# TRIAL EXAM B
Software Engineering using Formal Methods
TDA293 / DIT270

also serving as additional training material for the course
Formal Methods for Software Development, TDA294/DIT271

## Assignment 1 Linear Temporal Logic(LTL) (10p)

Consider the following PROMELA model:

```
byte mode  = 1;
byte count = 0;

active proctype m() {

endLoop:
  if
    :: mode = 1
    :: mode = 2
  fi;
  do
    :: mode == 1 && count < 30 -> count++
    :: mode == 2 -> count = 0; goto endLoop
    :: mode == 3 -> break
    :: else -> goto endLoop;
  od;
  count = 0
}

active proctype n() {
  do
   :: mode = 3
  od
}
```

(a) *(8p)* Formalise the following properties in LTL and indicate for each whether it is valid or not valid with respect to the above PROMELA model. (You do not need to provide explanation.) Assume the scheduler guarantees weak-fairness.

1. `count` is never greater-or-equal than 30.
2. If in some state `count` becomes greater than 0, it remains strictly positive until, eventually, `mode` becomes greater than 1.
   (Hint: With "until, eventually," we mean the strong until.)
3. If `count` is greater-than 0 in some state it will eventually be reset to 0 at some later point.
4. `mode` will eventually become 3.

**Solution**

*(a)*

1. $\neg\Diamond(count >= 30)$ *(or also accepted:* $\Box(count < 30)$*); invalid*

*2.* $\Box(count > 0 \rightarrow ((count > 0) U (mode > 1)))$*; valid*

*3.* $\Box(count > 0 \rightarrow \Diamond(count == 0))$*; valid*

*4.* $\Diamond(mode == 3)$*; valid*

(b) *(2p)* We now alter the scenario from (a), in so far as weak fairness is no longer assumed. For which of the four properties from (a) does this change the validity/invalidity status of the property?

**Solution**
*(b)*
*2., 3., and 4. become invalid.*

---

**Assignment 2 (First-Order Sequent Calculus)** (12p)

We work here in untyped first-order logic with the trivial type $\top$, which is omitted in the formulas below.

Let $p$ denote a predicate of arity 2 and $c, d$ be constant symbols. Prove that the following sequent is valid, using the *first-order sequent calculus.* For each step, name the rule you apply. If you invent a *new* constant, state that clearly.

You are only allowed to use the calculus rules presented in the lectures.

Your task is to build a proof for the following sequent:

$\forall x; \forall y; (p(x, y) \to p(y, x)),$
$\forall x; \forall y; \forall z; ((p(x, y) \wedge p(y, z)) \to p(x, z)),$
$\exists z; (p(d, z) \wedge p(z, c))$
$\Rightarrow$
$p(c, d)$

*Hint:* You may abbreviate formulas, but only if you clearly describe your abbreviations. The proof gets easier if you start with eliminating the existential quantifier, and then instantiate the formula $\forall x; \forall y; \forall z; \ldots$

**Solution**

exLeft introduce new constant e,
andLeft
allLeft on transitivity formula instantiate with d,
allLeft on resulting formula and instantiate with e,
allLeft on resulting formula and instantiate with c,
allLeft on symmetry formula and instantiate with d,
allLeft on result and instantiate with c,
impLeft on resulting formula from previous step
Two goals (1) and (2):

Goal (1) with $p(d, e) \wedge p(e, c) \to p(d, c)$ on left side:
Apply impLeft (two goals (3) and (4)

Goal (3) with $p(d, e) \wedge p(e, c)$ on right side:
Apply rule andRight (two new goals (3a) and (3b)): (3a) close (3b) close

Goal (4) with $p(d, c)$ on right and left side: close

Goal (2) with $p(c, d)$ on left side:
close with p(c,d)

## Assignment 3 (PROMELA and SPIN) (10p)

```
mtype = {vegetarian, meat, fish};

chan order = [5] of {mtype, int};
chan out = [5] of {mtype, int};


active [2] proctype cook() {
  // implement
}

active [3] proctype guest() {
  // implement
}
```

In a restaurant the cooks receive orders from their guests. An order includes the requested meal (vegetarian, meat or fish) and the guests `_pid`. The cooks put the prepared meal in the `out` channel from where the guest can pick them up.

