Model-Based Shrinking for State Based Testing

Pieter Koopman, Peter Achten, and Rinus Plasmeijer

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, the Netherlands
{pieter,p.achten,rinus}@cs.ru.nl

Draft. Feedback is most welcome

Abstract. Issues found in model-based testing of state-based systems are traces produced by the system under test, sut, that are not allowed by the model used as specification. It is usually easier to determine the error behind the reported issue when there is a short trace revealing the issue. The model-based test system treats the sut as a black box. Hence the test system cannot use internal information of the sut to find short traces. This paper show how the model can be used to systematically search for short traces producing an issue based on a long trace revealing an issue.

1 Introduction

In model-based testing, MBT, of state based systems there is a model that specifies allowed transitions between states. Each transition in an extended state machine is labeled with an input and the corresponding output. The conformance relation restricts the allowed outputs by the system under test, sut, to output covered by the model for the inputs specified for each reachable state [3,4].

In a state based system the reaction of the sut and the model on some input depends on the current state and hence on the history. The list of previous transitions, the trace, determines the current state. When the observed output for some transition of the sut is not covered by the model we have determined nonconformance. In the test jargon this is called an *issue*. In an ideal test world such an issue indicates an error in the sut. In the real world issues can also indicate errors in the model or interfacing problems between the sut and the test system.

In simple cases the error indicated by a discovered issue is obvious. This happens for instance when it is clear that the output that is generated by the sut is not allowed in the reached state. Often it is less obvious what caused the illegal transition. In such situations we have to take the trace in consideration since it determines how we arrived in the current state. It is obvious that analysing such issues is in general much easier when we have a short trace indicating the issue.

In the unguided test situation the test system has no idea where to search potential nonconformance and takes transitions in a pseudo random order. This can result in fairly long traces of several thousands of transitions. In this paper

we discuss strategies for finding smaller traces indicating an issue based on such a long trace.

The shrinking algorithm used for traces showing an issue acknowledges that the state of the sut can be quite different from the state of the model. Nevertheless, similar states for the model can correspond to similar states in the sut. We generate candidate traces to be tested based on the trace that is known to reveal an issue. Effective ways to generate a test suite are by eliminating individual transitions from the trace and by eliminating all transitions corresponding to a cycle in the model.

2 Conformance

A trace σ is a sequence of inputs and associated outputs from a given state. The empty trace, ϵ, connects a state to itself: $s \stackrel{\epsilon}{\Rightarrow} s$. We write $s \stackrel{\sigma}{\Rightarrow}$ to indicate $\exists u \,.\, s \stackrel{\sigma}{\Rightarrow} u$ and $s \stackrel{i/o}{\longrightarrow} \equiv \exists u \,.\, s \stackrel{i/o}{\longrightarrow} u$.

The inputs allowed in a state s are $\mathsf{init}(s) \equiv \{i | \exists o.s \stackrel{i/o}{\longrightarrow}\}$. The states after applying trace σ in state s are given by $s \,\mathsf{after}\, \sigma \equiv \{t | s \stackrel{\sigma}{\Rightarrow} t\}$. Operations like init, and after are overloaded for sets of states. Note that there are infinitely many traces and an infinitely long trace if the state machine contains a loop, that is $\exists\, s, \sigma \,.\, s \stackrel{\sigma}{\Rightarrow} s$, even if the machine is a finite state machine, or fsm.

For *conformance* between a model and a sut the observed output of the sut should be allowed by the model for each input i in the init after every trace σ. Formally, conformance of the sut to the specification spec is defined as:

$$\mathsf{sut}\ \mathit{conf}\ \mathsf{spec} \equiv \forall \sigma \in \mathsf{traces}_{\mathsf{spec}}(s_0).\forall i \in \mathsf{init}(s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma)\forall o \in [O].$$
$$(t_0\ \mathsf{after}_{\mathsf{sut}}\ \sigma) \stackrel{i/o}{\longrightarrow} \Rightarrow\ (s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma) \stackrel{i/o}{\longrightarrow}$$

Here s_0 is the initial state of the specification and t_0 is the initial state of the sut. Note that this conformance relation compares inputs and outputs. The actual states of the sut are never used, hence the sut is treated as a black box. Comparing inputs and outputs of the model and the sut implies that they have to use identical types of inputs and outputs. The states however, can be of completely different types.

