Resit Exam Testing Techniques
192170015
1 July 2015
8:45 — 11:45

To make this exam:

- You are allowed to have 1 A4 sheet with your notes and nothing else.
- **Make each exercise on a separate page.**
- Write your name on each separate page that you hand in.
- Hand in the exam as well.

We wish you a lot of success!

Note by Marcus: The solutions presented here are by no means complete. They do, however, give an indication of how a solution could look like.

Let me know in case any questions arise.

1 What is testing?

1. Describe the purpose of testing. *(1 points)*
2. In his guest lecture, Machiel van der Bijl presented how Axini uses model-based testing in practice. Describe their main reasons for using model-based testing. *(1 points)*

Answers:

1. We'll accept a multitude of answers here, given it's only 1 point.
   - Testing makes better software cheaper.
   - Testing supports debugging.
   - Testing shows that a system works.
   - Testing shows that a system is not working.
   - Testing reduces the risk of software not working.
   - Testing investigates the quality of software.
   - Testing is a mental discipline that results in low risk software.
   - ...

   cf. lecture 1 slide 10.
2. This used to be a guest lecture in testing techniques. We'll read it as "Describe the main advantages of MBT."
   - Enforcing formal-methods/formal-models improves unclear and poorly defined requirements.
   - Once model is created, tests are generated automatically.
   - Tests are evaluated automatically.
   - Reusability and the distribution of models.
   - Saves time in software development lifecycle.
   - Various model formalisms with multiple advantages.

   cf. assignment 4 exercise 1.

2 Blackbox Testing.

The C function *saved* calculates the total sum of money that results after saving for years a certain fixed amount per year with a fixed interest rate. More precisely, on Januari 1st of each year a certain amount `amount` is put in a bank account. Each year on December 31st, the bank sents out an account summary indicating the total amount of money in the account. The bank uses the function `saved` below to compute the total amount of money after `years` years of saving.

The three inputs must be greater than or equal to 0; the output is a real value (double in C).

```
double saved(int amount, double rate, int years)
{
  int j;
  double s;

  j = 1;
  s = amount * (1.0 + rate/100);

  do
  {
    s = (s + amount) * (1.0 + rate/100.0);
    j = j+1;
  }
  while (j<years);

  return s;
}
```

1. Give a formal specification for the function `saved` as pre- and postconditions, based on the informal description. *(2 points)*
2. Use the equivalence partitioning technique to divide the input suitable equivalence classes. *(2 points)*
3. Give a test set that covers all equivalence classes. *(1 points)*
4. Extend the test set following the principle of boundary value analysis. *(2 points)*
5. Use the principle of error guessing to extend your test suite. *(2 points)*
6. Give a test case that fails on the implementation above. *(1 points)*

Answers:

1. Since we did not properly deal in the course with pre- and postconditions, we'll use something similar like

   - Pre:
     - # inputs is 3.
     - input 1 integer ≥ 0.
     - input 2 double ≥ 0.
     - input 3 integer ≥ 0.
   - Post:
     - Output is double ≥ 0.
     - Output is $amount \cdot \sum_{j=1}^{year} (1 + \frac{rate}{100})^j$

2. We'll mostly reuse the results of 1. here:

	Valid	Invalid
# inputs	$= 3$ (1)	< 3 (2) ; > 3 (3)
input values	$inp_1 \geq 0$ (4) $inp_2 \geq 0$ (5) $inp_3 \geq 0$ (6)	$inp_1 < 0$ (7) $inp_2 < 0$ (8) $inp_3 < 0$ (9)
input type	inp_1 int.(10) inp_2 double (11) inp_3 int (12)	inp_1 $\not\text{int}$.(13) inp_2 $\not\text{double}$ (14) inp_3 $\not\text{int}$ (15)
output type	output is double (16)	-
output value	output is ≥ 0 (17)	-

A test suite covering all valid and invalid classes could be the following. Note that I leave out all the valid classes in the invalid test cases for readability.

