# Automated Testcase Generation for Numerical Support Functions in Embedded Systems

Johann Schumann
SGT, Inc., NASA Ames
NASA Ames Research Center
Moffett Field
California 94035
Johann.M.Schumann@nasa.gov

Stefan-Alexander Schneider
BMW AG
80788 München
stefan-alexander.schneider@bmw.de

## ABSTRACT

Modern cars contain large amounts of safety-relevant software, which requires proof of achievement of certain safety integrity levels. A major task is to demonstrate that the developed software fulfills its specification and does not contain undesired functionality. Due to their size and complexity testing of this software has become an extreme bottle neck.

To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software level shall be determined and the structural code coverage shall be measured, e.g. statement coverage, branch coverage or Modified Condition-/Decision Coverage, short MC/DC. In this context the model-based software development raises additional questions that are currently discussed. A central question here is for example what is meant by a meaningful model coverage with respect to embedded code?

In this paper, we discuss these items on a specific type of code: Small support routines (e.g., for trigonometric functions, table look-up, or interpolation) are usually left out of these considerations and assumed to be fully correct. However, such functions can contain far from trivial code that can produce bad surprises, in particular if the code is executed on a platform different from the modeling platform. We present a method for the automatic generation of test stimuli for support functions, which uses the KLEE tool and some promising results.

## Categories and Subject Descriptors

D.2.5 [**Software**]: Testing and Debugging; D.2.8 [**Software**]: Metrics—*code coverage metrics*

## 1. INTRODUCTION

Modern cars heavily depend on software for safe and reliable operation. Not only the engine and drive train is controlled via software, but many components like brakes, park assistants, power windows, communications, and entertainment are implemented in software and run on one or multiple processors. It is estimated that a modern mid-size car is running more than 100 millions lines of code [2]. With the increase of software size and complexity, model-based approaches have found their way into safety-relevant automotive applications. In a typical model-based development environment (Figure 1, middle column), the specification model is first designed using a high-level, domain-oriented modeling tool. Typically,[1] such tools, like Mathworks' Simulink/Stateflow [21] facilitate quick design, modeling, and analysis on a graphical level. Many safety and performance analyses can be directly performed on that model level, e.g., model reviews to ensure the intended functionality.

Such a specification model represents the so-called high-level application that is then automatically translated by translation, implementation, and generation steps into embedded C-code. For example, Mathworks' RealTime Workshop is used to generate code from Simulink and Stateflow models. Other available commercial modeling environments provide tools of similar functionality e.g. TargetLink form dSPACE [6] or ASCET from ETAS [8]. Such C-code generators play a central role in model-based development approaches. These generators work (in the sense of computer science) like compilers, e.g., [10], from a (semi-)graphical representation into a C-code representation of the translated function. The generated code is finally embedded into the software load running on a (usually small) processor or microcontroller. To give an idea what a typical high-level application in a car needs for representation: either 150 (semi-)graphical diagrams or 80,000 lines of generated C-Code.

For many applications, the automotive industry is using OSEK [25] as the underlying operating system. The OS kernel, (hand-coded) drivers, and the autogenerated application code are linked to form the executable. As shown in Figure 1, compilation and linking also has to be performed on hand-coded legacy code and the basic software. That software contains a number of support routines, which are being used by the application. Such routines typically include advanced floating point operations (like trigonometric

[1] A worthwhile alternative modeling language is Modelica, which is a non-proprietary, object-oriented, equation based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents [23].

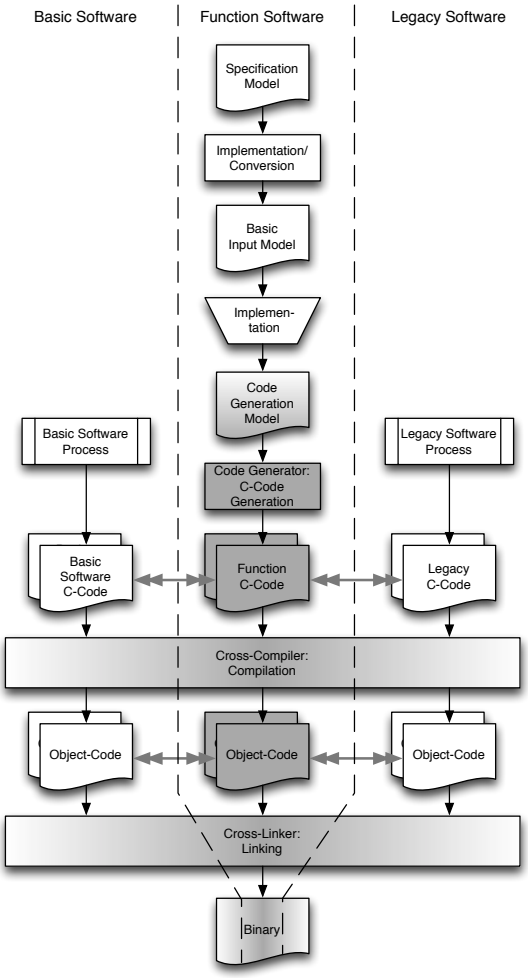


**Figure 1: The generic build process for a model-based development toolchain with an automatic code generator (from [19])**

functions, exponentials, logarithms, square roots) as well as support functions for the auto-generated code (e.g., table look-up, interpolation, or integrators). Many (embedded) compilers use the Netlib [24] math library, or parts thereof like FDLIBM [9]. Also, John Hauser's SoftFloat [12] is being widely used. Most underlying algorithms, approximations, and tables are based on [11].

