8Cage: Lightweight Fault-Based Test Generation for Simulink

Dominik Holling
TU München, Germany
holling@cs.tum.edu

Alexander Pretschner
TU München, Germany
pretschn@cs.tum.edu

Matthias Gemmar
ITK Engineering AG
Rülzheim, Germany
matthias.gemmar@itk-engineering.de

ABSTRACT

Matlab Simulink models, mainly used for the specification of continuous embedded systems, employ a data flow-driven notation well understood by engineers. This notation abstracts from the underlying computational model, hiding run time failures such as over-/underflows and divisions by zero. They are often detected late in the development process by the use of static analysis tools on the completely developed system. The responsible underlying faults are sometimes attributable to a single operation in a model. 8Cage is an automated test case generator for the early detection of such single operation related faults. It is configurable to detect these faults and runs automatically in the background. It tries to find potentially failure-causing operations and generates a test case to gather evidence for an actual fault. 8Cage is usable by developing/testing engineers with knowledge of Matlab. It does not require an expert to perform result validation or fault localization.

1. INTRODUCTION

Matlab Simulink [5] is a development environment for embedded systems software. It aids in model-based architecting, designing, implementing and testing. Simulink models use a data flow driven block-based notation similar to wiring plans. Blocks execute functions such as arithmetic, accumulative, limiting and other operations to the data they are given. Each block has a specific number of input and outputs which are connected to other outputs and inputs. Each connection represents a data flow. Special I/O blocks let data flow into and out of the model. The notation helps mechanical/electrical engineers by giving well-understood model notations and enabling code generation from them. Thereby, programming efforts for continuous and hybrid systems are minimized while readability for engineers is ensured.

Such a model-based approach is widely employed for automotive drive trains, aerospace engine control and dentist drill controllers, among others. For building complete software systems, Matlab allows abstraction by encapsulating so-called subsystems. These subsystems can be used in a different model, allowing their composition in levels. The lowest level is the unit level, constituting a model without any subsystems, using only built-in blocks of the Simulink library. Each layer above the unit level represents an integration level where two or more unit or integration levels are composed. The uppermost level is the system level consisting of all unit and integration levels.

When developing embedded software, the development cycle usually starts with the implementation of units. Once a unit is finished, it can enter unit testing by a tester. The tester will typically perform a black-box functional test according to the unit's specification. Units are then composed until the system level is reached. Integration and system testing are performed alongside. In a last step, static analysis is typically performed on the production ready system to detect the aforementioned run time detectable failures. If such failures are detected, a trained expert needs to perform fault localization and present the result to the respective developer(s). When the faults are fixed, the development cycle restarts with unit and integration tests.

Unfortunately, many essentially avoidable faults involving only a single or a simple composition of blocks are detected only by static analysis at the end of a development cycle. This creates an overhead of at least one full development cycle. Thus, earlier detection would yield cost benefits. In addition, developer assumptions about certain blocks or block combinations may be verified only as late as Hardware-In-the-Loop (HIL) tests. HIL tests requires the software to be deployed on the target platform and are performed even later than static analysis.

Problem: While the data flow-driven notation yields great understandability, it hides—and is supposed to do so—the underlying computational model of processor and memory architecture. This regularly leads to problems with generated code. Certain faults are invisible or hard to detect by review at the model level. Such faults include run time detectable under-/overflows, divisions by zero and possibly infinite loops. In addition, it may be hard for engineers developing a system to assess if their assertions hold.

Further typical faults include exceeded ranges of I/O signals. These ranges are specified by a developer or imposed by physics. For instance, the revolutions per minute (rpm) of a gasoline combustion engine may range from 0 to 9,500. Thus, a 16-bit unsigned integer of range 0 to 65,535 is sufficient to hold the rpm signal's range. However, the rpm signal will never be larger than 9,500 when used/given as I/O. There may be assumptions in certain units that rely on

ASE'14, September 15-19, 2014, Vasteras, Sweden.
ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2648622.

this range. Thus, the rpm signal's range must be rigorously checked to never exceed 9,500. Typically, the ranges are also checked by static analysis late in the development process. Detecting failures caused by invalid/unspecified ranges on a unit level would yield early detection benefits as well.