The procedure for guests and cooks in slightly more detail is as follows:

**Guests:**

1. Each guest chooses arbitrarily among the possible meals, composes and sends out her/his order.
2. The guest waits then for her/his order to arrive and takes it if it is addressed to her/him.
3. Afterwards, the guest orders either a further meal or leaves the restaurant.

**Cooks:**

1. A cook takes an order and prepares the meal.
2. The finished meal is then put in the `out` channel awaiting for client to pick it up.

**Tasks:**

(a) *(8p)* Implement the proctypes `cook` and `guest` according to the specified protocol.
*Hint:* Pattern matching on the content of variables can be achieved by using `eval`(*var*). For instance, `channel ? eval(i)` is only executable if the message in `channel` has the value of variable `i`.
**Solution**

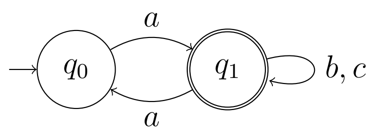
```
mtype = {vegetarian, meat, fish}

chan order = [5] of {mtype, int};
chan out = [5] of {mtype, int};


active [2] proctype cook() {

  mtype orderedMeal;
  int client;

endOrderLoop:
  do
    :: order ? orderedMeal, client; out ! orderedMeal, client
  od
}

active [3] proctype guest() {

  mtype meal, receivedMeal;

endStartOrder:
  if
    :: meal = vegetarian
    :: meal = meat
    :: meal = fish
  fi;
  order ! meal, _pid;
  out ? receivedMeal, eval(_pid);
served:  assert receivedMeal == meal;
  if /* leave the restaurant or continue ordering */
    :: goto exitRestaurant;
    :: goto endStartOrder;
  fi;
exitRestaurant:
  printf("Bye")
}
```

(b) *(2p)* Explain how you ensure that a guest takes only her/his meal and not the one of someone else. Put an assertion into the code ensuring that the received meal is the ordered one.
**Solution**

*pattern matching on _pid; assertion see above*

## Assignment 4 (Büchi Automata, $\omega$-expressions, and LTL) (9p)

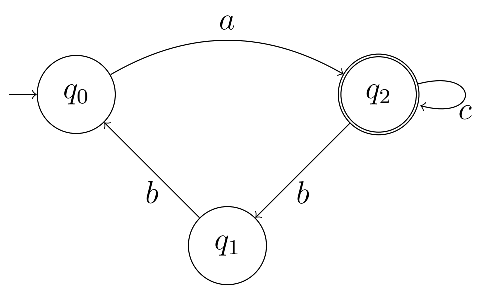
(a) *(2p)* Give the $\omega$-expression representing exactly the language recognised by the Büchi automaton below.



(b) *(2p)* Give the Büchi automaton recognising exactly the language represented by the following $\omega$-expression:

$$a((aa)^*bb)^\omega$$

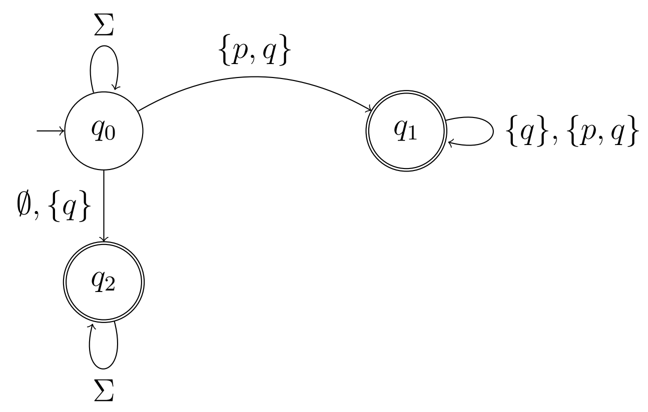
(c) *(2p)* Give the $\omega$-expression representing exactly the language recognised by the Büchi automaton below.