For any safety-relevant software, the necessity to evaluate the completeness of test cases and to demonstrate that there is no unintended functionality also holds for such small support functions. In this paper, we use the advanced symbolic virtual machine KLEE [1] to automatically generate test cases (i.e., stimuli) to provide the required code coverage during testing. As the decision procedures available with KLEE are very weak in handling floating-point arithmetic, we use code abstraction and an iterative test case generation mechanism, which will be discussed in detail and which will be illustrated with two numerical support functions.

The reminder of the paper is structured as follows: in Section 2, we give an overview over testing according to code coverage metrics and discuss some applicable standards. Section 3 focuses on specific issues on testing of automotive software. In Section 4, we will present major related work on automated test case generation. We will demonstrate our approach for test case generation with a numerical support function calculating the sin of a value (Section 5). In Section 6, we discuss, how KLEE can be used for our purposes and describe the abstractions necessary. We present experimental results for our example. Section 7 discusses a second example, a table lookup (interpolation) function, which is found in most model-based development environments. Section 8 finally concludes and discusses future work.

## 2. TESTING FOR CODE COVERAGE

Developing such huge amounts of code and "getting it right" has become a huge challenge for car manufacturers and subsystem vendors, and a large amount of money is spent on *verification and validation* (V&V). For example, a single code review of the 80,000 lines of C-code mentioned above costs at least 300,000 US Dollars. This figure is only reasonable if everything works as expected; otherwise the costs will be much higher. And there are some dozens of such functions in a car.

Over the years, many safety and development standards have been developed to address this functional safety aspects of the software development. One of the recently published, prominent standards is *ISO 26262 road vehicles* [14]. Here, and in similar standards like the new version DO 178-C [30] the interpretation of the validation standards play an important role for the automated generation of test cases (of automated generated code).

Code coverage is a common metric to evaluate the completeness of test cases and to demonstrate that there is no dead code, deactivated code, or unintended functionality. Code coverage is therefore an important test end criteria. Such metrics were originally introduced in [22] and popular metrics include (e.g., [32])

**statement coverage:** each statement of the code must covered by at least one test case,

**decision coverage (also branch coverage):** each conditional statement (e.g., if, switch, for, while) must be executed in its true and false condition,

**condition coverage:** each Boolean subexpression must be evaluated to true and false,

**Modified Condition/Decision Coverage (MC/DC):** each entry/exit point must be tested; all possible outcomes in a decision have to be tested. Furthermore, each condition has to be shown to affect the decision outcome independently, and

**path coverage:** every possible route through the code must be exercised by at least one test case.

In most cases, obtaining path coverage for a larger piece of software is not feasible. Therefore, MC/DC had been introduced and is required by several safety standards (e.g., DO-178B [29] for level A safety-critical code in civil aviation). This metric is stronger than decision coverage, but does not nearly need as many test cases as path coverage. With higher requirements to the coverage, the number of test cases increases tremendously, motivating the use of automatic test case generators.

# 3. AUTOMOTIVE COVERAGE REQUIREMENTS

The standard *26262 Road vehicles – Functional safety* [14] is an adaptation of IEC 61508 [4] in order to comply with requirements specific to the application sector of electrical and/or electronic (E/E) systems within road vehicles. *Part 6: Product Development at the Software Level*, Chapter *9 Software Unit Testing* is describing the sub-phase software unit testing. Here, the major objective is to demonstrate that software units fulfill the software unit design specifications.

The required structural coverage metric at the software unit level depends on the so-called Automotive Safety Integrity Level (ASIL) and is shown in Table 1.

| | **Methods** | **ASIL** | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Statement Coverage | ++ | ++ | + | + |
| 1b | Branch Coverage | + | ++ | ++ | ++ |
| 1c | MC/DC Coverage | + | + | + | ++ |

**Table 1: Different coverage metrics according to [14] 6-9, Table12. The symbol ++ indicates highly recommended and + recommended for the identified ASIL.**

The ASIL provides an automotive-specific risk-based approach to determine integrity levels four levels to specify the item's or element's necessary requirements of ISO 26262 and safety measures to apply for avoiding an unreasonable residual risk, with D representing the most stringent and A the least stringent level.