Solution and Contribution: We developed the 8Cage tool, an automated test case generator for Matlab Simulink models based on general fault models [9]. To our knowledge, it constitutes the first explicit operationalization of fault models. We demonstrate its capabilities by detecting over-/underflows, divisions by zero or small numbers, exceeded I/O ranges, and violated developer assumptions concerning a single block. Developers/testers can specify their own fault hypotheses and have them automatically tested. If a possibly failure-causing block is present, 8Cage tries to produce a test case constituting a counter example. Feedback is directly given to the developer; faults are easily located. Thereby, it eliminates the need for expert analysis with static verification tools for the described faults and bridges the model/source-code level. *Because 8Cage only uses the implicit specification of what should not happen*, it can detect failures in a model *even without a specification.* Although the faults currently detected by 8Cage are related to single blocks, such faults are common and recurring problems in Matlab Simulink models.

2. APPROACH

Our approach is based on fault models [9]. A fault model is a transformation of a correct into an incorrect behavior description and/or a partition of the input space. In the case of 8Cage, the behavior description is a Simulink model. The image of the transformation defines a smell, and smells reveal likely failure-causing blocks. To gather evidence for the presence of an actual fault, a partition of the input space for its exploitation is created, and one potentially failure-causing test case is selected and executed.

In the following two exemplary fault models and the implementation of the approach are detailed. We present two steps: Fault model specification and automatic detection. Automatic detection again consists of smell detection, test data generation, and test case execution.

2.1 Example Faults

All Simulink blocks that can cause over-/underflows have a binary property called "saturate on integer overflow" (SIO), which can be set by the developer. Enabling this property generates an over-/underflow-preventing safety check for the respective block. Enabling this property for every possible block would avert the problem of over-/underflows completely. However, each check requires processing time and may lead to dead code. Standards such as MISRA-C [8] require avoidance of dead code. Developers must decide whether to enable the SIO property. 8Cage provides help by searching for a way to cause an over-/underflow, showing the need to not deactivate the SIO property.

As an example, the built-in Abs block of Simulink calculates the mathematical absolute value function of the input x (i.e. $|x|$). The code generated from this block is: `if (x < 0) return -x; else return x;`. This implementation of the absolute value function is efficient and works for all integer inputs except the most minimal one. Let x be a signed 8-bit integer. x then has a range from -128 to 127 making -128 the only integer without a positive counter-part. Due to -128 being stored in two's complement (`1000 0000b`) and negation of it (flipping all bits and adding 1) will yield -128 again. A developer assumption for using the absolute value could be the non-negativity of the output, which cannot be guaranteed for all integers.

As a further example, consider a division by a small fixed-point number. Many electronic control units (ECUs) do not contain a floating point unit due to cost reasons. Thus, fixed-point numbers are chosen to represent decimals. In Matlab, fixed-point numbers use integer data types and fix the precision (i.e. the number of digits after decimal point). An 8 bit unsigned integer x with 1 bit of precision yields a range from 0 to 127.5 where the bit of precision can express .0 and .5 after the decimal point [5]. Computing $64/x$, where x can assume any value in its range, can lead to an overflow if $x = 0.5$. This constitutes a multiplication by 2 with a desired result of 128 and an actual result of 0.

Both examples are typically detected late, using static analysis. Automatically detecting them *far earlier* yields time and cost benefits. The same holds true for other elementary faults such as dividing by zero or other single block related faults. Note that 8Cage is not limited to only these fault models, but has been and can further be extended to any failure/assumption caused/violated by a single Simulink block.