Test Input	Covered Classes	Expected Result
[100;5.2;5]	(1),(4),(5),(6),(10),(11),(12),(16),(17)	Some value
[]	(2)	Error
[1]	(2)	Error
[1;1.2]	(2)	Error
[1;2;3;4]	(3)	Error
[-1;1.0;1]	(7)	Error
[1;-1.0;1]	(8)	Error
[1;1.0;-1]	(9)	Error
["s";1.0;1]	(13)	Error
[1;"s";1]	(14)	Error
[1;1.0;"s"]	(15)	Error

3. For boundary value analysis, we have 3 inputs, and an output with respective lower and upper bounds:

	Input 1	Input 2	Input 3	Output
Lower Bound	0	0	0	0
Upper Bound	MaxInt	MaxInt	Max Int	MaxInt

With these, we add the following test cases to the existing test suite:

	Covered BVA Class	Expectation
[-1;10;10]	BLB inp 1	Error
[0;10;10]	LB inp 1	Some value
[1;10;10]	ALB inp 1	Some value
[MAXINT-1;10;10]	BUB inp 1	Some value
[MAXINT;10;10]	UB inp 1	Some value
[10;-1;10]	BLB inp 2	Error
[10;0;10]	LB inp 2	Some value
[10;1;10]	ALB inp 2	Some value
[10;MAXINT-1;10]	BUB inp 2	Some value
[10;MAXINT;10]	UB inp 2	Some value
[10;10;-1]	BLB inp 3	Error
[10;10;0]	LB inp 3	Some value
[10;10;1]	ALB inp 3	Some value
[10;10;MAXINT-1]	BUB inp 3	Some value
[10;10;MAXINT]	UB inp 3	Some value
[10;10;0]	LB out	0
[0.01;0;1]	ALB out	0.01

Note that I did not calculate the values here. What is essential is, that we expect some positive value.

4. Error guessing may suggest to test for $rate$ vs $1 + (rate/100)$, i.e. a test case [1;50;1] vs [1;150;1]. Another possible guess is to check whether the function rounds correctly, i.e. is the result a float with 2 digits.

5. For instance [-1;10;10] - Error - fails on the implementation. The implementation simply calculates with a negative amount, while an error was expected.

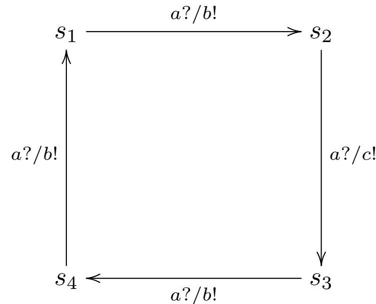


Figure 1: The FSM A, where s_1 is initial.

3 FSM testing

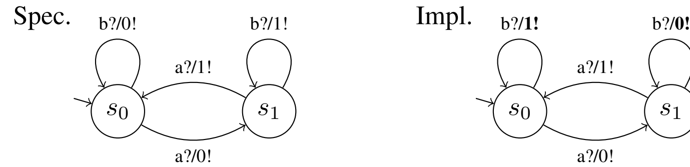
1. Describe the notion of soundness for FSM testing. *(2 points)*
2. Prove that the state tour method is sound. *(2 points)*
3. Prove that the state tour method is not complete. *(2 points)*
4. Use the transition test method to derive a test for the transition $s_2 \xrightarrow{a?/c!} s_3$. Describe the steps that you performed to obtain this test. *(2 points)*
5. Suppose you have tested a system implementation using FSM transition testing, and it passed all tests. Give four reasons why the implementation could not work as desired, despite the fact that FSM transition testing is complete. *(2 points)*

Answers:

1. A test (suite) is *sound* wrt the specification, if every *conforming* implementation passes the test suite. In the specific case of FSM testing, conformance is replaced by FSM equivalence. Examples of sound test (suites) are given by state tours, transition tours, or transition testing, cf. lecture 3 slide 16.
2. To show
$$\forall S \in \text{FSM}\ \forall \text{ state tour } \sigma\, of\, S\ \forall I \equiv S : I \text{ passes } \sigma$$
Therefore, let S be a specification FSM, let σ be a state tour for S, and assume $I \equiv S$, i.e. I is a conforming FSM. Assume $\sigma = a_1?/b_1\, a_2?/b_2\, \ldots a_n?/b_n!$. By definition of state tours
$$\lambda_S(\sigma) = \lambda_S(a_1?\, a_2?\, \ldots\, a_n?) = b_1!\, b_2!\, \ldots b_n!.$$
Then by definition of FSM equivalence, we have
$$\lambda_I(\sigma) = \lambda_I(a_1?\, a_2?\, \ldots\, a_n?) = b_1!\, b_2!\, \ldots b_n!,$$

i.e. $\lambda_I(\sigma) = \lambda_S(\sigma)$. We conclude: I passes σ. Therefore, the state tour method is sound.