Unfortunately, there is no example given how coverage should be measured in a given project. Especially for model-based software development, there is an on-going discussion on the interpretation of these coverage measurements. A central question here is whether the autocoder provides a clean 1-to-1 mapping of model elements to code or whether additional conditional statements or other code fragments are generated.[2]

[2]For example, *Note 4, 9.4.6, ISO 26262-6* proposes to perform back-to-back comparison tests to ensure that the behavior of the model with regard to the test objectives is equivalent to the automatically generated code. This is a clear indication that, at least from the point of view of the authors, a detailed knowledge is necessary on how the code is generated. Once all possible execution paths and their statements are known, e.g., from the specification, it is easy to derive stimuli for the models.

# 4. RELATED WORK

Because of the necessity to have large sets of test cases in order to test embedded software according to applicable standards, substantial research has been put into the topic of automatic testcase generation. The underlying idea of such tools is to generate, based upon information given by requirements, specifications, or the actual code, a set of test cases (usually pairs of stimuli and desired result) such that required testing metrics are met. In the realm of model-based development, several commercial tools are available.

Mathworks' Design Verifier [20] can analyze a Simulink-/Stateflow model and generate test cases that cover the entire signal and control flow of the model. These test cases can be used to exercise code that has been generated by the Realtime Workshop code generator. T-VEC [31] and the tool sl2tvec provide a tool chain that again takes Simulink-/Stateflow models and produces test cases, using a formal-methods (theorem proving) based approach. The T-VEC tool itself can generate test cases from table-based specifications.

The Reactis tool suite [28] translates Simulink/Stateflow models into RSML-e before other tools analyze the model or generate test cases. Finally, symbolic PathFinder [27] is a tool for the automatic generation of test cases for Java programs. It is an extension of the Java Pathfinder (JPF) model checker [15]. A tool-chain developed by Vanderbilt University translates relevant subsets of Simulink and Stateflow into Java code. The produced code then is used as an input to the symbolic pathfinder.

All tools discussed above (except symbolic pathfinder) use the model as their input specification and generate test cases that fully cover the model (given the model semantics). Thus, there is an ongoing discussion on the suitability of autogenerated test cases, which are based on the model semantics. If, for example, the code generator does not produce code, which is fully compatible to the model semantics, or advanced code optimizations take place, a full *code* coverage cannot be obtained. On the other hand, automatic test case generators, which directly work off the generated code might not be able to produce a fully covering set and does not feature traceability between test cases and the model.

# 5. RUNNING EXAMPLE

For the discussion of our method on the automatic test case generation for numerical support function, we use a standard implementation of the trigonometric function *sin* as our running example. Listing 3 shows a slightly abstracted version of a typical implementation; for details see [3, 11]. This function takes as argument a double value $x$ and returns a numerical approximation of $y = \sin x$ as a double value. In a first step, the argument of type double is pulled apart into its components: the exponent $exp$, the mantissa $m1, m2$, and the sign. This is a typical operation according to the IEEE floating point standard 754 [7] (more details can be found in, for example, [26] or [13]). Listing 1 shows the C data structure for 64 bit floating-point number and its components (according to IEEE [7]). This data structure is defined in such a way that construction and deconstruction of the floating point number is possible. Please note that this format usually is machine-dependent. Typical code con-

tains a large amount of C-macros to configure the code for the target hardware. For simplicity reasons, we assume a fixed target architecture (intel) but want to point out that such configuration mechanisms open up another space to be covered by testing.

**Listing 1: Memory format for double floating point numbers and data structure for bitwise access.**

```
union ieee_bits {
  double d;  // double
  struct {   // mantissa+exponent
    unsigned int m2:32;    // mantissa2
    unsigned int m1:20;    // mantissa1
    unsigned int exp:11;   // exponent
    unsigned int sign:1;
    } b;
  struct {
    unsigned int i1;       // 32 bit package
    unsigned int i2;
        } i; };
```

In Lines 6–12 of the sin algorithm (Listing 4), special cases like not-a-number, short NaN, or infinity is handled. If $x$ is very small, the result is approximated by $x - x^2$ or by Taylor series approximation ($x - x^3/3!$). Then the input argument is the normalized into a range between 0 and $\pi/2$. Because this has to be accomplished without loss of accuracy, a relatively complex algorithm is used (Lines 22–46). Finally the value is mapped to a range between 0 and 1 and a polynomial approximation of order 7 (Line 50) yields the final result. The evaluation of the polynomial POLYNOM(x2) is done using a Horner schema and implemented as a set of C macros (not shown).

Although there are no loops in this code, there is a substantially complex control flow with nested if-then-elses and case statements, which makes a manual development of test cases that fully cover all paths of the code very hard. In general, most of the support functions are very small (up to a few hundred lines of C-code), but the algorithms that they encode are far from trivial. Typical language at language constructs that are found include[3]:

**Macro Expansion:** Often configuration details and variants are implemented using `#define` macros and conditional compilation. Depending on the application, exactly the target configuration is used ("test what you run"), or testcases for all potential configurations can be generated. In this paper, we always executed the testcase generation with a given configuration, but being able to test all possible configurations can add a layer of confidence.