(d) *(3p)* Consider the LTL formula $\Diamond(p \to \Box q)$

Does the following Büchi automaton accept exactly those runs satisfying the the above formula? Explain your answer. (We suggest a few sentences.)

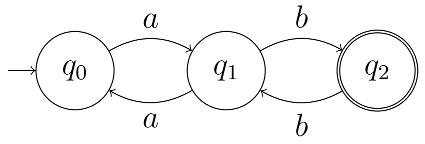
$\Sigma := \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$



**Solution**

(a) $a(aa+b+c)^\omega$

(b)



(c) $a(bba+c)^\omega$

(d) The answer to the question is yes. The reason is that both, the runs that satisfy the formula, and the runs ($\omega$-words of states) that are accepted by the Büchi automaton, can be characterised in the same way. It is exactly the set of runs which reach a state where either $p$ is false, or where $p$ is true and $q$ remains true from thereon.

## Assignment 5 (Java Modeling Language) (13p)

*(The description of this assignment has two pages.)*
A flight route is divided into a sequence of legs, each of which is a straight line between a startpoint $(startX, startY)$ and an endpoint $(endX, endY)$.

Consider the following Java classes:

```
public class Leg {

   private /*@ spec_public @*/ int startX;
   private /*@ spec_public @*/ int startY;
   private /*@ spec_public @*/ int endX;
   private /*@ spec_public @*/ int endY;

  // some methods

}


public class FlightRoute {

 private /*@ spec_public @*/ int size;
 private /*@ spec_public nullable @*/ Leg[] route;

 public void append(Leg leg) { ... }

 public int replace(Leg oldLeg, Leg[] newLegs) { ... }

 // some more methods
}
```

In the following, observe the usual restrictions under which Java elements can be used in JML specifications. You are not allowed to introduce any other methods, neither for implementation, nor for specification purposes.

1. Augment the class `Leg` with a JML specification stating that start and end point are not the same.
2. The class `FlightRoute` manages its legs using the array `route` of a fixed size. The integer typed attribute `size` points to the next free element of the array not yet occupied by a leg, i.e., all array components up-to but excluding `size` are non-null.

   Augment the class `FlightRoute` by JML specifications stating that

   (a) the attribute `route` is never `null`.

   (b) the attribute `size` is never negative and less-or-equal than the length of the array `route`.

(c) the array *route* does not contain duplicates (the same object does not occur twice)

(d) a route is a consecutive sequence of legs, i.e., a route does not have holes.

(e) if method `append` is called in a state where

- `size` is strictly smaller than the length of array `route`
- the given parameter `leg` appended to the route does not violate the property stated in items 2c and 2d

then the method terminates normally and in its final state

- the handed over leg has been appended to (added to the end of) the route and
- the field `size` has been updated correctly.

(f) if method `replace` is called in a state where

- the route contains the object `oldLeg`,
- the array `route` has enough space to store the route resulting from replacing leg `oldLeg` by a *non-empty* sequence of new legs `newLegs`, and,
- the replacement does not violate properties 2c and 2d

then the method terminates normally and in its final state

- leg `oldLeg` is no longer part of the route,
- the returned value is the index of the array component containing `oldLeg`
- the given sequence `newLegs` has been inserted at the index of the old leg and all to its right have been shifted according to the length of the inserted sequence. All preceding legs of the route remain unchanged.
- the attribute `size` has been updated correctly to reflect the new route

For the method contracts, please do not forget to provide the `assignable` clause.

