Heuristics for generating abstract test cases from $\mathcal{R}$ebeca model

Þröstur Thorarensen (throstur11@ru.is)

April 13, 2015

Abstract

Software testing can be a difficult task for humans and is a prime candidate for automation.Concurrent systems are no exception, even when employing paradigms such as the Actor model. Model checking goes a long way to verifying software correctness, but models always abstract away from the implementation. Mutant testing is a promising method to pinpoint programming errors, by combining the abstract model of a system with mutants of the implementation, it is possible to automatically generate an extensive suite of test cases with ease. The research demonstrates that software testing can easily be supplemented by automation and that even the simplistic methods employed in this study are capable of producing reliable results.

Contents

1 Introduction

A model describing a System Under Test (SUT) is usually an abstract, partial presentation of the SUT's desired behavior. Model-based testing is using such a model of the SUT to generate abstract test cases and then mapping those abstract test cases to executable test cases based on the back-end code. Test cases derived from such a model are at the same level of abstraction as the model. An abstract test case cannot be directly executed against a SUT and an executable test suite needs to be derived from a corresponding abstract test suite.

Model checking is a method of formally verifying finite-state concurrent systems according to a given specification. The specification is generally expressed as temporal logic formulas which are verified either with explicit-state model checking or symbolic model checking. Symbolic model checking is more efficient states can be represented as sets of states rather than as single states.

$\mathcal{R}$ebeca is an actor-based modeling language designed in an effort to bridge the gap between formal verification approaches and real applications. $\mathcal{R}$ebeca introduces the ability to analyze a group of reactive objects as single components of a system in the actor model.

Since we cannot test software with all inputs, **coverage criteria** are used to decide which test inputs to use. The software testing community believes that effective use of coverage criteria makes it more likely that test engineers will find faults in a program and provides informal assurance that the software is of high qualityand reliability.

Mutation testing is used to design new software tests and evaluate the quality of existing software tests. It involves creating *mutants* modified versions of the SUT that are based on *mutation operators*. Each mutant $m \in M$ where M is the set of mutants for a given artifact will lead to a test requirement[1] and yields more test requirements than any other test criterion in most situations (provided that the mutation operators are well-designed). In practise, if software contains a fault, there will usually be a set of mutantns that can only be killed by a test case that also detects that fault [Ammann and Offut, 2008].

2 Contribution

Coverage criteria is necessary when testing all but the most trivial software. Coverage critera can however be difficult to decide upon for complicated systems. Software developers and testers are often faced with the task of testing software. This can be error-prone and time consuming, making testing a useful target for automation.

In this study, we modeled a case study using $\mathcal{R}$ebeca and employed model checking to ascertain that the model is faithful to the specification. Following this, we generated the state space of the model and extracted (random) traces through the graph. Although employing some heuristics for which traces are extracted would have been nice, this has been left as future work. The extracted traces were transformed into test cases for the SUT.

We used Offutt et al. to generate mutants for the SUT. Each mutant was evaluated by the test cases in an attempt to find mutants that could not be killed. These mutants act as the coverage criteria for the randomly generated traces.

[1] Criterion 5.32 on Mutation Coverage

3 Related Work

TODO!

4 Method

4.1 Modeling the System Under Test

The case study intended as the SUT is the NASA GMSEC Message Bus, the specification of which was provided by Fraunhofer CESE. The model of the system was implemented as a `reactiveclass` in $\mathcal{R}$ebeca. Simply modeling the SUT as a `reactiveclass` was not sufficient for our purposes - the system is reactive and as such requires inputs from external entities to avoid idling. For this reason, a `reactiveclass` modeling a user of the SUT was implemented, non-deterministically sending messages to the SUT. To avoid a combinatorial explosion in generating the state space, the amount of messages sent must have a reasonably low upper bound - 20 was selected for this study.

It should be noted that an actual implementation of the SUT was not used in this study. Instead, the model was re-implemented as a Java class following the specification of the system[2]. This was done for a few reasons, notably giving control over used frameworks to the researchers and to double-check that the model is faithful to the specification. The SUT implementation was implemented as a Akka Java actor, as the Akka Testkit framework provides excellent tools to test actor based systems.

4.2 Coverage Criteria

Due to temporal constraints on the study, the coverage criteria for the generated test cases was set at 1000 random traces extracted from the state space of the model, in which only messages to and from the SUT are considered (all other transitions in the state space are considered as τ transitions and were discarded from the test case generation). The state space of the model was exported from Afra ($\mathcal{R}$ebeca IDE) as an `.aut` [3] file suitable for visualization with software such as CADP.