**Control Flow:** The full set of control flow constructs (conditional expressions, if-then-else, case statements, and loops) are used. Loops with fixed upper bounds as well as endless loops `for(;;){...}` with exit conditions are found,

**Data Types and Operations:** Besides the common data types for floating point numbers and integers, struct, union, and bit vectors are common. Arrays and data structures are of fixed size. Conversion of data types by casting or the use of union to construct and deconstruct floating-point numbers can be found frequently.

[3]Here, we just analyze C code functions.

# 6. TEST CASE GENERATION WITH KLEE

In our approach, we use the tool KLEE to automatically generate test cases. KLEE [16, 1] is a symbolic virtual machine, which runs on the LLVM (Low-Level Virtual Machine) compiler infrastructure [18]. For the explanation of the test case generation, let us consider this small program snippet

```
if ( x < 0 || x > 100)
    stmt_A;
else
    stmt_B;
```

This small program has three paths; two reach statement A (via a true in the first part of the condition, and via a true in the second part), and the third path reaches statement B. In a traditional, concrete execution the variable x has to have a numeric value, and thus execution only hits one of the three paths. In symbolic execution, the variable x is declared to be symbolic. Then, all paths are executed using a backtracking search and assembling so-called path constraints, i.e., constraints about the symbolic variables that need to be satisfied to reach that path. Table 2 shows our path conditions.

| Path# | Path Constraint |
|---|---|
| 1 | $[X < 0] : A$ |
| 2 | $[X > 100] : A$ |
| 3 | $[\neg X < 0 \wedge \neg X > 100] : B$ |

**Table 2: Path numbers and path constraints with symbolic variable $X$**

Once all paths have been searched, the set of path constraints are tried to be solved for the symbolic variables. Powerful constraint solvers are used for this purpose. If a solution is not unique (as in our case), a random value in the applicable range is chosen. Such a solution, in our case, for example, $\langle -1, 121, 55 \rangle$ then comprises a set of test cases, which fully cover all paths of the program.

KLEE performs it search for test cases using many refinements and optimizations of this basic algorithm. Constraint solvers for integer arithmetic, as well as some other popular data types are built into this tool. Since, however, there is no solver for full floating-point arithmetic, KLEE only supports very few operations, although it can (syntactically) handle float and double data types. In most cases, KLEE silently instantiates the variable with a random value, which is not of much use.

## 6.1 Abstraction for test case generation

We therefore apply abstraction to our program under consideration. Figure 2 shows the details of this process. Starting with the original source code P (here the code in file `sin.c`), we apply an abstraction $\mathcal{A}$ to it. Then the abstracted program is given to KLEE, which in turn automatically produces test cases $\mathcal{T}$ that, when executed provide a full path

coverage on the abstracted program. We, however, will apply $\mathcal{T}$ to the original program.

If the abstraction $\mathcal{A}$ was strongly preserving (with respect to the desired code coverage metric), $\mathcal{T}$ would provide our desired set of test cases. However, due to the undecidability of real numbers, no such abstraction exists in general. We therefore use an iterative process: starting with an abstraction $\mathcal{A}_1$, we generate test cases $\mathcal{T}_1$, which fully cover the abstracted program. We then execute the test cases on the original code and measure the coverage. If we obtain 100% coverage, we take this set. Otherwise, we start KLEE again to obtain a new set of test cases,[4] or refine the abstraction. For our examples, obtaining a suitable abstraction was straight-forward, but there is no guarantee.

By using abstraction, we also cannot use the obtained abstracted function result as an oracle. Rather, we have to use an external reference implementation of our support function to compare the results of the tests.

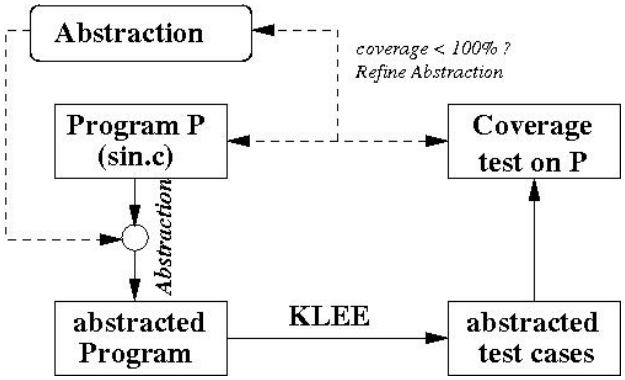


**Figure 2: The use of abstraction for the automated testcase generation. Starting with an initial abstraction, test cases are generated by KLEE for the abstracted program. These are executed on the original program P and code coverage is measured. If the desired coverage criteria are not met, the abstraction is refined and the process starts over again.**

Specifically, we use the following abstraction elements:

- All `double` definitions were replaced by `int`.
- All floating-point constants in the code (e.g., for $\pi$) were multiplied by $10^N$ and converted to an integer. The parameter $N$ (usually 5–7) is modified during the abstraction loop.
- All floating-point arithmetic operations were adjusted accordingly to accommodate the abstracted data type. For example, `floor(x)` was replaced by `x`.
- Where appropriate, floating point arguments for the support functions were taken apart (Figure 1) and the individual parts were passed as arguments. For example, `sin (double x)` was converted into `abs_sin (long l1, long l2)`. These individual variables `l1, l2` then were defined as symbolic for the test case generation. Another possibility for passing the individual parts could be `abs_sin (int exp, int sign, int m1, int m2)`.