Testing state based machines is built on this conformance relation. Model-based testing checks the conformance relation for a finite number of finite traces. Since there can be infinitely many or infinitely long traces, testing conformance is in general an approximation of the relation *conf*.

Instead of generating traces of the specification and verify whether they are accepted by the sut the test algorithm of G∀st, `testConf`, maintains the after states of the current trace. It determines for n transitions on a single trace *on-the-fly* [5,6] whether the observed behaviour of the sut is conform to the model. If there are no after states testing has determined nonconformance. If the number of steps to do is zero, testing this trace is terminated without any conformance problems. Otherwise we choose an arbitrary input that is accepted by the specification in

one of the current states, apply it to the sut and observe the corresponding output. The new set of states for the specification is the set of states that is obtained after the transition $\xrightarrow{i/o}$. The algorithm continues testing with one step less and the new set of states.

```
:: Verdict = Pass | Fail | Truncate

testConf :: Int [S] T → Verdict
testConf n [] t = Fail
testConf 0 ss t = Pass
testConf n ss t
    | isEmpty inputs = Pass
    | otherwise      = testConf (n–1) (after ss i o) t2
where inputs = init ss
      i      = elemFrom inputs
      (o,t2) = sut t i
```

In order to test conformance we evaluate `testConf` N `[s0] t0` for M traces. When there is a *test goal* (some specific behaviour) we use this to select the input i from the current init, otherwise the test system selects a pseudo random element. The real implementation of this algorithm in G∀st is more involved since it is parameterized by the sut and spec, has to collect the trace, guide input selection, allow pseudo random input selection, etc. When a trace is found that proves nonconformance, the trace is shown together with the intermediate model states on the trace. The sut is treated as a black box, hence its states cannot be shown.

There is another mode of conformance testing relevant for this paper. Here we have a given list of inputs (inserted of generating it on-the-fly) and check whether there is conformance of sut and the specfor this sequence of inputs. If the input at some point is not part of the initof the specat some point testing this sequence is truncated; there is no issue found, but we can neither continue the conformance check.

```
testConf2 :: [I] [S] T → Verdict
testConf2 n [] t = Fail
testConf2 [] ss t = Pass
testConf2 [i:r] ss t
    | isMember i inputs = testConf2 r (after ss i o) t2
    | otherwise         = Truncate
where inputs = init ss
      (o,t2) = sut t i
```

In order to use model-based testing for abstract data types, adt, with a state machine as specification, the sut must behave as a state machine. For the sut we construct a very simple machine that stores the actual adt as its state. For the spec we use a state machine with a state that contains enough information to check the transitions. Such a state is typically a naïve implementation of the interface offered by the adt, or an abstraction of it.

3 Binary Search for Minimal Traces

Without a systematic way to produce smaller traces from a trace revealing an issue we used a binary search technique to find small traces revealing an issue. Initially we test with a large upper bound on the length of the traces. When there appears to be a trace of length n revealing an issue we try to find an issue with length $n/2$ as upper bound. If this succeeds we continue with $n/4$ as upper bound. Otherwise we continue with $3n/4$ as upper bound. By repeating until the difference in trace length is sufficiently small we can find small traces revealing an issue.

For the new test with a smaller upper bound we use different traces, If there is an issue with a prefix of the current trace it would have been discovered before. By default we generate traces on the fly by a pseudo random choice of one of the inputs in the initof the current state. By just using another seed for the pseudo random choices we typically find enough other traces.

It is not guaranteed that the shorter trace finds *the same* issue. Hence it makes sense to store the original trace and check it when the issue with a shorter trace is resolved.

Although this methods works, it is somewhat unsatisfactory as it uses only the length information of the issue found to guide the search for shorter traces revealing an issue.

