TRIAL EXAM C
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 PROMELA (12p)

In this assignment, we model a small part of a wifi network. A number of devices, modelled by the process `Device`, compete to get access to the network, which however has limited capacity (3 in our example).

Consider the following PROMELA model.

```
#define numOfDevices 5
#define limit 3

byte numOfUsers = 0;
chan ch = [0] of { byte, bool };

proctype Device(byte i) {
  bool answer;
  do
    (to be filled in by you)
  od
}

active proctype AccessControl() {
  byte id;
  do
    :: ch ? id , _ ->
       if
         :: numOfUsers < limit -> ch ! id, true
         :: else -> ch ! id, false
       fi
  od
}

init {
  byte i = 0;
  atomic {
    do
      :: (i >= numOfDevices) -> break
      :: else -> run Device(i); i++
    od
  }
}
```

Note that we do not actually model the network to be accessed. Rather, we only model the competing devices, plus a single `AccessControl` process which grants or denies access, depending on the number of devices currently accessing the network. A single channel, called `ch`, is used to communicate access requests (where the second argument does not matter), permissions (**true**), and denials (**false**).
Your answers to the questions below should remain valid even if the numbers in the definitions of `numOfDevices` and `limit` change.
(For the continuation of this assignment, see next page.)

(a) [8p]
Complete the process `Device`, according to the following instructions. Only the place marked by "*(to be filled in by you)*" should be completed, everything else in the above PROMELA model should be left unchanged. After being granted access, device n enters the network by incrementing `numOfUsers`, and printing out "`device` n `enters network`". Devices whose request gets denied print out "`device` n `cannot enter now`". Those devices which entered the network perform activities therein, here modelled by printing out "`device` n `using network`", and after that leave the network, by decrementing `numOfUsers`, and printing out "`device` n `leaves network`". In both cases (whether the device was denied access, or granted access and entered-used-left the network), the same device will start over by sending a new (identical) request, and all that infinitely often.
Your solution has to ensure that `numOfUsers` *never* exceeds the `limit`. At the same time, it would be too restrictive to only allow, for instance, that only one device uses the network at once. Instead, your solution has to allow runs with up to `limit` devices using the network at once.

(b) [1p]
Explain briefly why your solution guarantees that `numOfUsers` *never* exceeds the `limit`.

(c) [1p]
Write a separate process that allows to verify this property with SPIN (without using LTL).

(d) [1p]
Explain briefly why your solution allows `numOfUsers` to reach `limit`.

(e) [1p]
Write a separate process that allows to confirm this using SPIN (without using LTL).

Solution
(a)

```
proctype Device(byte i) {
  bool answer;
  do
    :: ch ! i, false;
       if :: atomic { ch ? eval(i), true;
                      numOfUsers++; }
             printf("device %d enters network\n", i);
             printf("device %d using network\n", i);
             numOfUsers--;
             printf("device %d leaves network\n", i)
          :: ch ? eval(i), false;
             printf("device %d cannot enter now\n", i)
       fi
  od
```

```
}
```

Another alternative:

```
proctype Device(byte i) {
  bool answer;
  do
    :: ch ! i, false;
       atomic {
         ch ? eval(i), answer;
         if
           :: answer -> numOfUsers++;
                          printf("device␣%d␣enters␣network\n", i)
           :: else -> printf("device␣%d␣cannot␣enter␣now\n", i)
         fi  }
       if
         :: answer -> printf("device␣%d␣using␣network\n", i);
                      numOfUsers--;
                      printf("device␣%d␣leaves␣network\n", i)
         :: else
       fi
  od
}
```

(b+d)
(The explanation is not given here.)

(c)

```
active proctype VerifierB() {
  assert(numOfUsers <= limit)
}
```

(e)

```
active proctype VerifierC() {
  assert(numOfUsers != limit)
}
```

If SPIN finds a failing run, that shows that `numOfUsers` can reach `limit`, which is what we wanted to show.

Assignment 2 Linear Temporal Logic (LTL) (10p)

Consider the following PROMELA model:

```
byte x = 0;
bool b = false

active proctype P() {
  do
    :: x < 20 -> x = 20; b = true
    :: x >= 0 -> if
                   :: x < 30 -> x++
                   :: else -> x = 10
                 fi
  od
}
```

Take your time to understand the behavior of `P`. Then consider the following properties, each of which *might or might not* hold:

1. `b` will be **true** at some point.
2. `x` will always be ≥ 10.
3. At some point, `x` will be 10.
4. At some point, `x` will be 11.
5. From some point on, `x` will always be ≥ 10.
6. `x` will infinitely often be 11.
7. If `b` will never be **true**, then `x` will infinitely often be 11.

