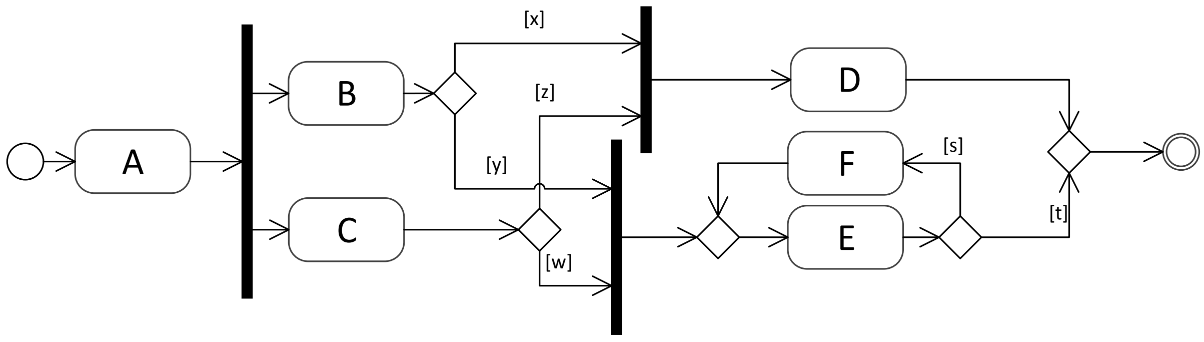
4th Seminar – Analysis and Testing of Models – Solutions

1 Static analysis of process models

Let's analyse the following process model.



a. What conditions will ensure that the process model is fully specified (i.e. consistent, without contradiction)?
Solution
Analogously to state machines, a process model is fully specified if and only if at every decision node **at least one** of its outgoing arc's guard condition is true – in other words, we can always choose at least one path to go on. In order for this to be true the following is sufficient: the set of guard conditions of a decision is a complete system of conditions. Actually, the following weaker condition is still sufficient: it has to be a *conditionally* complete system of conditions, which means that the system of conditions only has to be complete if another condition holds (like only if we take a certain path in the model), and not all the time (this matters only at the third decision).
   - Therefore, the conditions are:
     - $x \lor y$ (after the execution of the activities A and B)
     - $z \lor w$ (after the execution of the activities A and C)
     - $w \land y \to s \lor t$ (the right side must be true after every execution of the loop).

b. What further conditions will ensure that the process model is also deterministic?
Solution
A process model is deterministic if and only if at every decision node **at most one** of its outgoing arc's guard condition is true – in other words we can always choose at most one path to go on, and not more. In order for this to be true the following is sufficient: the set of guard conditions of a decision is a mutually exclusive system of conditions. Actually, the following weaker condition is still sufficient: it has to be a conditionally mutually exclusive system of conditions, which means that it only has to be mutually exclusive if a condition holds (like only if we take a certain path in the model), and not all the time (this matters only at the third decision).
   - Therefore, the conditions are:
     - $\neg(x \land y)$ (after the execution of the activities A and B)
     - $\neg(z \land w)$ (after the execution of the activities A and C)
     - $w \land y \to \neg(s \land t)$ (the right side must be true after every execution of the loop)
   - We can simplify these conditions if we consider them together with the conditions from task a.:
     - $y = \neg x$ (since x and y can't be true or false at the same time, so x XOR y)
       Detailed deduction: $(x \lor y) \land (\neg(x \land y)) \equiv (x \lor y) \land (\neg x \lor \neg y)$
       $\equiv (x \land \neg x) \lor (x \land \neg y) \lor (y \land \neg y) \lor (y \land \neg x) \equiv (x \land \neg y) \lor (y \land \neg x) \equiv x$ XOR y
     - $w = \neg z$ (analogously to the previous one)

c. What further conditions will ensure that the process model is also free of deadlock?
Solution
Deadlock: waiting in one state forever (because we can't leave it for some reason). First of all, we obviously have to assume that the tasks themselves will eventually terminate – if it's not the case then the process model is not the source of the non-termination. A deadlock can happen if

after the fork node one decision leads to the upper join node while the other decision leads to the join node below. In this case both join nodes will wait forever for the second token, which will never arrive. Generally it is bad if only the token from one of the incoming branches of a join node arrives.

