 1 and invalid when it is 0.

## 4 Combining the Two Worlds

Our goal is to combine the static analyzer Astrée with BTC Embedded*Tester*® in order to validate whether an alarm is a true error or a false alarm. The key idea is to automatically transform an alarm reported by Astrée (described in Sec. 2.1) into a C-observer that can be integrated in ET (described in Sec. 3.2). This transformation includes inserting some code fragments into the source code under test and generating the observer code that exploits the information provided by the inserted code. The code modifications are free of side effects and do not interfere with normal program execution. The integration includes creating an ET profile for the modified version of system under test and importing the observer into the profile. ET supports generating test vectors from the observers. This approach can be applied to any category of errors detected by Astrée. In the following we demonstrate our approach with four exemplary alarm categories.

As already stated in Sec. 2.1, alarm messages contain the precise location of the code fragment that is responsible for the alarm. The code insertion phase uses the alarm message and its location information to store the error condition into variables at the appropriate location. These variables include one flag variable which indicates if the alarm condition is reachable. Note that the flag variable needs to be global so that it becomes a part of an interface of the system under test: only then it becomes observable by ET. The code insertion is explained in detail for each of the four selected alarm categories in the following subsections. The observer code just consists of checking the

flag value. If the flag value is true, an error condition has been reached. The observers are designed to return 0 when the flag value is true. Our generic observer code is shown in Fig.1.

```
unsigned char eval_OBS_1() {
  unsigned char evalResult = 1;
  if (__ET_flag_1 != 0)
    evalResult = 0;
  return evalResult;
}
```

**Fig. 1.** Generic Observer Code

### 4.1 Division by Zero

The alarm message for divisions by zero is `<type> division by zero [interval]`, where "`type`" can be either integer or float. A simple example of an integer division by zero alarm is shown below.

```
[ call#analysis_wrapper@3 at astree.cfg:3.5-21
  loop@16=1/3 at astree.cfg:16.2-24.3
  call#fuelratecontroller@22 at astree.cfg:22.4-22
  call#IntakeAirflowEstimation@9931 at fuelratecontroller.c:9931.3-26
  call#Tab2DS17I2T4169_a@10231 at fuelratecontroller.c:10231.4-21
  ALARM (A): integer division by zero [0, 65535] at fuelratecontroller.c:9596.27-9597.66 ]
```

The location details `fuelratecontroller.c:9596.27-9597.66` in the alarm message provide the code fragment for which the alarm is issued: it extends from column 27 in line 9596 to column 66 in line 9597. The corresponding code fragment, a division expression, is highlighted after clicking on the alarm message in the GUI:

```
9594        /* 1. Y-Interpolation. */
9595    __ET_flag_1 = (Aux_U16_b == 0);
9596      Aux_U16_c = (UInt16) ((((UInt32) (((UInt32) Aux_U16_a) * ((UInt32) z_table[0]))) + ((UInt32)
9597        (((UInt32) Aux_U16) * ((UInt32) z_table[1])))) / Aux_U16_b);
9598      z_table += Aux_U8;
```

We parse the division expression to retrieve its denominator and then generate code which stores the condition under which the denominator becomes zero into a flag variable. We assume that execution of the denominator expression has no side effects. The generated assignment is put at the program point just before the division expression (cf. line 9595 in the above image). The flag variable `__ET_flag_1` is then used in an observer as shown in Fig.1.

### 4.2 Overflow in Arithmetic

The alarm message for arithmetic overflows of numerical types reads `<type> arithmetic range [interval] not included in [interval].`

The two intervals in the message show the actual value range computed for the expression and the acceptable value range, respectively. Here is an example of an alarm message on an overflow of an `unsigned int`.

```
[ call#analysis_wrapper@3 at astree.cfg:3.5-21
 loop@5=1/3 at astree.cfg:5.2-13.3
 call#fuelratecontroller@11 at astree.cfg:11.4-22
 call#IntakeAirflowEstimation@1888 at fuelratecontroller.c:1888.3-26
 call#Tab2DS17I2T4169_a@2165 at fuelratecontroller.c:2165.4-21
 ALARM (C): unsigned long arithmetic range [0, 6516407190] not included in [0, 4294967295] at fuelratecontroller.c:1600.28-1601.53 ]
```