4.3 Generating Test Cases

4.3.1 Abstract Test Cases

Random traces were extracted from the state space. The traces act as abstract test cases for the SUT, as the behavior of an actor based model to an outside observer can be fully modeled based only on the messages that are being sent between actors.

In order to extract individual traces from the $\mathcal{R}$ebeca model, we implemented simple Java software to create a graph of the state space based on an `.aut` file as input, with the output being a list of traces through the graph. Branching decisions in the state space were made at random. The traces

[2]Using this approach proved to be quite useful. We were able to pinpoint an error in the model due to incorrect handling of an edge case which was correct in the implementation (which our properties for model checking did not cover). Although a complete implementation should also catch this, using the complete implementation increases the complexity of the project and reduces the amount of time that could be dedicated solely to the research topic at hand.

[3]ALDEBARAN/*AUTomaton* file format. Specification: `http://www.inrialpes.fr/vasy/cadp/man/aut.html#sect2`

extracted from the state space are then used as abstract test cases. The code can be found in the `trace-extract` package of the `trace-mutants` GitHub repository.

4.3.2 Concrete Test Cases

The `trace-extract` package also includes a Java class for transforming the abstract test cases to concrete test cases. The `TestGenerator` class generates a Scala test spec for the Akka Testkit framework[4].

4.4 Mutant Generation

Mutants were generated using the muJava Offutt et al. [2015] mutation system for Java programs. Some slight modifications were made to the source code of muJava to facilitate mutant compilation.The artifacts of the modifications are available in the `gen-mutants` package of the `trace-mutants` GitHub repository. Every available mutant operator was applied to the original source, generating a total of 316 mutants.

4.5 Test Case Evaluation

The test cases can be evaluated with the `evaluate-mutants` package of the `trace-mutants` GitHub repository. `evaluate-mutants` uses the file system to set up individual environments for each mutant to be compiled and tested in and then uses Scala-SBT to execute the test cases.

The result of a test case evaluation can give different information depending on the results.
If all mutants are killed, either:

- The coverage of the test suite is insufficient; or
- The SUT conforms to the model and the test suite satisfies the coverage criteria.

Otherwise:

- There is a bug in the SUT, representative of a mutant that was not killed; or
- There is a discrepancy between the SUT and the model.

5 Results

Although muJava attempted to generate 356 mutants, only 316 mutants of our implementation of the SUT could be compiled. 277 of those mutants modified the `onReceive(Object message)` method whereas 79 modified other parts of the program, including the message constructors and the overloaded `boolean equals(Object other)` methods.

[4]The `TestGenerator` class hard-codes the messages that should be sent and received by the SUT as a result of fast prototyping, but the code is available on GitHub

We found that 50 random traces from the state space did not generate a sufficiently diverse set of traces to reliably kill all mutants. In one run, 5 mutants survived the 50 randomly chosen traces: `COR_21, ODL_33, SDL_17, SDL_19, VDL_14`. The `SDL` mutants modified the `boolean equals(Object other)` method whereas the other mutants modified the `onReceive(Object message)` method. `ODL_33` and `VDL_13` resulted in the same mutation (removing a predicate from an `OR` conditional) whereas `COR_21` replaced the `OR` operator to an `AND` operator.

Similar results were encountered during development of the test framework. Since randomness plays such a large part in determining whether or not a set of traces through the state space graph provides adequate coverage, it was decided to generate 1000 random traces. We found that generating 1000 traces resulted in all mutants being killed every time, although theoretically it is of course always possible to generate unhelpful test cases when using randomness.

We were able to find one bug discrepancy between the model and the implementation, in which an edge case was being handled incorrectly in the model but correctly in the implementation. This demonstrates at least some usefulness with our method.

6 Discussion

Bar!

7 Conclusion

Foo!

8 Future Work

TODO: Mention that a complete implementation should be used instead of a mock!

References

Paul Ammann and Jeff Offut. *Introduction to software testing.* Cambridge University Press, New York, 2008. ISBN 978-0-521-88038-1.

Fraunhofer CESE. Fraunhofer center for experimental software engineering. `http://cs.gmu.edu/ offutt/mujava/`, 2015.

Jeff Offutt, Yu-Seung Ma, Yong Rae Kwon, and Nan Li. mujava. `http://cs.gmu.edu/ offutt/mujava/`, 2015.

Scala-SBT. Simple build tool. `http://www.scala-sbt.org/`, 2015.

Þröstur Thorarensen. Github repository for trace-mutants. `https://github.com/ThrosturX/trace-mutants`, 2015.