4 The Desire for Small Traces

Theoretically all traces chowing nonconformance are equally good in falsifying conformance. In practice however, small traces are preferable. The next thing one does after spotting nonconformance is investigating the source of this issue. We want to know whether we can blame the specification or the system under test. IN general specifications are similar artefacts as the system under test. This implies that we cannot assume that the specification is always correct.

In rare situations it is obvious that the observed combination of input and output is incorrect and we only have to consider the last transition of the sequence showing the issue. In most situation we need to observe how the end up in a state where this issue was observed. A short trace is here obviously easier than a long one.

It seems tempting to test all input sequences in a breath first order to find minimal traces showing nonconformance. In practise this is ably feasible in toy examples. Usually the states and inputs can be parameterised with data types and hence there are so many short sequences to consider that we can only test very short sequences using a breath first strategy. To cope with this problem we usually test with a limited number, by default 100, of rather long, by default 1000, inputs. When we find an issue during these tests, we can spent some additional effort to find a short trace showing the issue.

Until we implemented shrinking we used a kind of binary search for small traces. When the default test show that there is an issue with a trace of length

n_i it is worthwhile to search for a small trace. Hence we try to find an issue with upper bound $n_i/2$ for the length of the trace. Since we know that there is an issue we typically increase the number of traces tried. When we find an issue with this length we try to find an issue with length $n_i/4$, otherwise we try to find it with upper bound $3n_i/4$. Although this is not guaranteed to find minimal traces, this heuristic appears find reasonable short traces. This binary search approach uses only the length of the trace showing that there is an issue. It is tempting to use more information of this trace in order to find a short trace.

5 Shrinking

The technique to find smaller counterexamples based on an already found counterexample in model-based testing is called shrinking. It is well know from QuickCheck [1,2]. It appears to be rather effective in model-based testing on logical properties. Shrinking systematically generates candidate test cases based on a test value that is known to falsify the property at hand. When the test value is a list we can for instance take the first or second half of the list as candidate test values as well as the init or tail of this list. When one of these test cases appears to falsify the logical property, shrinking can proceed with this new counterexample until we have found some minimal counterexample.

In principle we can use this algorithm also for traces. Drawbacks of this algorithm are that it does not use any information of the model. Usually testing a trace with the sut is much slower than computing with traces inside the test system. Testing a trace with the sut requires that the current input is transferred to the sut and the corresponding answer of the sut is transferred back to the test system where it must be checked against the model. Depending on the sut a single step in the sut can take quite some time, especially when real world objects have to be manipulated.

In real world applications the first traces that are found that reveal an issue have often a length of some hundreds or thousands of transitions. Since there are 2^n candidates of smaller traces for a trace of length n, exhaustive testing of these smaller traces is unfeasible in general. We have to rely on heuristics.

5.1 Eliminating Single Transitions

When the states in the model before and after a transition are identical, the transition is most likely a no-operation or query on the state. There is no guarantee whatsoever that the states in the sut for such a transition are identical when the states in the model are identical, but the trace where such a transition is removed is a good candidate for testing.

Note that we shrink the trace here based on the states in the model corresponding to the trace rather than shrinking on the list of transitions as data structure which is the default shrinking strategy.

As expected this shrinking algorithm is very effective in removing irrelevant single transitions. For a trace of length n we have to compare $O(n)$ model states.

We have to test $O(n)$ new traces. Since many transitions change the state the number transitions that can potentially be remove is usually much smaller than n. We generate a new test case for each of these transitions. When the issue still occurs for the trace without the considered transition, it is permanently removed from the test cases. Otherwise the transition appears to be essential in the trace for the sut and remain in the traces investigated.

5.2 Eliminating Cycles

In a similar way we can remove sequences of transitions yielding cycles in the model. The previous shrinking approach is basically a limit situation of eliminating cycles with cycle-length one. We treat this separately since detecting cycles is more expensive than checking the states involved in a single transition.

Again, there is no guarantee that cycles in the model correspond to cycles in the sut that can be removed, but it makes good test candidates.