[4]In many cases, KLEE uses random selection during the generation process. Thus, multiple runs can yield different sets of test cases.

This abstraction is extremely simple. However, it showed to be suitable for our analyzed examples. The reason is that most Boolean conditions in the code are not results of comparisons of results of floating point operations (e.g., `x*y` $\geq \pi$), but concern integer-based comparisons (e.g., `exp` $\geq$ `5`).

## 6.2 Experimental Results

Table 3 lists the test cases that have been generated for the sin function. A total 44 test cases are produced in about one second CPU time (on an Intel Macbook Pro). The table lists the input parameter $x_n$ (double) and the Error $E_{ref}$ with respect to the reference implementation, when the original `sin` function is executed on an Intel Macbook Pro. Test cases 3 and 22 trigger code for "inaccurate results" in the utility function; in these cases, the error calculation is not applicable N/A. Otherwise, the error is smaller than 1e-11 except for two test case, 31 and 32. It seems that the large error in these cases also correspond to the handling of large arguments, which lead to inaccurate results.

Table 3 shows that most arguments show up with a positive and negative sign. It also shows that the arguments of 0 and NaN ("not a number") show up multiple times. In these cases, KLEE produced test cases with different abstracted values. When combining these pairs $\langle l_1, l_2 \rangle$ into a double, identical values are the result.

The code coverage was measured with different tools. We used gcov (a part of the GNU C-compiler suite) for statement and branch coverage, and LDRA [17] to measure statement as well as MC/DC coverage. The initial results of measurements were surprising: whereas gcov reported 100% statement coverage, LDRA claimed that Line 42 in Listing 4 was not covered. Obviously, the empty statement in the default branch cannot be reached because of the value restriction of variable `bot2`. Thus, according to LDRA, this piece of code cannot obtain a 100% MC/DC coverage. Interestingly, if Line 42 is removed, 100% statement coverage is obtained, but a branch of the case statement is uncovered; hence again no full MC/DC coverage.

This observation is an indication, how careful unit testing has to be performed, as different tools can have subtle differences in their semantics of certain language constructs. The C case statement is such a typical example, which also produces detailed discussions in coding standards.

# 7. EXAMPLE II: TABLE LOOKUP

One of the most common block types in model-based systems like Simulink is the "Table Lookup" or interpolation block. Given an input $u$, it calculates an approximation of $f(u)$, whereby values of $f$ for monotonously increasing values of $x$ are given. In other words, a vector $\vec{x} = \langle x_0, x_1, ..., x_{n-1} \rangle$

| $n$ | $x_n$ | $E_{ref}$ |
|---|---|---|
| 1 | 0.0488281250 | 1.9e-11 |
| 2 | 0.048828145354091221 | 1.9e-11 |
| 3 | 5902958103587056517 12.0 | N/A |
| 4 | 137438954176.75451660156250 | 0 |
| 5 | 131072.02110725082457065582 | 1.1e-16 |
| 6 | 1.021675541361788 | 0 |
| 7 | 2048.0 | 5.6e-17 |
| 8 | 2097152.28180931787937879562 | 1.1e-16 |
| 9 | 1048576.0440363052263855934 | 1.1e-16 |
| 10 | 512.011448051895968092 | 0 |
| 11 | 8192.0103308404868585058 | 1.1e-16 |
| 12 | 8192.0195236411855148617 | 0 |
| 13 | 2097152.13230490731075406075 | 0 |
| 14 | 68719477974.53822326660156250 | 0 |
| 15 | 68719484038.4492340878906250 | 1.1e-16 |
| 16 | 1.022142187527940 | 0 |
| 17 | 8192.0189697450514358934 | 0 |
| 18 | 137438958099.9297790527343750 | 3.5e-18 |
| 19 | 32.070940668450703 | 1.1e-16 |
| 20 | -0.0488281250 | 1.9e-11 |
| 21 | -0.048828146286775316 | 1.9e-11 |
| 22 | -5902958103587056517 12.0 | N/A |
| 23 | -1.017512665761998 | 1.1e-16 |
| 24 | -4096.026914338559436146 | 1.1e-16 |
| 25 | -2199023747489.17871093750 | 1.1e-16 |
| 26 | -137438977261.382751464843750 | 0 |
| 27 | -2097152.39780779741704463959 | 2.2e-16 |
| 28 | -2097152.28940287604928016663 | 1.1e-16 |
| 29 | -2097152.37323756702244281769 | 0 |
| 30 | -524288.0 | 0 |
| 31 | -137438963969.7497253417968750 | 3.4e-06 |
| 32 | -68719489512.50517272949218750 | 3.8e-06 |
| 33 | -1.01257808057709 | 0 |
| 34 | -1.017452079403846 | 1.1e-16 |
| 35 | -33554434.43163714557886123657 | 0 |
| 36 | -32.0541842897661127 | 1.1e-16 |
| 37 | -524288.05263395013753324747 | 5.6e-17 |
| 38 | -1.0749030579782 | 1.1e-16 |
| 39 | 0.044409 | 0 |
| 40 | 0.0 | 0 |
| 41 | 0.0 | 0 |
| 42 | inf | NaN |
| 43 | NaN | NaN |
| 44 | NaN | NaN |