- Therefore the following conditions are necessary to ensure the "proper meeting of tokens" at the same join node:
  - $x = z$
  - $y = w$

d. What further conditions will ensure that the process model will eventually terminate?

Solution

First of all, we obviously have to assume that the tasks themselves will eventually terminate – if it's not the case then the process model is not the source of the non-termination. The process will not terminate if there is a deadlock, but we guaranteed it with the previous conditions that there will be no such deadlock. The other source of non-termination could be the livelock, which means that the process is stuck in a set of states from which it can't leave (so in an infinite loop). The infinite loop will occur only if we choose the lower path and the exit condition will never be true.

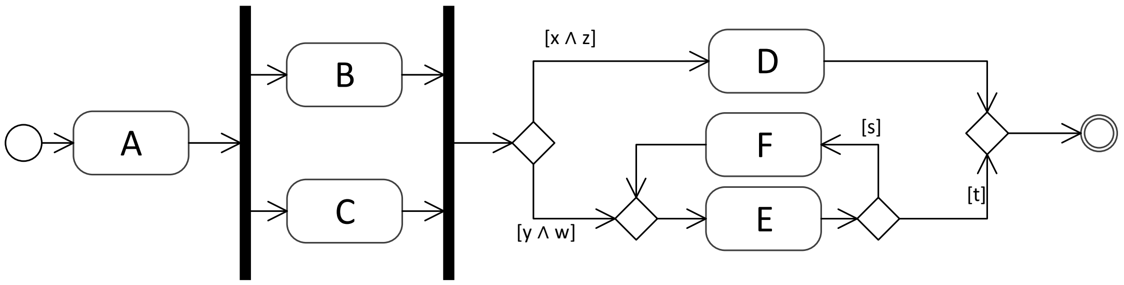
- Therefore, the necessary conditions are:
  - If $\neg x$ (so y and w is true), then *sooner or later* t must be true.
    Side note: similarly to the above conditions from the previous tasks we can write this condition as a logic formula. There is a sooner-or-later symbol in temporal logics, the so called *future* operator (F), but temporal logics are not in the scope of this course: $\neg x \implies \mathsf{F}t$

e. Is the process model well-structured? If not, then how could we make it into an equivalent, but well-structured model? Does this help with the previous problems?

Solution

Using the method presented at a previous seminar we can show that the process is not well-structured. We can make it well-structured by putting a join node right after B and C, and putting a decision node after that join whose guard conditions are equivalent to the previous two decision node's guard conditions. The new guard conditions will reflect the conditions given in task c. to avoid deadlock.

This modification will eliminate the possibility of the deadlock discussed in task c., but it doesn't guarantee anything else (like absence of livelock, or determinism), although it improves the model's clarity.



2 Dynamic analysis with testing

We have the following *requirements* for function `f()`:

R1 Function `f()` must produce at least one output during its execution.

R2 Function `f()` must terminate eventually regardless of the input sequence.

R3 The last output produced by function `f()` must be 0.

The following code demonstrates a possible implementation of the function in the C programming language:

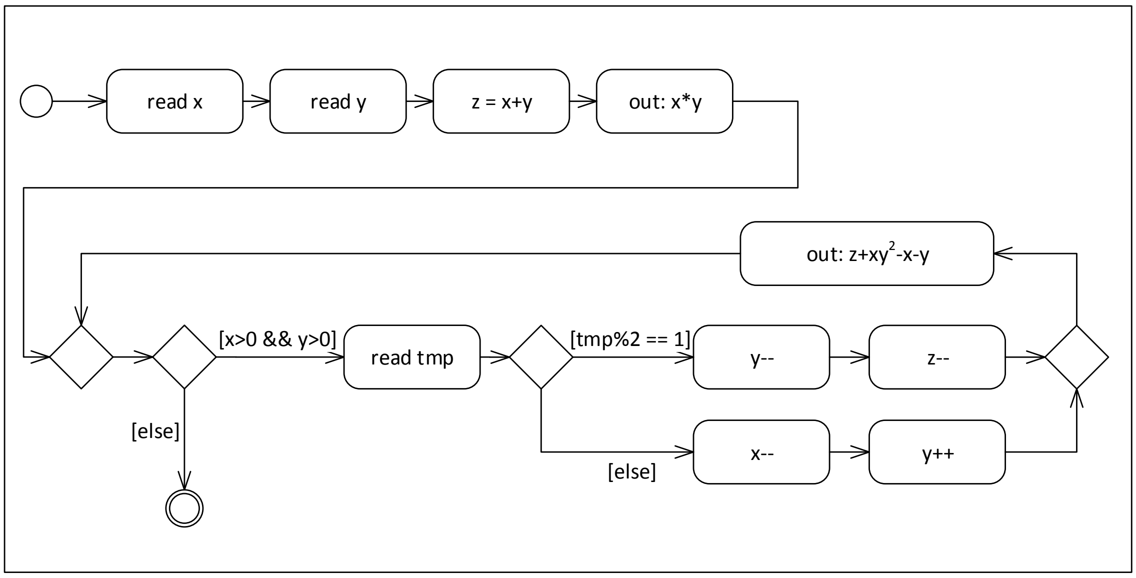
```
int readInput();
void writeOutput(int out);

void f() {
  int x = readInput();
  int y = readInput();
  int z = x + y;
  writeOutput(x * y);
  while (x > 0 && y > 0) {
    if (1 == readInput() % 2) {
      y--;
      z--;
    } else {
      x--;
      y++;
    }
    writeOutput(z + x * y * y - x - y);
  }
}
```

Let's verify the correctness of the function in the following steps!

a. Model the control flow of `f()` as a process model!

Solution

Remark: reading an input should be a separate task even if in the code it occurs in a condition. This is because we don't know the exact properties of the input channel, the inputs could come from a far away database, so reading it takes longer than evaluating a guard condition.



b. Why can we be sure that requirement R1 holds?

Solution

It can be seen on the process model that every execution path contains the first output task.

c. Why can we be sure that requirement R2 holds?