There are some subtleties here caused by traces that use a cycle more than once, i.e. the same sequences of states in the model occurs two or more times in the trace. If we are not careful in the test case generation algorithm this generates many redundant test cases. Apart from removing the cycle twice, we can generate a trace where the first or second traversal of the cycle is removed. To make things worse we can also find a cycle between between the second state on this cycle, and the third state and so on. Although the traces are produced by eliminating different transitions, the resulting traces are identical. Our shrinking algorithm prevents the generation of such test cases.

It is worthwhile to note that this does not prevent models with cycles, nor traces with cycles. Cycles in the model are usually necessary to model systems. It is also worthwhile to use these cycles in the test cases. This algorithm just checks if a an issue occurs when a cycle is removed from a trace indicates an issue.

6 Implementing Shrinking

In order to have a somewhat general implementation of shrinking we define a binary tree for shrinking:

```
:: Shrinks i = Shrinks [i] (Shrinks i) (Shrinks i) | NoShrinks
```

A general function tests the sequence of inputs, `[i]`, in the node `Shrinks`. If this sequence of inputs shows an issue shrinking continues with the first subtree, otherwise it continues with the second subtree. In this we we achieve a nice separation between generating candidates and using test results on one hand, and all the bookkeeping needed to execute the associated tests on the other hand.

6.1 Element Elimination

Using this machinery the greedy elimination of individual elements is achieved by generating the appropriate tree. The index n scans all inputs in the given list of inputs that is know to show nonconformance. When inputs2, with element n removed, also yields nonconformance we remove it from the list of inputs forever. Otherwise, we keep it in the list and continue with the next element.

```
elemElimination :: [i] → Shrinks i
elemElimination inputs = elim 0 (length inputs) inputs
where
    elim n len inputs
    | n < len
        ♯ inputs2 = removeAt n inputs
        = Shrinks inputs2 (elim n (len−1) inputs2) (elim (n+1) len inputs)
        = NoShrinks
```

This has clearly a left to right bias in trying to remove elements. As a consequence we have observed that it is sometimes worthwhile to apply this algorithm a few times in order to obtain the smallest input sequences.

6.2 Cycle Elimination

Eliminating cycles is more attractive than generating just individual transitions. Most cycles are longer than a single transition and hence removing a cycle is more effective than removing a single transition. Moreover, when the cycles are essential in the behaviour of the machines, we cannot remove the inputs on the cycle one by one. As soon as one of the inputs is removed, testing it truncated for that input sequence and we will not continue with this input sequence in the algorithm above.

Detecting cycles in the input sequence is hard. Only when a cycle is traversed two, or more, times we can detect this in the input sequence by encountering the same sequence of inputs two, or more, times. It is worthwhile to remove individual transitions before we are looking for cycles in this way. A single irrelevant transition can destroy the detection of a cycle bases on inputs.

Since the model-based cycle elimination discussed in the next section is more powerful and effective, we do not extend the treatment of this optimisation. Experiments have show that these kind of cycles do occur in practice. Sometimes such a cycle is essential to show nonconformance, but often it can be removed.

6.3 Model-Based Cycle Elimination

A cycle is a trace starting and ending in the same state. Since the system under test is a black box we cannot observe its state and hence we cannot detect cycles there. For the model however, we know everything. Apart from the inputs used in the trace we can also record the states. As soon as we discover two times the same model state in the trace we have found an cycle and we can try if it can be removed. This algoritm generates again a Shirnks tree.

```
cycleElimination :: [i] [s] → Shrinks i | gEq{|*|}, gLess{|*|} s
cycleElimination [] _ = NoShrinks
cycleElimination [i] _ = NoShrinks
cycleElimination inputs states = elim (findCycles states) inputs
where
    elim [c=:(f,t):cycles] inputs
        # inputs2 = cut f t inputs
        = Shrinks inputs2
            (elim (updateCycles f t cycles) inputs2)
            (elim cycles inputs)
    elim [] inputs = elemElimination inputs
```

The last line shows that we try to remove individual transitions after we have removed the cycles. Since we are looking at states here rather than at repetitions of inputs, there is no reason to remove individual transitions before we are removing cycles.