2.2 Fault model specification

8Cage allows to specify single block fault models by using the Simulink syntax and a small run time extension. Each Simulink block has properties such as its name, type and inputs/output with a specified data type comprising the static properties required by 8Cage. In addition, 8Cage requires knowledge about failure-causing input and/or output values at run time for each described block. These values do not necessarily need to be specified as absolute values (i.e. -128), but can also be specified as relative values (e.g. $MinValue$). The absolute value fault model above can be specified by telling 8Cage to locate blocks of type "Abs." Its properties should be "Saturate on integer overflow" set to value "off." The input value should have an integer type and the value should be $MinValue$. Division by a small fixed point number can be specified analogously.

Developers/Testers can specify these fault models using a custom XML schema. Thus, the specification of developer assumptions on single blocks is possible.

2.3 Automatic detection

8Cage allows the operationalization of provided fault models by performing automated smell detection, test data generation and test case execution. The purpose of the automatic detection is to find model smells and to generate evidence that these smells are actual faults.

Configuration: The configuration parameters of 8Cage include the model file, its respective configuration files and the (sub-)system to test. In addition, the number of steps to simulate the model needs to be provided.

Smell detection: During smell detection, the model is loaded into Matlab and static block properties are checked. These include block types/names and I/O data types. For the absolute value fault model, smell detection searches for blocks of type "Abs." All found "Abs" blocks are checked for their SIO properties to be off and integer input data types. If a block matches, it is marked in the model and

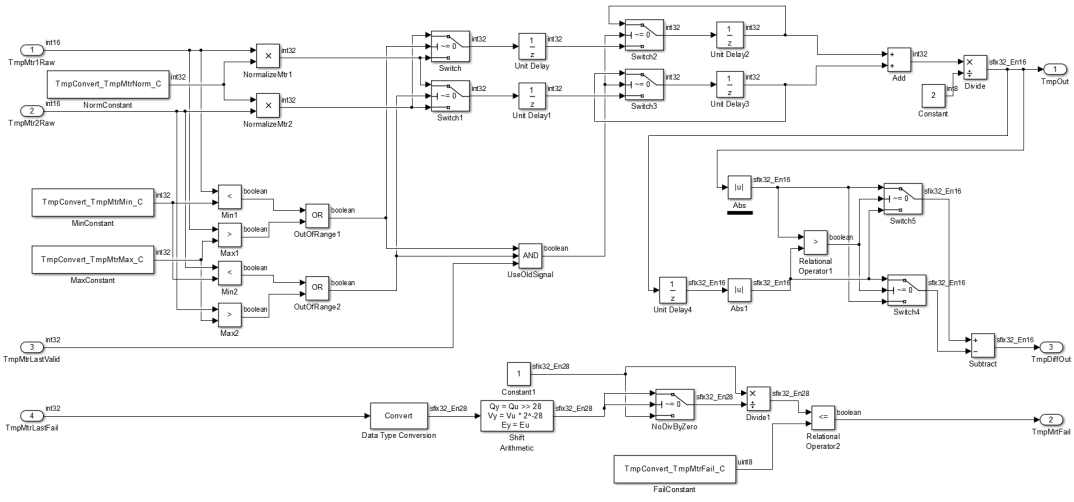

Figure 1: A typical Matlab Simulink unit

code is generated to perform the next step. All found blocks constitute model smells.

Test data generation: This step checks dynamic properties of I/O values. It uses the markers in the model to create markers in the derived code which it symbolically executes using KLEE [1]. KLEE is directed toward the markers and generates test input data. Currently, only test input data is generated since Matlab already provides an oracle for over-/underflows. However, when checking for signal ranges or developer assumptions, a respective oracle is generated.

Test case execution: A test case is created to gather evidence for the presence of an actual fault. The test case is created in a test execution engine for Simulink models. The created test case is run as a Model-in-the-Loop (MIL) test as the single block faults will already fail within Matlab itself. The results are determined and stored for reporting.