**Table 3: Generated Testcases for the *sin* support routine, and the error with respect to a reference implementation.**

of monotonously increasing values and a vector of function values $\vec{F} = \langle f(x_0), \ldots, f(x_{n-1}) \rangle$ are given. Whenever the code for the table lookup block is executed with a value $u$, the following steps are performed:

1. if $u$ is outside the range of $\vec{x}$, return $f(x_0)$ or $f(x_{n-1})$,
2. otherwise, determine the index $0 \leq i < n$ such that $x_i \leq u \leq x_{i+1}$, and
3. calculate the table lookup value as $\frac{1}{2}(f(x_i) + f(x_{i+1}))$ or by linear interpolation.

For efficiency reasons, step (2) is usually implemented using an iterative binary search.

We have analyzed a generic version of a 1-dimensional table lookup function, which is somewhat similar to Mathworks `rt_look.c`.[5] Listing 2 shows a sketch of such a function.

The original code contains an optional assertion _assert (...) , which can check the loop invariant during runtime. For the use of this code in a safety-relevant embedded environment, software tests must be carried out that fully cover this support routine. We are again using the tool KLEE to automatically generate test suites that provide full code coverage. In this case, we generate test cases for the following two scenarios:

1. given a vector $\vec{x}$ of monotonously increasing values and vector $\vec{F}$, find values for $u$ such that all paths in the look-up support function are covered, and
2. given the length $n$ of interpolation table, generate pairs $(\vec{x}, u)$ such that the code is fully covered.

In both cases, we assume, without loss of generality, $F$ to be the identity function, i.e., $f(ind) = ind$.

**Listing 2: Simplified Pseudo-Code for Table Lookup.**

```
double lookup(double *x, double *f,
  int len, double u){

if (u <= x[0]) // outside the table (left)
  return f[0];
else if (u >= x[len-1]) // outside (right)
  return f[len-1];
else
  for(;;){    //do binary search
    _assert( (x[bot] < u) && (u < x[top]));
    ind = (bot + top)/2;
    if (...
        top = ...
        bot = ...
    else
        return f[ind];
    ...
    }
}
```

For this function we again used our abstraction mechanism as discussed in Section 6 above. The driver code to force KLEE to generate test cases is very straight-forward. For scenario (1), Listing 3 shows the code.

**Listing 3: KLEE Test Driver**

```
main(){
  const static int x[] = {-2,-1,0,1,2};
  const static int F[] = {0,1,2,3,4};
  const int len = 5;
  int u, val;

  klee_make_symbolic(&u, sizeof(u), "u");
  val = lookup(x,F,len,u);
}
```

[5] `rtw_demos/rt_look.c` is found in Mathworks' distribution of RealTime Workshop.

Obviously, the test stimuli $u_i$ depend on the given vector $\vec{x}$. For our example, for $\vec{x} = \{-2, 0, 3, 5, 8\}$, the following six test cases are generated in less than 0.1 seconds:

$$u = \{-2147483648, -1, 0, 2, 4, 6, 8\}$$

Again, abstraction is necessary to achieve full coverage. In particular, the minimal distance between $x_i$ and $x_{i+1}$ in the abstracted program must be at least 2 in order to trigger the divide-and-conquer paths in the code. Otherwise, a vector like $\vec{x} = \{-2, -1, 0, 1, 2\}$ would only generate an incomplete set of 5 test cases. The use of a scaling factor during abstraction easily avoids that problem.

All these test cases have been generated with the assertion in the code turned off. When the assertion is turned on, it is textually replaced by a conditional statement that aborts the execution if the assertion is not met. Interestingly, KLEE still finds a test set, which covers the code to a full 100%. This indicates that there exist stimuli $u$, which, for a given table $\vec{x}$ cause the abortion of the execution. In an embedded system, such a behavior could have disastrous consequences. A closer look at the code reveals that the actual binary search loop is correct, but the assertion in `rt_look.c` is wrong (R2011b).

