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

Þröstur Thorarensen (throstur11@ru.is)

July 5, 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.

A $\mathcal{R}$ebeca model was created based on a message busspecification, after which Afra was used to generate the state space as an `aut` file. Traces were then extracted from the state space to be used as abstract test cases. A test generator converted the abstract test cases into concrete test cases. The actual implementation of the message bus was input into muJava to generate mutants of the message bus. Each concrete test case was executed for every mutant until all mutants could be killed - if a mutant passed every test case, it cannot be considered killed, generally revealing a bug.

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 quality and 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 mutants 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 muJava 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.

The goal of testing each mutant is to attempt to uncover a bug in the implementation or a disparity

[1] Criterion 5.32 on Mutation Coverage in Ammann and Offut [2008]

between the implementation and the specification. Figure 1 depicts the entire process from start to finish, taking a specification and implementation as inputs and returning the results of evaluating each generated mutant with each test case.

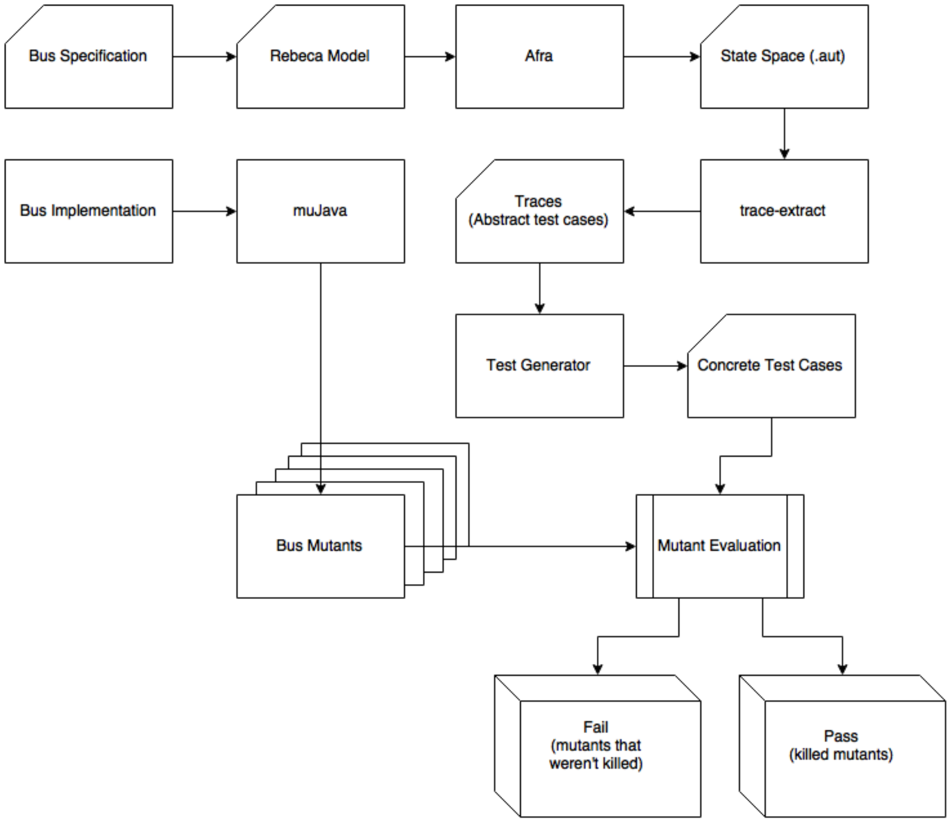

Figure 1: Flow chart of the process for validating an implementation based on a specification.

3 Related Work

Sirjani and Jaghoori [2011] discuss $\mathcal{R}$ebeca after 10 years of experience in analyzing actors. The paper introduces $\mathcal{R}$ebeca in detail along with explaining the supporting model checking tools. The paper focuses on state-space reduction techniques which may come in handy for future work, along with Tasharofi's dissertation on efficient testing of non-deterministic actor programs which introduces novel solutions in reducing the number of explored interleavings of a non-deterministic concurrent actor system [Tasharofi, 2014].

Ammann and Offut [2008] go into great detail in explaining the basics of software testing, covering concepts used in this research such as coverage criteria and mutants. Furthermore, Offutt et al. [2015] have developed an excellent tool for Java for generating mutants, which lead to test requirements for syntax-based coverage criteria. The tool, muJava, is used in this research to generate method-level mutants.