Report generation: The tool generates a report concerning found smells and possible evidence for the presence of an actual fault. This report details results of the test cases and contains them for developer/tester reproducibility.

The configuration and smell detection steps are performed directly in Matlab Simulink catering to the developer/tester familiarity. Smells can be examined in the model for quick fixes. Configuration and detected smells are input to test data generation and test case execution steps. Both are performed centrally on machines running Matlab, KLEE and the test execution engine.

3. EXAMPLE

To demonstrate 8Cage, a full automatic detection run on a representative unit (see figure 1) is presented. Its representativeness is the result of a survey of multiple software systems at our industry partners. It represents a unit to read a temperature value from two redundant sensors. The inputs TmpMtr1Raw and TmpMtr2Raw (top left) are the raw temperature values. These values are normalized and concurrently checked to be in a valid range. In case a sensor value is not within range, the other sensor's value is used instead. If both sensor values are currently invalid and the last signal was valid (i.e. input TmpMtrLastValid (bottom left) is set), the last valid sensor readings are used. Finally, the average of the two sensor values is calculated by adding and dividing them by two, constituting the first output TmpOut (top right). A difference between the last and the current value of TmpOut is calculated by taking the absolute value of both sensor values and subtracting the smaller from the larger value. This difference becomes the second output TmpDiffOut (bottom right). In addition to the raw temperature values, the sensors also report their failure using the input TmpMtrLastFail (bottom right). These values are converted to the required format and checked against a bit mask. If the bit mask matches, a failure is reported.

During configuration, the model and its model parameters are selected as system to test. Fault models are chosen (e.g. absolute value overflow) and the number of simulation steps is set (e.g. 4). After setting these parameters, 8Cage performs smell detection, test data generation and test case execution for the chosen fault models. The absolute value fault model's smell detection will reveal several matching absolute value blocks with integer inputs. Selecting the underlined block named Abs in figure 1 (middle right), test data generation finds a minimum input of -32768 possibly leading to a failure. An actual fault in the model is detected as the test case results in an overflow. All other fault models are handled the same. Finally, a report will be issued to the requester containing the detected absolute value overflow.

A video showing the exemplary run employing multiple fault models and a final report can be found at `https://www22.cs.tum.edu/en/tools/8Cage`.

The overall execution time for finding potentially failure-causing blocks in smell detection is 7 seconds. Generating code from the model requires 45 seconds. Both these tasks are run in and limited by Matlab. Executing the generated test cases takes 4 seconds per test case caused by the test execution engine. Symbolic execution for each potentially failure-causing block uses 90 seconds of single threaded computation time. Adding only one input signal increases this time to 10 minutes. Performing 10 instead of four steps requires 7 hours of computation time. We are currently investigating ways of improvement in this area.

4. RELATED WORK

8Cage is related to static analysis tools. Tools using abstract interpretation are Astrée [3] and Polyspace Code Prover [10]. Both can detect run time failures such as over-/underflows and out of bounds pointers as well as potentially infinite loops and potentially dead code. However, it requires an expert to prepare a software system for analysis in these tools as they work on a C/C++ code level. Thus, analyzing each unit often is deemed excessive work and only the completely developed system is analyzed. Results are returned by coloring source code—code verified to cause a run-time failure; code that may cause a failure; code that will not cause a problem; and code deemed dead. Unfortunately, because of approximations, often large parts of the source code is marked as potentially failure causing. Then, an expert has to walk through the C code. Finally, faults must be localized to go from the reported failures to the actual faults. This result and suggestions to fix are then given back to the developers (or testers in some cases). 8Cage allows the developer/tester to start the automatic detection and yields results directly traceable to blocks in the model. Thus, no expert is involved in the analysis. However, 8Cage only detects faults related to single blocks on a unit/integration level. This makes the use of static analysis tools still advisable to detect other faults at a later stage.