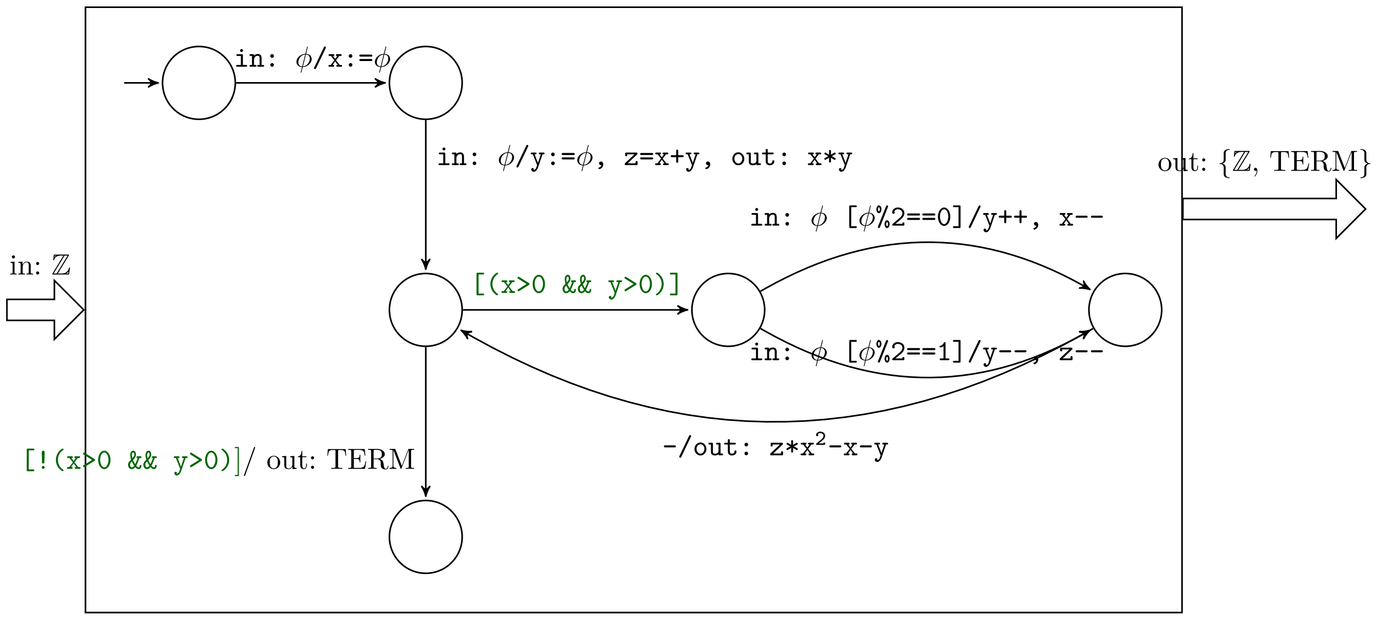
Solution

The termination could be affected only by the loop. We enter the loop when x and y are positive; x never increases, so the second branch of the `if` condition would terminate the loop eventually. If we are not decreasing x, then we are decreasing y in the other branch, which will also terminate the loop eventually. There is the possibility that y will overflow and becomes negative, but that will also terminate the loop.

d. (Extra task.) Create a state machine that is equivalent with function `f()`. Model the `readInput()` calls as an input channel and the `writeOutput()` calls as an output channel. Model the termination of the function by producing a special output in the state machine while transitioning into an absorbing state (a state without outgoing transitions).

Solution

Let x, y, z be state variables, which we can use in guard conditions and actions (like in Yakindu). We could model every code line as a different state, but we can merge some states and model multiple code lines as actions on the same transition (like modelling the loop section with two loop transitions on the same state). Both are good solutions, but from a practical point of view it is better to separate testing the loop condition from the internals of the loop.



e. We are verifying requirement R3 using testing. Our test case is the input sequence $t_1 = \langle 2, 3, 5, 7, 11, 13, ...\rangle$. Do we detect any errors?
Solution
The program is doing the following:
- $t_1 = \langle 2, 3, 5, 7, 11, 13, ...\rangle$
- $x = 2$
- $y = 3$
- $z = 5$
- out: 6
- we enter the loop
- entering the true branch
- $y = 2$
- $z = 4$
- out: $4 + 8 - 2 - 2 = 8$
- staying in the loop
- entering the true branch
- $y = 1$
- $z = 3$
- out: $3 + 2 - 2 - 1 = 2$
- staying in the loop
- entering the true branch
- $y = 0$
- $z = 2$
- out: $0 + 2 - 2 - 0 = 0$
- exiting the loop
- execution is over

So requirement R3 is not violated.

f. Calculate *instruction coverage* on the process model: during the execution of the test case what portion of the tested code was executed. How does this metric relate to the control flow model?
Solution
We execute 10 instructions and we leave out 2, so the instruction coverage is $10/12 \approx 83\%$. On the control flow model we can count the number of reached tasks, so we can calculate task coverage (we count a decision-merge pair as one).

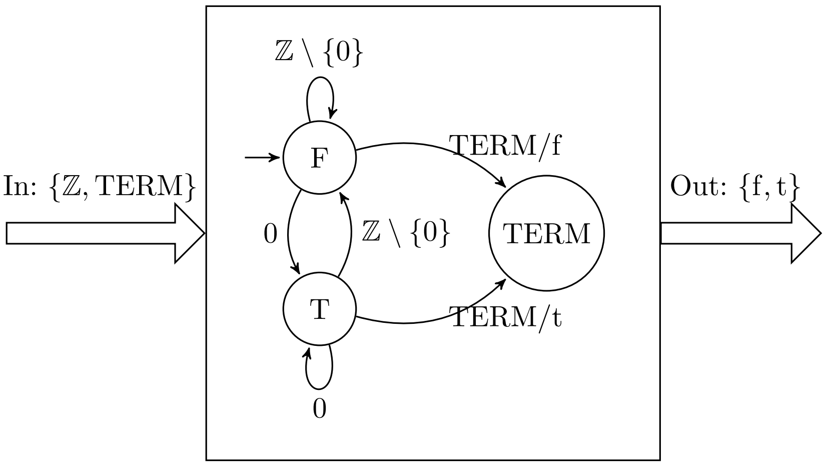
g. Our second test case for verifying R3 is the input sequence $t_2 = \langle 1, 2, 4, 1, 2, 4, ...\rangle$. Do we detect any errors? How much is the combined instruction coverage of the test suite consisting of the two above test cases?
Solution
$t_2 = \langle 1, 2, 4, 1, 2, 4, ...\rangle$ The program does the following:

- $x = 1$
- $y = 2$
- $z = 3$
- out: 2
- entering the loop
- entering the false branch
- $x = 0$
- $y = 3$
- out: $3 + 0 - 0 - 3 = 0$
- exiting the loop
- execution is over

So requirement R3 is not violated.

h. (Extra task.) What kind of test coverage metrics can be calculated on the previous state machine?
Solution
The coverage metrics should be approximately the same if we build the models the same way – but can differ greatly if we merged some states as suggested. On the state machine we can calculate the state coverage (it could easily be a 100% if we merged some states), but the transition coverage would probably be more interesting.

i. Create a *test oracle* state machine that can decide whether R3 holds or not based on the input and output sequence and termination of `f()`. What does the oracle do in case of the previous test inputs?
Solution
The created test oracle decides the result of the test based on the test case and the output of the function, and gives a True/False result. The important states are T (”the last output was 0”) and F (”the last output was not 0”), and the state machine switches between them based on the outputs of the tested state machine (which is the input of the test oracle). If the tested state machine terminate, then the test oracle transitions to an absorbing state giving the appropriate output. We could specify the initial state as state F, or we could add an other state to the oracle that stand for ”no output yet”. We can simulate the oracle and the program with the previous test case to see that the requirement isn't violated.



j. Give a test case that detects an error in the program! What implicated that our previous test cases weren't perfect and we needed to extend them?
Solution
A possible test case: $t_3 = \langle -1, -1, -1, ... \rangle$. The program does the following:

- $x = -1$
- $y = -1$
- $z = -2$
- out: 1
- skipping the loop
- execution is over

This execution violates requirement R3, which is detected by the test oracle too.

The lesson is: it's not enough to calculate instruction and branch coverage, we also need to measure the *robustness* of the code by testing it with outlier or critical parameters and with their immediate surrounding (like 0 in case of x and y). This testing method will prove very useful for later programming homeworks.
This is a dynamic verification method since we actually had to *execute* the instructions.

k. (Homework.) Let's add the input sequences $t_3 = \langle 0, 1, 2, 3, 4, 5, ... \rangle$ and $t_4 = \langle 1, 2, 3, 4, 5, 6, ... \rangle$ to our test case collection. Do we detect any errors? How does the instruction coverage change?
Solution
Homework.

l. Specify the exact conditions under which requirement R3 doesn't hold and recommend some fixes!
Solution
Lets consider the possibilities.
   - A trivial solution would be `void f() { writeOutput(0); }`, but we would like a solution that reflects the behaviour of the original code.
   - if either x or y is negative and the other one is not zero, then we skip the loop and the last output would be $x \cdot y$. In this case we have to decide what the proper behaviour should be, like writing a 0 output instead of entering the loop. Or we could discard any negative input if we consider them invalid.
   - The relation $z = x + y$ always holds (inspect the two branches inside the loop and see how the values change), so the output inside the loop will only contain the product term $x \cdot y \cdot y$. If we exit the loop, then one of the variables is zero, so the output will be zero too. This is not true if a variable overflows and becomes negative.
   - So we only have to ensure that y won't overflow from `MAXINT` to `MININT`. We could handle this by raising an error and stop if y exceeds some upper boundary, or we detect the overflow and write a zero output, etc.