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. 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. 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. The SUT implementation was implemented as an Akka Java actor since the Akka Testkit framework provides excellent tools to test actor based systems. Of the tools provided by the Akka Testkit, its methods for sending and receiving messages to and from Akka actors are by far the most useful for this study. Relying on open-source software, the Akka Testkit and ScalaTest make it extremely easy to test mutants based on the generated test cases. See section 4.3 for more details. The specification of the SUT can be found in appendix A.

4.2 Coverage Criteria

The coverage criteria for the test suite generated in this research is the set of all mutants that must be killed. Mutation testing is a kind of syntax-based testing, where the coverage criteria is the syntactic description of a software artifact. Recall that *Criterion 5.32* in Ammann and Offut [2008] states that each mutant $m \in M$ where M is the set of mutants for a given artifact will lead to a test requirement - each test requirement produced by the elements in M is a part of our coverage criteria. The test suite consists of a tuple (M, T), where M is the set of mutants and T is a list of traces[2]. 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.

[2]It would be more useful to have a *set* of traces, but since randomness does not guarantee uniqueness, we can accept using a list.

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

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 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.

Since abstract test cases ase just traces of sent messages, they are in a format representing the receiver and the message. Here is an example:

$$\begin{aligned}
&bus:INITIAL-att:ACK-att:CRAZY-bus:CREATECONNECTION-bus:DISCONNECT\\
&-att:ACK-att:FAIL-att:CRAZY-att:CRAZY-bus:UNSUBSCRIBECALLBACK\\
&-bus:UNSUBSCRIBE-att:FAIL-att:FAIL-att:CRAZY-att:CRAZY\\
&-bus:CONNECT-bus:SUBSCRIBE-att:ACK-att:ACK-att:CRAZY-att:CRAZY\\
&-bus:DISCONNECT-att:ACK-att:CRAZY-bus:SUBSCRIBECALLBACK\\
&-bus:DISCONNECT-att:FAIL-att:FAIL-att:CRAZY-att:CRAZY\\
&-bus:CREATECONNECTION-bus:DESTROYCONNECTION-att:FAIL-att:ACK\\
&-att:CRAZY-att:CRAZY-bus:PUBLISH-att:FAIL-att:CRAZY\\
&-bus:SUBSCRIBECALLBACK-att:FAIL-att:CRAZY-bus:UNSUBSCRIBECALLBACK\\
&-att:FAIL-att:CRAZY-bus:CONNECT-att:FAIL-att:CRAZY-bus:PUBLISH\\
&-bus:UNSUBSCRIBE-att:FAIL-att:FAIL-att:CRAZY-att:CRAZY\\
&-bus:UNSUBSCRIBE-bus:DESTROYCONNECTION-att:FAIL-att:FAIL\\
&-att:CRAZY-bus:CREATECONNECTION-att:IDLE-att:ACK\\
&-bus:DESTROYCONNECTION-att:IDLE-att:ACK-
\end{aligned}$$

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]. The executable test cases are quite simple in nature - messages are sent to the SUT and the test environment expects a `Success` or a `Failure` message in return. If the test environment receives an unexpected message, the test case is considered as failed[5].

The test cases are mapped from simple traces to a series of messages that should be sent and received correspondingly. The requirement is for the messages to be *sent* in the same order as the traces indicate, but they can be *received* out of order. The mapping can be done dynamically but for this study a `TestGenerator` was written as a Java class with a hard-coded map to translate elements inside a trace to a send/receive pair.

[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

[5]As discussed in later sections, the goal is to get test cases to fail (hopefully sooner rather than later).

4.4 Mutant Generation

Mutants were generated using the muJava mutation system for Java programs [Offutt et al., 2015]. 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.

Class mutants can be generated by muJava by using class-level mutation operators. This was not done since the simplified implementation consisted mostly of a single class (with inner classes as message types). muJava defines 12 method-level operators which were all applied to the SUT implementation. Arithmetic, conditional and logical operators can be either *replaced*, *deleted*, or *inserted*[6] in the code. Relational, shift and assignment operators can be replaced but insertion or deletion is not done. Ma and Offut [2005] provide further details for mutation operators. Mutants that do not compile are considered killed, but are not counted towards to total number of mutants.

4.5 Test Case Evaluation

The test cases in the generated test suite 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 test suite attempts to kill each mutant by running the executable test cases. If a mutant fails some test case, it is considered killed. The goal of a test suite is to kill all mutants, although determining which mutants are not being killed can be beneficial, especially for further development of the coverage criteria.