Static analysis is also performed by smell finding tools such as Lint [4] and Polyspace Bug Finder [10]. They return possible problems for C code without dynamic execution. 8Cage also performs a lint-like smell detection. However, 8Cage always aims to gather evidence for an actual problem to be present by creating a test case and executing it. Smell detection is also performed by the Simulink Model Advisor [6] leading to similar results as 8Cage's smell detection, but without test case generation.

Tools related to 8Cage in the field of model-based testing of Simulink models are, among others, Simulink Design Verifier [7], Reactis [11] and TPT [12]. These tools allow the manual specification and automated execution of test cases. Design Verifier and Reactis can generate test cases for coverage. In addition, Reactis can generate random test inputs. TPT uses a graphical test case notation abstracting from actual I/O in a keyword-driven way. In contrast, 8Cage generates test cases that target specific faults within the Simulink model by operationalizing fault models.

5. DISCUSSION AND CONCLUSION

We have described an operationalization of fault models using 8Cage. 8Cage currently detects faults related to single blocks in Matlab Simulink models. These faults are described as fault models [9] using parts of the Simulink model syntax with a minor extension. They are operationalized by 8Cage in automatic detection consisting of smell detection, test data generation and test case execution. During smell detection, the static aspects of a block (i.e. the image of the transformation) such as its type, name and properties are utilized to find potentially failure-causing blocks (i.e. model smells). Test data generation performs a symbolic execution using the dynamic aspects to search for failure-causing inputs to the model (i.e. an input space partition). Test case execution creates a test case using the failure-causing inputs and provides evidence for a fault.

Scalability of symbolic execution is a concern. Test data generation takes a significant proportion of the analysis time. However, experience shows that almost all single block faults can be found within the first 5 time steps, a pattern also exploited by unfolding heuristics in static analyzers. Test data generation has to be performed for every potentially failure-causing block. Because test generation for each block is independent from another, it can be parallelized.

8Cage is currently unable to perform symbolic execution on floating points numbers. As electronic control units in the automotive area often lack floating point units because of cost, this is not a drawback for 8Cage yet. However, it is projected that floating point units will be built in within a few years. Thus, we are currently looking into a floating point version of KLEE [2].

8Cage can solely detect prespecified faults. New or different faults need to be specified to be detectable. Thus, the library of fault models needs to be maintained.

We are unaware of tools to detect faults related to a single block at an early stage and developers/testers presented with 8Cage had a very positive response. In particular developers could detect these faults before even checking their model into version control.

We currently enhance 8Cage to feed results directly back into Simulink. This helps to cater to developers/testers only familiar with Matlab even better. In addition, we are running analyses on further real-world models. Our ideas likely generalize to integration faults, and we are currently working on integrating composite and integration fault models involving multiple Simulink blocks.

6. REFERENCES

[1] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, 2008.

[2] P. Collingbourne, C. Cadar, and P. Kelly. Symbolic crosschecking of floating-point and simd code. In *Proc. EuroSys*, 2011.

[3] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proc. PLAS*, pages 21–30, 2005.

[4] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[5] MATLAB. *version 7.11.1.866 (R2010b SP1)*. The MathWorks Inc., Natick, Massachusetts, 2011.

[6] MATLAB. *Consult the Model Advisor*. The MathWorks Inc., Natick, Massachusetts, 2014.

[7] MATLAB. *Simulink Design Verifier*. The MathWorks Inc., Natick, Massachusetts, 2014.

[8] MISRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, Oct. 2004.

[9] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar. A generic fault model for quality assurance. In *Proc. MODELS*, pages 87–103. 2013.

[10] P. C. Prover. *Static Analysis with Polyspace Products*. Mathworks, June 2014.

[11] Reactis. *Reactis Product Suite*. Reactive Systems Inc., Cary, North Carolina, 2014.

[12] TPT. *TPT - Time Partition Testing*. Piketec Gmbh, Berlin, Germany, 2014.