Our second scenario produces test cases for the table lookup, which are independent of the actual interpolation table $\vec{x}$. Rather, stimuli in the form of a vector and a lookup value $u$ are produced as test cases. This means that we force KLEE to also handle all the values of $\vec{x}$ symbolically. Table 4 shows a generated test suite for a vector of 5 elements. By definition of the lookup function, the elements of $\vec{x}$ must have monotonously increasing values. In order to enforce this precondition, we simply add this constraint to the code in our test driver:

```
if(x[0] < x[1] && x[1] < x[2] && ..)
  r=lookup(..);
else
  r=-1;
```

Thus KLEE generates testcases for covering the actual lookup plus test cases for yielding a dummy result of $r = -1$. Table 5 shows the number of all test cases $C_0$ and valid test cases $C$ to cover the lookup function, generated for different numbers of $n$, as well as the execution time (on a 2.4GHz Macbook Pro in seconds). Although it is obvious, that, due to the infinite loop, no finite set of test cases is sufficient to fully test this code, we demonstrated that automated test case generation can be used to easily generate sets of fully covering test cases for commonly used lengths of interpolation tables.

| $n$ | $C_0$ | $C$ | t[s] |
|---|---|---|---|
| 5 | 15 | 11 | 0.3 |
| 10 | 32 | 9 | 1.2 |
| 20 | 66 | 19 | 4.6 |
| 30 | 96 | 29 | 10.2 |
| 100 | 330 | 231 | 130 |

**Table 5: The number of test cases generated for different lengths of the lookup table. $C_0$ is the number of test cases generated by KLEE, $C$ is the number of valid test cases (i.e., vectors with increasing values), and run-time $t$ for their generation.**

# 8. CONCLUSIONS

Despite numerous software development standards and processes, numeric support functions are often left out in safety considerations. Even if these numeric support functions are considered "proven in practice", subtle differences in target architecture or configuration can yield bad surprises. A full test on the model level does not suffice if there is a chance that support functions might work incorrectly.

In this paper, we describe an approach of using KLEE to automatically generate suitable sets of test cases. We use abstraction to overcome KLEE's weak handling of floating point numbers. Of course, in such an environment, the test case generator should be complete and correct, properties, which are in general lost with our abstraction. However, we are not using KLEE's results as oracle and the degree of coverage is tested on the target code by an external tool. Redundant test cases are not harmful. Nevertheless there are strong economical needs to reduce number of tests. Again, redundant test cases can be easily detected.

Our method of automatically generating sets of test cases for a multitude of numerical support function is is in particular of interest for all model-based approaches and applications where test cases are to be generated from a specific formal test specification, e.g., as described in the VASE method for the qualification of software development tools in automotive applications [19].

# 9. REFERENCES


[1] Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves and van Renesse [5], pp. 209–224

[2] Charette, R.: This car runs on code (2009). URL `http://spectrum.ieee.org/green-tech/advanced-cars/this-car-ru%ns-on-code`

[3] Cody, W.J.: Software Manual for the Elementary Functions. Prentice Hall (1980)

[4] Commission, I.E.: Iec 61508 : Functional safety of electrical/electronic/programmable electronic safety-related systems. 1st IEE Automotive Electronics Conference (HCSS+++++), 7–13 (2005). URL `http://link.aip.org/link/IEESEM/v2005/i10926/p7/s1&Agg=doi`

[5] Draves, R., van Renesse, R. (eds.): 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. USENIX Association (2008)

[6] dSPACE: Targetlink. URL `www.dspaceinc.com/en/inc/home/products/sw/pcgs/targetli.cfm`

[7] of Electrical, I., Engineers, E.: Ieee standard for floating-point arithmetic (2008)

[8] ETAS: Etas - ascet software products - software products & systems. URL `www.etas.com/en/products/ascet_software_products.php`

| # | $u$ | $\hat{x}$ |
|---|---|---|
| 1 | 0 | -2147483648, -2147483647, -2113929216, 0, 1 |
| 2 | 4098 | 4, 6, 7, 4096, 4099 |
| 3 | 264 | 64,265,291,8481,131337 |
| 4 | 10 | -1610612732, -1610612730, 10, 536870920, 536870922 |
| 5 | -2147483639 | -2147483648, -2147483646, -2147483639, -1610612728, -1610612727 |
| 6 | -2147483643 | -2147483648, -2147483646, -2147483645, -2147483644, -2130706427 |
| 7 | -2147479552 | -2147483648, -2147483646, -2147483645, -2147479552, -2147479551 |
| 8 | -2147481472 | -2147483136, -2147481471, -2147481464, -1610602368, 262273 |
| 9 | -2147483647 | -2147483648, -2147483647, 0, 4, 6 |
| 10 | 6 | 0, 1, 2, 3, 6 |
| 11 | 0 | 0, 1, 4, 6, 7 |

**Table 4: Generated stimuli for $n = 5$. All paths of the code are fully covered by this test set.**


[9] fdlibm. URL http://www.netlib.org/fdlibm/

[10] Frank, S., Grabmüller, M., Hofstedt, P., Kleeblatt, D., Mai, P., Pepper, P., Schneider, S.A.: Safety of compilers and translation techniques - status quo of technology and science. In: Automotive – Safety & Security 2008 Sicherheit und Zuverlässigkeit für automobile Informationstechnik, ???, pp. 45–59. Shaker Verlag, ??? (2008). DOI http://www.shaker.de/de/content/catalogue/index.asp?lang=de&ID=8&ISBN=9%78-3-8322-7681-2&search=yes. URL ???