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 (generally due to lacking mutation operators); 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.

We encountered some issues with the chosen build tool, Scala-SBT. It is not well suited for testing on parallel distributed file systems as it requires file system locking, a feature that is not generally present on file systems residing on networked storage. We therefore replaced the tool with a simple fail-fast build script, which has the potential to increase performance. The performance increase depends on randomness[7].

[6] Only unary operators can be inserted.

[7] When abstract test cases are compiled into concrete test cases, it's possible to split them into batch sizes. This introduces slight overhead for each generated batch but if a mutant fails on the first batch, then the rest will be ignored.

5 Results

Although muJava attempted to generate 356 mutants, only 316 mutants of our implementation of the SUT could be compiled[8]. 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.

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[9]: `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. 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 also tested 250 traces[10] with all mutants being killed, while 100 traces[11] did not prove to be sufficient (4 survivors, the same as for the 50-trace run with the exception of `COR_21`). Due to the nature of randomness, we cannot guarantee that any number of traces will invariantly kill all mutants (provided that killing all mutants is at all possible) - at least not until some heuristics are applied to trace extraction.

We were able to find one 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 that in order to kill all mutants, it is important that the model and implementation be faithful to each other. Mutants generated from a correct implementation but an incorrect model are more difficult to kill and create gaps in the coverage, as do mutants generated from an incorrect implementation but a correct model.

6 Conclusion

The research shows that by modeling a SUT, it is possible to easily create many abstract test cases from the state space of the model. The research does not go into heuristics for selecting the best abstract test cases but uses randomness instead. We found that 100 abstract test cases extracted from the state space of the model did not reliably satisfy the coverage criteria, but larger amounts such as 250 and 1000 managed to satisfy the criteria.

The goal of using an automatically generated test-suite such as the one demonstrated in this study is to kill all mutants. If all mutants can be killed, then either the SUT conforms to the specifications applied to the model or more test requirements are necessary to detect bugs. A failing test suite will uncover a fault in either the model or implementation.

The results are indicative that certain mutants are less likely to be killed as the number of abstract test cases decreases when compared with other mutants. It might be beneficial to pinpoint these

[8]This is a limitation in muJava, every single mutant that muJava could generate was generated.
[9]data/0050_5_0.traces
[10]data/0250_0_0.traces
[11]data/0100_4_0.traces

mutants and use them as inexpensive coverage criteria for regression testing. Those mutants could also possibly reveal sensitive areas in the system.

7 Future Work

Despite the contibution to the field made by this study, there is still much research left to be explored. Most notably, the work should be repeated with a complete implementation of a system rather than with an implementation that is deliberately nearly identical to the model itself. Increased complexity in a system's implementation can introduce hidden side effects that could possibly affect the results of the test cases.

Furthermore, due to the unexpected issues with Scala-SBT on parallel distributed file systems (as detailed at the end of section 4.5), the build system (which acts also as the test runner) should be replaced by a more suitable alternative. This has been done in part, but remains to be tested on distributed systems[12].

A highly desireable improvement on the current work would be to add some sort of heuristics to the selection of traces from the state space of the system. Currently the traces are chosen at random with no guarantees of any actual coverage - with actual heuristics in place it will be possible to guarantee some sort of coverage rather than depending on randomness.

There seems to be a bias towards specific mutants not being killed even when completely different random traces are used as abstract test cases. It might be useful to explore where this bias comes from and what other information could be hiding in those mutants.

The most computationally intensive (or at least time-consuming on the hardware this study was performed on) was compiling and running each test case for each mutant. Killing the mutants can be optimized by generating better test case heuristics, even if the test cases are simply re-ordered. The mutants themselves can also be reordered to achieve increased performance since each mutant should be able to uncover a bug (and a bug can often lead to other bugs) – by examining the first failing mutant immediately it should be achievable to patch the bug before the test cases have fully terminated. This of course would work best if any mutants with more bias are tested first. Finally, it may also be helpful to discard any test cases that provably can not kill mutants (or if they are extremely unlikely to do so, save them for last – even save them for a final validation).

[12]The build system was replaced with a bash script in an effort to allow tests to be run on a high-performance computing cluster, but the cluster was ongoing maintenance and the required tools were out-of-date, if at all present.

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.