A cycle is given by the first and last input on that cycle. In order to detect cycles we couple states and index numbers in the list of transitions in `pairs`. Next we sort these pairs on the states in these pairs. When consecutive pairs have the same state we have found a cycle. In order to obtain small traces as soon as possible we sort the cycles from large to small.

```
findCycles :: [s] → [(Int,Int)] | gEq{|*|}, gLess{|*|} s
findCycles [] = []
findCycles [s] = []
findCycles states
    # pairs     = sortBy (λ(i,s) (j,t).s−←t)
                    [(i,s) \\ i ← [0..] & s ← init states]
      groups    = groupby (snd (hd pairs)) [] pairs
    = sortBy (λ(a,b) (c,d).b−a > d−c) (mkCycles groups)
```

mkCycles :: [[a]] -¿ [(a,a)] mkCycles [] = [] mkCycles [[]:r] = mkCycles r mkCycles [[i:r]:next] = reverse [(j,i)
j ¡- r] ++ mkCycles [r:next]

When a cycle is removed from the input, the indices of all other cycles have to be adapted by `updateCycles`.

```
updateCycles :: Int Int [(Int,Int)] → [(Int,Int)]
updateCycles f t [(x,y):r]
    | y<f   // new cycle before current cycle
        = [(x,y):updateCycles f t r]
    | x>t   // new cycle after cuurent cycle
        = [(x−t+f,y−t+f):updateCycles f t r]
    | otherwise // cycles overlap: remove new cycle
        = updateCycles f t r
updateCycles f t [] = []
```

7 Examples

We have a tested our shrinking algorithm in a number of examples. In all of these examples shrinking was able to produce traces revealing the issue that were an order of magnitude shorter than the original trace found.

As first example we use 10 different and erroneous implementation of a simple vending machine for coffee. The model for these implementations is defined as:

```
:: Product  = Coffee | Espresso | Double | French | Wiener
:: Input    = Coin1 | Coin2 | Choice Product | Reset | Info | Go
:: Output   = Cup Product | Change Int | Text String
:: State    = {product :: Maybe Product, balance :: Int}
state0 = {product = Nothing, balance = 0}

spec :: State Input → [Trans Output State]
spec s Coin1        = [Pt [] {s & balance = s.balance+1}]
spec s Coin2        = [Pt [] {s & balance = s.balance+2}]
spec s (Choice p)   = [Pt [] {s & product = Just p}]
spec s Reset        = [Pt (if (s.balance > 0) [Change s.balance] []) state0]
spec s Info         = [Ft accept] where accept [Text _] = [s]; accept _ = []
spec s Go
   = case s.product of
       Just p
           | s.balance ≥ value p
               = [Pt [Cup p] {state0 & balance = s.balance − value p}]
       _   = [Pt [] s]
```

The state of this machine records the product chosen (if any), and the amount of money incertted. The inputs of this machine are coins, with value one or two, a choice for one of the products, a reset, or a request for information. When the Go button is pressed the machine will produce the required product is the balance is sufficient.

The test results for a typical implementation with different seeds are given in Table 1. In this tables the numbers 1–10 indicate test runs with a different seed. The column labelled "avg" is the average of these columns. The row "issue" is the length of the trace found that shows the issue. The next row gives the length of this trace after element elimination. The row "trans" contains the number of transitions needed to find this small trace. The row labelled "cycles" contains the length of the trace after cycle elimination. In the row "both" the length of the trace after cycle elimination and subsequent element elimination is given. The final row contains the number of transitions needed in this combined approach.

Table 2 contains the averaged results for 10 different erroneous implementations of this vending machine. For each of these vending machines we record the average value over 10 different seeds for pseudo random input generation.