3. We prove this by giving a counter example



   Obviously, $\sigma = a?/0!\ a?/1!$ is a state tour. However, the erroneous implementation on the right side is not found. This shows that the state tour is not complete.

4. A transition test for the transition $s_2 \xrightarrow{a?/c!} s_3$ consists of a synchronising sequence, a transfer sequence, the input alongside the expected output, and the state verification.

   There is no synchronising sequence in the given FSM . Hence, we assume the functionality of a reset button to synchronise with the initial state s_1. The transfer sequence to reach s_2 is given by $a?/b!$. We then apply input $a?$ and expect output $c!$. To verify that we're in s_3, we apply $a?\ a?\ a?\ a?$ and expect $b!\ b!\ b!\ c!$.

   The test case then becomes

   $$t = \{a?/b!\ a?/c!\ a?/b!\ a?/b!\ a?/b!\ a?/c!\}\,.$$

5. In case the implementation invalidates one or more of our assumptions. Among them

   - It is non-deterministic, or
   - it has more states than spec. Recall that this is an assumption to guarantee completeness of the transition tour.

   Furthermore, the implementation might be dependent on other systems, i.e. failed synchronisation etc. There might be issues with the testing tool, or errors in the test suite, e.g. "over-optimised" test suites.

   Note that these are not *all* reasons, but a mere illustration of possible reasons.

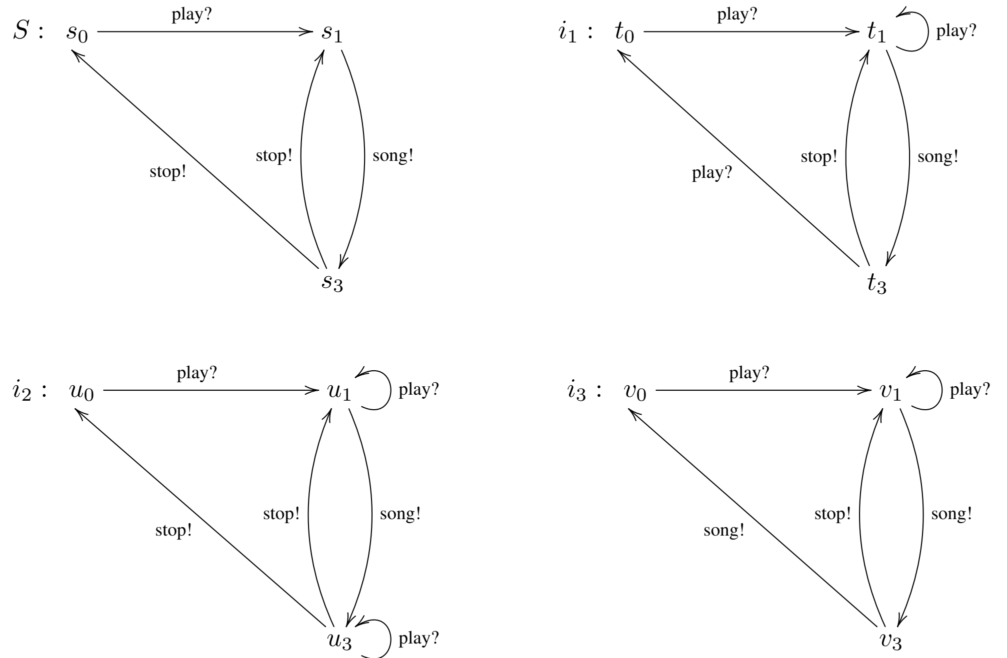


Figure 2: Transition systems S, i_1, i_2, and i_3; s_0, t_0, u_0, v_0 are initial states.

4 Ioco

The specification S in Figure 2 represents a simplified specification of an MP3 player. After pushing the play button, a song is started. When the song is over, the MP3 player either moves to a state where it starts a new song, or (if the play list is finished), it stops playing and waits for the user to push the play button again.

1. Add δ-transitions to the transition systems whenever required. *(1 points)*
2. Describe the role of quiescence in QTSs. *(2 points)*
3. Which of the IOLTSs i_1, i_2, i_3 in Figure 2 are ioco-correct implementations of S? If an implementation is incorrect provide a test case that fails on this implementation. *(7 points)*

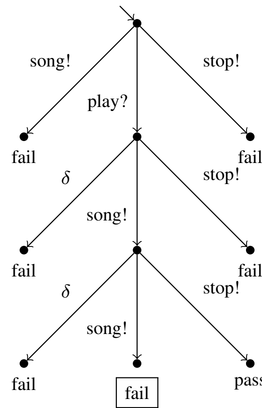
Answers:

1. We add a δ-self loop in s_0, t_0, u_0 and v_0, respectively. These states do not enable an output-action.

2. Quiescence describes the absence of outputs. Specification models need to account for lack of outputs in certain states, i.e. is no output *at all* desired. In ioco-theory quiescence is commonly denoted via δ-transitions. These transitions are added in *quiescent* states, i.e. states that do not enable outgoing output, or internal actions. For example, states s_0, t_0, u_0 and v_0 are quiescent.