Yu-Seung Ma and Jeff Offut. Description of method-level mutation operators for java. *muJava*, 2005.

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

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

Marjan Sirjani and Mohammad Mahdi Jaghoori. Ten years of analyzing actors: Rebeca experience. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer, 2011. ISBN 978-3-642-24932-7. URL http://dx.doi.org/10.1007/978-3-642-24933-4_3.

Samira Tasharofi. *Efficient testing of actor programs with non-deterministic behaviors*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

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

Appendices

A Bus Specification

The specification of the SUT specifies the following methods:

- createConnection:
  precondition: no other connection objects can exist
  postcondition: a new connection exists
- destroyConnection
  precondition: a connection object must exist
  postcondition: the connection object is destroyed – any subscriptions and messages are deleted
- connect
  precondition: a connection object must exist
  postcondition: a connection has been established
- disconnect
  precondition: a connection must be established
  postcondition: the connection has been broken – (subscriptions and messages are not deleted)
- subscribe
  precondition: a connection must be established and no other subscriptions can exist
  postcondition: a subscription has been created
- subscribeCallback
  precondition: a connection must be established and the number of callback subscriptinos must not exceed 10
  postcondition: a callback subscription has been created
- unsubscribe
  precondition: a subscription must exist and a connection must be established
  postcondition: all subscriptions have been removed
- unsubscribeCallback
  precondition: a callback subscription must exist and a connection must be established
  postcondition: a single callback subscription has been removed
- publish
  precondition: a connection must be established
  postcondition: a message has been published (into a message queue)
- getMessage
  precondition: a connection must be established and a message must exist in the queue
  postcondition: a message has been removed from the queue and sent to the caller