The message gives the location `fuelratecontroller.c:1600.28-1601.53` of the code fragment i.e., the arithmetic expression for which the alarm is issued. The fragment is highlighted at lines 1600 and 1601 in the image below.

```
1595    __ET_var_1 = ((UInt32) (((UInt32) Aux_U16_a) * ((UInt32) z_table[0])));
1596    __ET_var_2 = ((UInt32) (((UInt32) Aux_U16) * ((UInt32) z_table[1])));
1597
1598      __ET_flag_1 = (4294967295 - __ET_var_2 < __ET_var_1);
1599
1600      Aux_U16_c = (UInt16) ((((UInt32) (((UInt32) Aux_U16_a) * ((UInt32) z_table[0]))) + ((UInt32)
1601       (((UInt32) Aux_U16) * ((UInt32) z_table[1])))) / Aux_U16_b);
```

The arithmetic expression is parsed in order to retrieve its operator and the corresponding operands. In this example, it is '+' operator. Now we have to check if the operands can evaluate to such values that when are added to each other they cause an overflow. The corresponding code is shown in lines 1595-1598 in the above image. Note that the operands, which may be complex expressions, are stored into variables of type "`type`" and it is ensured that the inserted code does not raise the overflow alarm itself. The code generated for this category of alarm can vary based on the arithmetic operator (i.e., whether it is + or - etc. ), whether it is signed or unsigned arithmetic and whether the overflow is w.r.t. minimum and/or maximum bound.

### 4.3 Overflow in Conversion

The third category of alarms is associated with overflows in conversions, i.e., type casts. The alarm message reads `<type1>-><type2> conversion range [interval] not included in [interval]`. An expression responsible for this alarm evaluates to a value of type "`type1`". An overflow may occur when a type of an expression value is converted implicitly or explicitly to type "`type2`". This alarm message shows the range of potential values of the expression before and after conversion in the first and second interval, respectively. We determine the region of the first interval that does not overlap with the second interval. The required instrumentation then consists of storing the expression value into a variable of type "`type1`" and checking whether a value of this variable is in the non-overlapping region of the first interval. The check is put at a location just before the expression that is responsible for an alarm.

### 4.4 Invalid Dereference

The fourth category of alarms considered is associated with invalid pointer dereferences: `Invalid dereference: dereferencing <value> byte(s)`

`at offset(s) <value> may overflow the variable <name> of byte-size <value>`. This alarm is raised when dereferencing a pointer expression that points to an invalid location, e.g., because it has not been properly initialized. The message indicates the offset, expressed in bytes, and the byte size of the dereferenced variable "`name`". An example of alarm message is shown below:

```
[ call#analysis_wrapper@4 at astree.cfg:4.5-21
 loop@17=2/3 at astree.cfg:17.2-27.3
 call#f0@24 at astree.cfg:24.4-6
 call#f1@1884 at controller.c:1884.3-5
 call#f2@2187 at controller.c:2187.16-18
ALARM (A): invalid dereference: dereferencing 2 byte(s) at offset(s) 34+2*[0;305] may overflow the variable global_array of byte-size 612 at controller.c:1480.13-21 ]
```

and the corresponding code fragment `array[n]` is highlighted at line 1480 in the image below:

```
1471    int __ET_var_1 = 0;
1472    if (array == (const UInt16 *)global_array)
1473       __ET_var_1 = 1;
1474
1475    array += ((UInt16) (((UInt16) (((UInt16) v1) * ((UInt16) n))) + ((UInt16) v2)));
1476    val1 = array[0];
1477    if (v3 == 0) {
1478       if (__ET_var_1 == 1)
1479          __ET_flag_1 = (((array + n) - (const UInt16 *)global_array) >= 306);
1480       val2 = array[n];
1481    }
```