[11] Hart, J.F., Cheney, E.W., Lawson, C.L., Maehly, H.J., Mesztenyi, C.K., Rice, J.R., Thacher, J.H.G., Witzgall, C.: Computer Approximations. SIAM Series in Applied Mathematics. John Wiley and Sons (1968)

[12] Hauser, J.: Softfloat. URL http://www.jhauser.us/arithmetic/SoftFloat.html

[13] Huckle, T., Schneider, S.A.: Numerische Methoden: Eine Einführung für Informatiker, Naturwissenschaftler, Ingenieure und Mathematiker. Springer (2006)

[14] ISO: International standard iso 26262 road vehicles — functional safety, first edition 2011-11-15. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_t%c_browse.htm?commid=46752

[15] Javapathfinder. URL http://babelfish.arc.nasa.gov/trac/jpf

[16] The klee symbolic virtual machine. URL http://klee.llvm.org

[17] LDRA: Ldra tool suite. URL http://www.ldra.com

[18] The llvm compiler infrastructure. URL http://llvm.org

[19] Lovric, T., Mai, P., Schneider, S.A.: The validation suite approach to safety qualification of tools. SAE Technical Paper 2009-01-0746 (2009)

[20] Mathworks: Design verifier. URL www.mathworks.com/products/sldesignverifier/

[21] Mathworks: Toolvalidierung: Softwareentwicklung für funktionscodes in der automobilindustrie. URL http://www.mathworks.com

[22] Miller, J.C., Maloney, C.J.: Systematic mistake analysis of digital computer programs. Commun. ACM **6**(2), 58–63 (1963)

[23] Modelica.Association: Modelica and the modelica association. URL http://modelica.org/

[24] Netlib repository at utk and ornl. URL http://www.netlib.org/

[25] OSEK: Osek vdx portal. URL http://www.osek-vdx.org

[26] Overton, M.L.: Numerical computing with IEEE floating point arithmetic - including one theorem, one rule of thumb, and one hundred and one exercices. SIAM (2001)

[27] Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In: Proc. of ISSTA (2008)

[28] Reactis: Software testing and validation with reactis. URL http://www.reactive-systems.com/

[29] RTCA: Do-178b: Software considerations in airborne systems and equipment certification (1992). URL http://www.rtca.org

[30] RTCA: Do-178b: Software considerations in airborne systems and equipment certification (2011). URL http://www.rtca.org

[31] T-VEC: URL http://www.t-vec.com

[32] Wikipedia: Coverage testing. URL http://en.wikipedia.org/wiki/Coverage_testing

**Listing 4: Simplified Code for Numeric Support Function *sin*.**

```
sin (double x) {

md.d = x;                    // break the argument up into parts
xexp = (int) md.b.exp;
                             // handle boundary conditions
if (xexp == IEEE_MAX) {        // x is Not-a-Number or infinity
   if (md.b.m1 || md.b.m2)
      return x;              //  x is Not-a-Number
   else
      return NaN;            // 0.0/0.0
   } else if (xexp == 0) // x is infinity or denormalized
      return x;
   else if (xexp <= (IEEE_BIAS - IEEE_MANT - 2))
      return x - x*x;            // x is very small
   else if (xexp <= (IEEE_BIAS - IEEE_MANT/4))
      return x - x^3/6;    // x is small
   }
if (x < 0) {
      set sign to 1; // negative
      x = - x;
   }
// map to range of argument to x <= pi/2
if (xexp < IEEE_BIAS) { // 2^52/4 < x < 1
   skip;                 // x already < pi/2
else if (xexp <= (IEEE_BIAS + IEEE_MANT)) {
    xm = b * 2_pi_hi + 1/2c
    xn.d = xm + mag52
    bot2 = xn.b.m2 & 3u
    // split xm into top 26 and bottom 26 bits
    split(a1, a2, xm);
    exactmul2(x3,x4, xm,a1,a2, pi2_hi,pi2_hi_hi,pi2_hi_lo);
    exactmul2(x5,x6, xm,a1,a2, pi2_lo,pi2_lo_hi,pi2_lo_lo);
    x = ((((x - x3) - x4) - x5)- x6) - xm*pi2_lo2;
            // reduce to 0 <= x <= pi/2

    switch(bot2) {
      case 0: if (x < 0.0) { x = -x; sign ^= 1; } break;
      case 1: if (x < 0.0) { x = pi2_hi + x; } else { x = pi2_hi - x; } break;
      case 2: if (x < 0.0) { x = -x; } else { sign ^= 1; } break;
      case 3: sign ^= 1; if (x < 0.0) {x = pi2_hi + x;} else {x = pi2_hi - x;}break;
      default: ;
      }
   } else { // 2^53 <= x
     return LOSS;
   }
x = x* _2_pi_hi;// map to range 0 <= x <= 1
if (x > X_EPS) {
   x2 = x*x;
   x = POLYNOM(x2); // Horner 7th degree
   } else {
   x = x * pi2_hi;
   }
if (sign) x = -x;
return x;
}
```

**Figure 3: The Pseudo-Code for the Argument Qualification and Polynomial Approximation of the Numeric Support Function** $(double)\sin((double)\,x)$**.**