From the results in these tables we can conclude that we can often find significant smaller traces by using another seed for the pseudo random generation. Shrinking can in almost all situation reduce the size of such an issue considerably. The final traces found are on average about a factor 50 smaller that the initial

	1	2	3	4	5	6	7	8	9	10	avg
issue	730	148	230	726	37	293	304	398	145	665	368
elems	7	5	6	5	5	6	6	5	7	6	6
trans	266259	10992	26802	263272	836	45881	46420	79062	10740	222087	97235
cycles	13	12	23	11	14	10	16	7	20	15	14
both	7	5	7	5	5	6	7	5	7	6	6
trans	868	243	656	808	188	3374	616	440	409	1911	951

Table 1. Length of traces and number of transitions needed to find these traces for the vending machine.

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	avg
issue	368	63	19	469	126	47	149	91	34	368	173
elems	6	6	2	5	5	4	4	2	2	6	4
trans	97235	3524	399	153464	26341	2012	11906	5948	968	73541	37534
cycles	14	15	10	11	16	10	10	8	5	14	11
both	6	6	2	5	5	4	4	2	2	6	4
trans	951	321	98	1206	476	142	222	144	61	956	458

Table 2. Similar average results as in Table 1, but for 10 different implementations

traces found. In this example element elimination and cycle elimination followed by element elimination yield almost the same traces. The number of transitions needed to find these minimal traces is almost two orders of magnitude smaller when we use cycle elimination. This effect is stronger when the initial issue found is longer.

In this situation the issues found by element elimination and cycle elimination are equal. This is caused by the fact that the specification is input enabled; any input can be applied in each state. As a consequence none of the cycles is essential and we can remove them element by element. When there are real cycles in the system this does not work. Table 3 is very similar to Table 1, here we used a similar system that does contain real cycles.

	1	2	3	4	5	6	7	8	9	10	avg
issue	70	48	37	103	37	48	15	48	92	15	51
elems	70	48	37	103	37	48	15	48	92	15	51
trans	2753	1676	1173	5669	1173	1440	259	1374	4907	259	2068
cycles	15	26	26	15	26	15	15	15	26	15	19
both	15	26	26	15	26	15	15	15	26	15	19
trans	248	488	462	281	462	252	259	186	601	139	338

Table 3. Length of traces and number of transitions needed to find traces for a system with cycles.

From this table we can see that element elimination cannot do anything useful when the system requires cycles. Cycle elimination does work as expected with these cycles and removes superfluous cycles. Hence, we conclude that our model-based removal of cycles is a valuable contribution for shrinking traces for model-based testing with state based systems.

In principle those minimal traces can be found by the binary search approach used previously. Since this is only guided by the length of the traces, this approach requires even more transitions and is not guaranteed to find small traces. The shrinking approach described here is superior to the simple binary search method.

8 Conclusion

To simplify finding errors it is worthwhile to shrink the trace producing an issue in model-based testing based on state machines. The model of the machine can be used to determine suitable test cases. Identical, or equivalent, states in the model indicate potential equivalent states in the `sut`. By eliminating the transitions between both visits to such a state we generate shrunk test cases. These significant smaller traces make it much easier to analyse the source of nonconformance between the model and the system under test.

References

1. T. Arts, L. M. Castro, and J. Hughes. Testing Erlang data types with Quviq Quickcheck. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 1–8, New York, NY, USA, 2008. ACM.
2. J. Hughes. Software testing with quickcheck. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer, 2010.
3. P. Koopman and R. Plasmeijer. Testing reactive systems with Gast. In S. Gilmore, editor, *Proceedings of the 4th Symposium on Trends in Functional Programming, TFP '03*, pages 111–129. Intellect Books, 2004. ISBN 1-84150-122-0.
4. P. W. M. Koopman and R. Plasmeijer. Fully automatic testing with functions as specifications. In Z. Horváth, editor, *CEFP*, volume 4164 of *Lecture Notes in Computer Science*, pages 35–61. Springer, 2005.
5. R. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer, STTT*, 2(4):382–393, 2000.
6. A. van Weelden, M. Oostdijk, L. Frantzen, P. Koopman, and J. Tretmans. On-the-fly formal testing of a smart card applet. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Proceedings of the 20th International Information Security Conference, SEC '05*, pages 564–576, Makuhari Messe, Chiba, Japan, May 2005. Springer-Verlag. Also available as Technical Report NIII-R0428.