In the specific case of the pointer dereference in the example message above, `array+n` accesses offsets from 34 to $34 + 610$ bytes, which exceeds the valid offset range of $[0, 612 - 2]$ bytes. To capture the error condition here we have to check whether dereferencing `array+n` would leave the feasible range as shown at lines 1471-1473 and 1478-1479 in the image above. The comparison `(array == (const UInt16*) global_array)` is used to identify the correct alarm context. In general the function `f2` containing the code with the alarm is invoked from different call sites with different parameters. The specific alarm under analysis is for one specific context where `f2` is called with `global_array` as a parameter. Generating the instrumentation code for pointer dereferences currently is ongoing work. We plan to use program slicing to explicitly construct the possibly invalid pointer value and check it for feasibility. Astrée already provides a program slicer which can be reused for this purpose.

## 5 Experiments and Practical Experience

We performed experiments to investigate the applicability of our approach with a couple of control applications: namely, a fuel rate controller from the automotive domain and a simple flight control system from the avionics domain. The fuel rate controller is a fixed-point implementation generated by dSPACE TargetLink [9] from a MATLAB/Simulink model and consists of 2837 lines of code. Also the avionics example is a model-based design; here Esterel SCADE [10] has been used as a code generator. The implementation consists of 2205 lines of floating-point C code.

Executing an Astrée analysis on each of the two applications provides us with a list of alarms. The obtained alarms did not include division by zero alarms. Thus, for our

purpose we modified the fuel rate controller code to induce an integer division by zero alarm and the flight controller code to induce a float division by zero alarm.

In our implementation we have extended Astrée to automatically generate observers and insert the associated instrumentation code snippets into the source code for each alarm selected by the user. The implementation is ongoing work: so far we have successfully accomplished the automatic handling of division by zero alarms; the support of overflow alarms currently is restricted to a limited set of C-operators. For the examples where the implementation is not available yet, we have manually written the observers and the instrumentation code snippets in the same form as the implementation shall provide.

The generated observers have been verified with ET. This includes generating the ET profiles from the instrumented source code, importing the observers into those profiles, and generating the stimuli vectors that cover the observer properties. As explained in Sec. 3.3, a PLL string is used to represent the observer property. In our case, we are interested in generating a stimuli vector for the trace that demonstrates the actual alarm condition. This is the case when the flag variable is true, i.e., when the corresponding observer returns 0. So, the PLL string that is used during verification is "O:[OBSID]:V:0", where "V:0" stands for the return value zero. It is possible that no vector is generated during stimuli vector generation. This indicates that the specified code properties are not reachable during any execution. In our case, it means that the alarm under investigation is a false alarm. Table 1 shows the result of stimuli vector generation for our observers performed at the highest level of the subsystem hierarchy of the software under analysis.

In order to ascertain the correctness of the obtained results, we debug the generated stimuli vectors when the status is *covered*, whereas we do a manual inspection of the code when the status is *unreachable*. ET provides a feature to produce a debugging

**Table 1.** Stimuli Vector Generation Results: *covered* - stimuli vector is generated (true error); *unreachable(inf)* - stimuli vector is not generated (false alarm)

| Alarm Category | Controller | Alarm Message | Status |
|---|---|---|---|
| Integer division by zero | fuel rate | ALARM (A): integer division by zero [0, 65535] at fuelratecontroller.c:9596.27-9597.66 | unreachable(inf) |
| Float division by zero | flight | ALARM (A): float division by zero [0., 10000.] at ComputePitchRoll_FlightControl.c:512.28-54 | covered |
| Arithmetic overflow | fuel rate | ALARM (C): unsigned long arithmetic range [0, 6516407190] not included in [0, 4294967295] at fuelratecontroller.c:1600.28-1601.53 | unreachable(inf) |
| Conversion overflow | fuel rate | ALARM (C): implicit signed int->unsigned char conversion range [-1, 254] not included in [0, 255] at fuelratecontroller.c:9568.6-16 | unreachable(inf) |
| Invalid dereference | fuel rate | ALARM (A): invalid dereference: dereferencing 2 byte(s) at offset(s) 34+2*[0;305] may overflow the variable PressEst_z_table of byte-size 612 at fuelratecontroller.c:9449.18-28 | covered |

environment directly from the vector. Through debugging it is possible to analyze the execution trace covered by the vector. In our case, we check if the trace reaches the alarm condition.