**Solution**

*[1; 1+1+1+2+2+5]*

```
public class Leg {

  private /*@ spec_public @*/ int startX;
  private /*@ spec_public @*/ int startY;
  private /*@ spec_public @*/ int endX;
  private /*@ spec_public @*/ int endY;

 /*@ public invariant startX != endX || startY != endY; @*/

  // some methods
}

public class FlightRoute {
  //@ public instance invariant size >= 0 && size <= route.length;
  private /*@ spec_public @*/ int size;

  // route not null; attention removing the nullable is too strong
  // as non_null requires the array elements to be non_null too
  /*@ public instance invariant route != null; @*/

  /*@ public instance invariant // no duplicates
    @ (\forall int i;\forall int j;
    @         i>=0 && i<j && j<size;route[i]!=route[j]);
    @ public instance invariant // consecutive
    @   (\forall int i; i>=0 && i<size-1;
    @                     route[i].endX == route[i+1].startX
    @                  && route[i].endY == route[i+1].startY );
    @*/
  private /*@ spec_public nullable @*/ Leg[] route;

  /*@ public normal_behavior
    @ requires size < route.length;
    @ requires (\forall int i; i>=0 && i<size; route[i] != leg);
    @ requires size > 0 ==> (leg.startX == route[size - 1].endX &&
    @                              leg.startY == route[size - 1].endY
);
    @ ensures route[\old(size)] == leg;
    @ ensures size == \old(size) + 1;
    @ assignable size, route[size];
    @*/
  public void append(Leg leg) { ... }

  /*@ public normal_behavior
    @ requires (\exists int i; i>=0 && i<size; route[i] == oldLeg);
    @ requires newLegs.length >= 1;
    @ requires size <= route.length - newLegs.length + 1;
    @ requires (\forall int i; i>=0 && i<size;
    @                  (\forall int j; j>=0 && j<newLegs.length;
    @                            route[i] != newLegs[j]));
    @ // The following requires clause was accepted as sufficient,
    @ // even if it does not check whether newLegs has holes already.
    @ requires size > 0 ==>
    @   (   newLegs[0].startX == oldLeg.startX
    @    && newLegs[0].startY == oldLeg.startY
    @    && newLegs[newLegs.length - 1].endX == oldLeg.endX
    @    && newLegs[newLegs.length - 1].endY == oldLeg.endY);
    @ // The next requires clause was not asked for; any other
    @ // solution or ignoring the issue was accepted, too.
    @ requires
    @  (\forall int i; i>=0 && i<newLegs.length; newLegs[i] != oldLeg);
    @ ensures (\forall int i; i>=0 && i<size; route[i] != oldLeg);
    @ ensures \old(route[\result])==oldLeg;
    @ ensures (\forall int i; i>=0&&i<\result;
    @                         route[i]==\old(route[i]));
    @ ensures (\forall int i; i>=\result && i<\result+newLegs.length;
    @                         route[i]==newLegs[i - \result]);
    @ ensures (\forall int i;
    @                  i>=\result+newLegs.length && i<size;
    @                  route[i]==\old(route[i-newLegs.length+1]);
    @ ensures size == \old(size) + newLegs.length - 1;
    @ assignable size, route[*];
    @*/
  public int replace(Leg oldLeg, Leg[] newLegs) { ... }
  // some more methods
}
```

**Assignment 6 (Loop-Invariant)** (6p)

Consider the following JML annotated method:

```
/*@ public normal_behavior
  @ requires true;
  @ ensures
  @  (\forall int i; i>=0 && i< values.length;
  @      \result[i] == values[values.length - 1 - i] );
  @*/
public int[] reverse(int[] values) {
  int[] out = new int[values.length];
  int i = 0;
  while (i<values.length) {
    out[i] = values[values.length-1-i];
    i++;
  }
  return out;
}
```

Provide a strong enough loop invariant for method `reverse` such that the method's post-condition can be verified. Provide also a variant (decreasing term) and the loop's assignable clause as precise as possible.

**Solution**

```
/*@ loop_invariant    i>=0
  @                && i <= values.length
  @                && (\forall int j;
  @                            j>=0 && j<i;
  @                            out[j]==values[values.length-j-1]);
  @ decreasing values.length - i;
  @ assignable i, out[*];
  @*/
```

(total 60p)