B $\mathcal{R}$ebeca Model - Full Code

```
reactiveclass MessageBus(15) {
  knownrebecs {
    Attacker att;
  }

  statevars {
    // ATT vars
    boolean connectionExists;
    boolean connected;
    boolean subscribed;
    int subs;
    // callbacks and messages
    int messages;
    int numCallbackSubscriptions;
    // other
    int destroyedMsgs;
  }

  msgsrv initial() {
    connectionExists = false;
    connected = false;
    subscribed = false;
    subs = 0;
    messages = 0;
    numCallbackSubscriptions = 0;
    destroyedMsgs = 0;
  }

  msgsrv createConnection() {
    if (sender == att) {
      if (!connectionExists)
      {
        connectionExists = true;
        att.ack(1);
      }
      else
      {
        att.fail(); // error
      }
    }
  }

  msgsrv destroyConnection() {
    if (sender == att) {
      if (connectionExists)
      {
        connected = false; // disconnect
        numCallbackSubscriptions = 0; // delete subscriptions
        subscribed = false; // delete subscriptions
        destroyedMsgs += messages;
        messages = 0; // delete messages
        connectionExists = false; // destroy connection object
```

```
      att.ack(2);
    }
    else
    {
      att.fail(); // error
    }
  }
}

msgsrv connect() {
  if (sender == att) {
    if (connectionExists && !connected)
    {
      connected = true;
      att.ack(3);
    }
    else
    {
      att.fail(); // error
    }
  }
}

msgsrv disconnect() {
  if (sender == att) {
    if (connected)
    {
      connected = false;
      att.ack(4);
    }
    else
    {
      att.fail(); // error
    }
  }
}

msgsrv subscribe() {
  if (sender == att) {
    if (subs < 2)
    {
      subs = subs + 1;
    }
    if (connected && !subscribed)
    {
      subscribed = true;
      att.ack(5);
    }
    else
    {
      att.fail(); // error
    }
```

```
  }
}

msgsrv subscribeCallback() {
  if (sender == att) {
    if (connected && numCallbackSubscriptions < 10)
    {
      numCallbackSubscriptions += 1;
      att.ack(7);
    }
    else
    {
      att.fail(); // error
    }
  }
}

msgsrv unsubscribe() {
  if (sender == att) {
    if (subs >= 0)
    {
      subs = subs - 1;
    }
    if (connected && subscribed)
    {
      subscribed = false;
      att.ack(6);
    } else if (connected && numCallbackSubscriptions > 0) {
      if (subs < 2)
      {
        subs = subs + 1;
      }
      att.ack(6);
    }
    else
    {
      att.fail(); // error
    }

    if (connected) {
      // delete all callbacks unconditionally
      numCallbackSubscriptions = 0;   // what if we are not connected?
    }
  }
}

msgsrv unsubscribeCallback() {
  if (sender == att) {
    if (connected && numCallbackSubscriptions > 0)
    {
      numCallbackSubscriptions -= 1;
      att.ack(8);
```

```
      }
      else
      {
        att.fail(); // error
      }
    }
  }

  msgsrv publish() {
    if (sender == att) {
      if (connected)
      {
        if (subscribed || numCallbackSubscriptions > 0) {
          messages += 1; // add the message to the queue

          int i; // invoke callback function for each subscription
          for (i = 0; i < numCallbackSubscriptions; i = i + 1)
          {
            att.callback();
          }
        }
        att.ack(9);
      }
      else
      {
        att.fail(); // error
      }
    }
  }

  msgsrv getMessage() {
    if (sender == att) {
      if (connected && messages > 0)
      {
        att.receive();
        messages -= 1;
      }
      else
      {
        att.fail(); // error
      }
    }
  }
}

// the Attacker is basically just a "dumb" app that doesn't follow the
   API properly
// its purpose is to try to find counterexamples to the properties
reactiveclass Attacker(32) {
  knownrebecs {
    MessageBus bus;
  }
```

```
statevars {
  int connected;  // 0 = disconnected; 1 = connected; 2 = connecting
  int rMsgs;     // 'received'/processed messages
  int rCallbacks; // # of invoked callbacks
  int cSubs;     // # of callback subscriptions (owned)
  int published;  // # of published messages
  int acks;
  boolean stop;
  boolean hasUnsubscribed;
}

msgsrv initial() {
  connected = 0;
  rMsgs = 0;
  rCallbacks = 0;
  cSubs = 0;
  published = 0;
  acks = 0;
  stop = false;
  hasUnsubscribed = false;
  self.ack(0);
  self.crazy(-1); // continue after failures
}

msgsrv receive() {
  rMsgs += 1;
}

msgsrv callback() {
  rCallbacks += 1;
}

// this message server receives success codes and simulates a
    ridiculous test-run
// this server only receives ACCEPT acknowledgements, the ID specifies
    what action succeeded
msgsrv ack(int id) {
  acks = acks + 1;
  if (acks > 20) { // only simulate 20 "stupid" steps
    stop = true;
  }

  // handle changes
  if (id == 1) {
    connected = 2;
  } else if (id == 2 | id == 4) {
    connected = 0;
  } else if (id == 3) {
    connected = 1;
  }
```

```
    if (id == 0) {        // used to activate command chain
      connected = 2;
      bus.createConnection();
    } else if (id == 9) { // published
      published += 1;
      self.idle();
    } else if (!stop) {
      int old = id;
      self.crazy(old);
    } else if (stop && connected == 0 && (id == 2 || id == -1)) {
      self.idle();
    } else if (stop) {
      hasUnsubscribed = true;
      bus.destroyConnection();
    } else {
      self.idle();
    }
  }

  // try something else
  msgsrv fail() {
    acks = acks + 1;
    if (acks < 20) {
      self.crazy(-1);
    } else {
      self.idle();
    }
  }

  // performs a random action
  msgsrv crazy(int old) {
    int id;

    id = ?(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

    if (id == 0) {
      bus.createConnection();
    } else if (id == 1) { // connection object created
      bus.connect();
    } else if (id == 2) {
      if (old == 2) {
        stop = true;
      }
    } else if (id == 3) {
      bus.subscribe();
    } else if (id == 4) {
      hasUnsubscribed = true; // destroying the connection loses
          subscriptions
      bus.destroyConnection();
    } else if (id == 5) {
      bus.publish();
    } else if (id == 6) {
```

```
      bus.disconnect();
    } else if (id == 7) {
      bus.unsubscribeCallback();
    } else if (id == 8) {
      hasUnsubscribed = true;
      bus.unsubscribe();
    } else if (id == 9) {
      bus.subscribeCallback();
    }
  }

  msgsrv idle() {
    self.idle();
  }
}

main {
  MessageBus bus(att):();
  Attacker att(bus):();
}
```

The $\mathcal{R}$ebeca model includes two `rebec`s: the message bus itself and an application (`Attacker`) that sends messages to the message bus at random. This ensures that as long as the `MessageBus rebec` is correct, the generated state space will represent the state space of a correct implementation of the SUT.