The results show that all alarm categories investigated can be successfully handled for both input applications.

## 6 Future Work and Conclusion

In this article, we have presented an approach to automatically classify the alarms produced by a static analyzer as true errors or false alarms by applying model-based testing techniques to stimulate appropriate error conditions. We have described the principles of this interaction between static analysis and model-based testing and have developed an implementation based on a tool coupling between the static runtime error analyzer Astrée and the model-based testing tool BTC Embedded*Tester*®. Our approach significantly reduces the effort for alarm analysis, i.e., investigating alarms to find out whether they are true errors which have to be fixed, or whether they are false alarms. As the tool coupling can run fully automatically it also opens the alarm investigation process to users with less experience than manual investigation requires. Preliminary experiments demonstrate the viability of our approach with fixed-point and floating-point applications from the automotive and aerospace domains. To the best of our knowledge this is the first successful combination of static analysis and model-based testing to exploit synergies between these techniques in the V&V process of safety-critical software.

Our future work, in a first step, aims at completing the implementation to handle all four exemplary alarm categories automatically. In a further step, the mechanism has to be extended to cover the full set of Astrée alarms. Also further experiments on industry-relevant applications will be conducted to check how the proposed method scales with large-scale software projects.

**Acknowledgement.** The work presented in this paper has been supported by the ITEA2 project TIMMO-2-USE and the EU ARTEMIS Joint Undertaking under grant agreement no. 269335 with the German BMBF (MBAT project).

## References

1. AbsInt GmbH. aiT Worst-Case Execution Time Analyzer Website, `http://www.AbsInt.com/ait`
2. AbsInt GmbH. StackAnalyzer Website, `http://www.AbsInt.com/sa`.
3. AbsInt GmbH. Astrée Website, `http://www.AbsInt.com/astree`.
4. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace 2010, number AIAA-2010-3385, pp. 1–38. American Institue of Aeronautics and Astronautics (April 2010)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

6. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), San Diego, California, USA, June 7-14, pp. 196–207. ACM Press (2003)
7. BTC Embedded Systems AG. BTC BTC Embedded*Tester*® Website, `http://www.btc-es.de/index.php?idcatside=2`.
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
9. dSPACE GmbH. TargetLink Website, `http://www.dSPACE.com/go/TargetLink`
10. Esterel Technologies. SCADE Suite, `http://www.esterel-technologies.com/products/scade-suite`
11. Ferdinand, C., Heckmann, R.: Static Memory and Execution Time Analysis of Embedded Code. SAE 2006 Transactions Journal of Passenger Cars - Electronic and Electrical Systems 9 (2007)
12. ISO/IEC 9899:1999 (E). Programming languages – C (1999)
13. Kästner, D., Brockmeyer, U., Pister, M., Nenova, S., Bienmüller, T., Dereani, A., Ferdinand, C.: Combining Model-based Analysis and Testing. In: Embedded Real Time Software and Systems Congress ERTS$^2$ (2014)
14. Kästner, D., Ferdinand, C.: Proving the Absence of Stack Overflows. In: SAFECOMP 2014: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security (to appear, 2014)
15. Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Astrée: Proving the Absence of Runtime Errors. In: Embedded Real Time Software and Systems Congress ERTS$^2$ (2010)
16. Souyris, J., Pavec, E.L., Himbert, G., Jégu, V., Borios, G., Heckmann, R.: Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In: Proceedings of the 5th International Workshop on Worst-case Execution Time (WCET 2005), Mallorca, Spain, pp. 21–24 (2005)