3. Both i_1 and i_2 are ioco to S.

   IOLTS i_3 is not ioco. First of all, it is not input-enabled. For the sake of this exercise (Note from Marcus: I assume this was a typo?), let us assume that i_3 is input-enabled by adding a self-loop in v_3 with the label *play?*. A test case, that would fail i_3 is given by:

   The provided test contains a trace that ends in a leaf labeled with *fail* if composed with i_3, i.e. trace $play?\ song!\ song!$

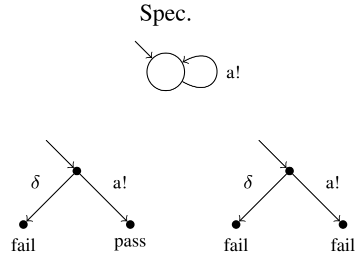
5 Proofs

Are the following statements true or false? For a true statement give a proof, for a false statement, give a counter example.

1. Suppose that we have a sound test suite for S. Suppose that we change one of the pass verdicts in a fail. Statement: the resulting test suite is still sound. *(3 points)*
2. Suppose that we have a sound test suite for S. Suppose that we change one of the fail verdicts in a pass. Statement: the resulting test suite is still sound? *(3 points)*
3. Suppose that we have a complete test suite for S. Suppose that we change one of the fail verdicts in a pass. Statement: the resulting test suite still complete? *(3 points)*

Answers:

1. Generally, this need not be true. We provide a counterexample:



The left hand side is a sound test case for the specification given above. Changing the pass label after *a!* from *pass* to *fail*, however, makes the test case on the right hand side not sound anymore. An implementation, which is ioco wrt the specification would now fail the test.

2. This statement is true. Let T be a sound test suite for S, and assume $I \sqsubseteq_{ioco} S$ for an input enabled QTS I. Because T is sound and $I \sqsubseteq_{ioco} S$, we know that

$$\forall t \in T : v_t(I) = pass.$$

This implies

$$\forall \sigma \in exec_t(I) = ctraces(t \parallel I) \cap ctraces(t) : a(\sigma) = pass,$$

i.e. all "encountered" traces will have the pass label.

Now let T' be the test suite created from T by changing some *fail* labels of traces in tests to *pass* labels. We can easily see that

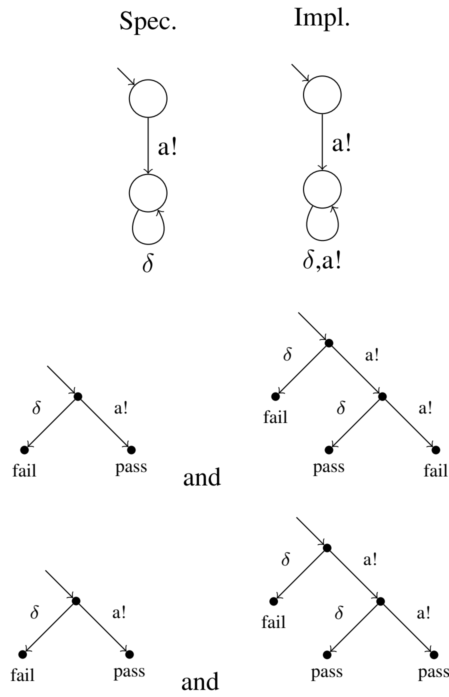
$$\forall t' \in T' : v_{t'}(I) = pass,$$

as we still have that, for all $t' \in T'$

$$\forall \sigma \in exec_{t'}(I) : a(\sigma) = pass.$$

Hence, T' is still sound.

3. Generally, this need not be true. We provide a counterexample:



Note that the above test suite is complete according to Proposition 6.5. by Brinksma, Timmer, Stoelinga (also see Assignment 5). However, changing the *fail* verdict in the top-right test case after trace *a! a!* to *pass*, makes it incomplete. To illustrate, note that the implementation above would now pass this test suite, even though it is not ioco wrt the specification.