C Implementation - Full Code

```
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

import java.util.Queue;
import java.util.LinkedList;

public class JBus extends UntypedActor {

  public interface BusMessage {
    // intentionally left blank
  }

  public final static class CreateConnection implements BusMessage {
    private final int id;

    public CreateConnection(int id) {
      this.id = id;
    }
  }

  public final static class DestroyConnection implements BusMessage {
    private final int id;

    public DestroyConnection(int id) {
      this.id = id;
    }
  }

  public final static class Connect implements BusMessage {
    private final int id;

    public Connect(int id) {
      this.id = id;
    }
  }

  public final static class Disconnect implements BusMessage {
    private final int id;

    public Disconnect(int id) {
      this.id = id;
    }
  }

  public final static class Subscribe implements BusMessage {
    private final int id;

    public Subscribe(int id) {
      this.id = id;
```

```java
  }
}

public final static class SubscribeCallback implements BusMessage {
  private final int id;

  public SubscribeCallback(int id) {
    this.id = id;
  }
}

public final static class Unsubscribe implements BusMessage {
  private final int id;

  public Unsubscribe(int id) {
    this.id = id;
  }
}

public final static class UnsubscribeCallback implements BusMessage {
  private final int id;

  public UnsubscribeCallback(int id) {
    this.id = id;
  }
}

public final static class Publish implements BusMessage {
  private final int id;
  String message;

  public Publish(int id, String message) {
    this.id = id;
    this.message = message;
  }
}

public final static class GetMessage implements BusMessage {
  private final int id;

  public GetMessage(int id) {
    this.id = id;
  }
}

public final static class Success implements BusMessage {
  @Override
  public boolean equals(Object o) {
    if (o instanceof Success) {
      return true;
    }
    return false;
```

```
    }
  }

  public final static class Error implements BusMessage {
    @Override
    public boolean equals(Object o) {
      if (o instanceof Error) {
        return true;
      }
      return false;
    }
  }

  private boolean connectionObjectExists = false;
  private boolean connected = false;
  private boolean subscribed = false;
  private int     callbacks = 0;
  private int     callbackInvocations = 0; // internal counter
  private Queue<String> messages = new LinkedList<>();

  public void onReceive(Object message) throws Exception {
    if (message instanceof CreateConnection) {
      if (!connectionObjectExists) {
        connectionObjectExists = true;
        getSender().tell(new Success(), getSelf());
      } else {
        getSender().tell(new Error(), getSelf());
      }
    } else if (message instanceof DestroyConnection) {
      if (connectionObjectExists) {
        connectionObjectExists = false;
        connected = false;
        subscribed = false;
        callbacks = 0;
        messages.clear();
        getSender().tell(new Success(), getSelf());
      } else {
        getSender().tell(new Error(), getSelf());
      }
    } else if (message instanceof Connect) {
      if (connectionObjectExists && !connected) {
        connected = true;
        getSender().tell(new Success(), getSelf());
      } else {
        getSender().tell(new Error(), getSelf());
      }
    } else if (message instanceof Disconnect
        && connected) {
      connected = false;
      getSender().tell(new Success(), getSelf());
    } else if (message instanceof Subscribe
        && connected && !subscribed) {
```

```
      subscribed = true;
      getSender().tell(new Success(), getSelf());
    } else if (message instanceof SubscribeCallback
        && connected && callbacks < 10) {
      callbacks += 1;
      getSender().tell(new Success(), getSelf());
    } else if (message instanceof Unsubscribe
        && connected && (subscribed || callbacks > 0)) {
      subscribed = false;
      callbacks = 0; // possibly differs from model
      getSender().tell(new Success(), getSelf());
    } else if (message instanceof UnsubscribeCallback
        && connected && (callbacks > 0)) {
      callbacks -=1 ;
      getSender().tell(new Success(), getSelf());
    } else if (message instanceof Publish
        && connected) {
      /// BEWARE POSSIBLE BUG!
      if (subscribed || callbacks > 0) {
        messages.add(((Publish) message).message);
      }
      getSender().tell(new Success(), getSelf());
    } else if (message instanceof GetMessage
        && connected && messages.size() > 0) {
      String msg = messages.poll();
      getSender().tell(new Success(), getSelf());
    } else {
//        unhandled(message);
      getSender().tell(new Error(), getSelf());
    }
  }
}
```

The implementation is analogous to the $\mathcal{R}$ebeca model. Details are omitted but the the implementation and model are faithful to each other.