(a) [6p]
Formulate each of the properties 1. - 7. in Linear Temporal Logic.

(b) [4p]
For each of the properties 1. - 7., tell whether or not the property is valid in the transition system given by the above PROMELA model. (You don't need to explain your answer.)

Solution

(a)

1. `<>b`
2. `[](x >= 10)`
3. `<>(x == 10)`

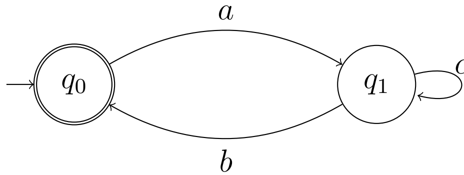
4. `<>(x == 11)`
5. `<>[](x >= 10)`
6. `[]<>(x == 11)`
7. `(!<>b) -> []<>(x == 11)`

(b)

1. *invalid*
2. *invalid*
3. *valid*
4. *invalid*
5. *valid*
6. *invalid*
7. *valid*

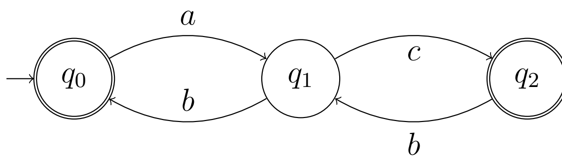
Assignment 3 (Büchi Automata and Model Checking) (8p)

(a) [2p]
Give the ω expression describing the language accepted by the following Büchi automaton:



Solution
$(ac^*b)^\omega$

(b) [3p]
Give the ω expression describing the language accepted by the following Büchi automaton:

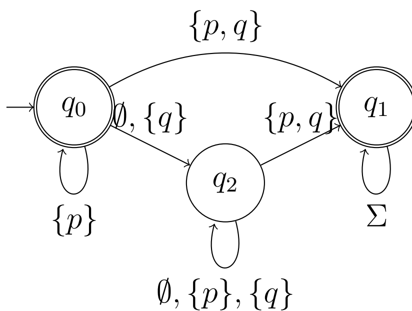


Solution
$a(ba + cb)^\omega$

(c) [3p]
Give a Büchi automaton that accepts exactly those runs satisfying the LTL formula:

$$\Box p \vee \Diamond(p \wedge q)$$

Solution
$\Sigma := 2^{\{p,q\}}$



Assignment 4 (First-Order Sequent Calculus) (8p)

Prove the validity of the following untyped first-order formulas, only using the sequent calculus. You are only allowed to use the rules presented in the SEFM lectures! Provide the name of each rule used in your proof as well as the resulting sequent, and make clear on which sequent you have applied the rule. When applying a quantifier rule, justify that the respective side condition is fulfilled.

(a) [4p]

$$\neg(\forall\ x; (\neg p(x) \land \neg q(x))) \to \exists\ x; (p(x) \lor q(x))$$

(b) [4p]

$$\exists\ x; (p(x) \lor q(x)) \to \neg(\forall\ x; (\neg p(x) \land \neg q(x)))$$

Solution

In this document, we only provide the names of the used proof rules. In the exam, you were requested to give more information, including the resulting sequents, for each proof step. See the question text.

(a) `impRight`,
`notLeft`,
`allRight`, $x \mapsto c$, *c fresh constant,*
`exRight` *with* c,
`orRight`, `andRight`, *gives:*
1) `notRight` *with* c, `close` *for* $p(c)$
2) `notRight` *with* c, `close` *for* $q(c)$.

(b) `impRight`,
`notRight`,
`exLeft`, $x \mapsto c$, *c fresh constant,*
`allLeft` *with* c,
`andLeft`,
`notLeft` *on* $p(c)$ *and* $q(c)$,
`orLeft` *gives:*
1) `close` *on* $p(c)$,
2) `close` *on* $q(c)$.

Assignment 5 (Java Modeling Language) (10p)

Consider the JAVA classes `Interval` and `IntervalSeq`:

```
public class Interval {
    private final int start, end;

    public Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public int getStart() {
        return start;
    }

    public int getEnd() {
        return end;
    }
}

/**
 * Class to represent sequence of intervals.
 */
public class IntervalSeq {

    protected int size = 0;

    protected Interval[] contents = new Interval[1000];

    /**
     * Insert a new element in the sequence;
     * it is not specified in which place
     * the element will be inserted
     */
    public void insert(Interval iv) {
        // ...
    }

    // more methods

}
```

In the following, observe the usual restrictions under which JAVA elements can be used in JML specifications.

(For the continuation of this assignment, see next page.)

(a) [3p]
Augment class `Interval` with JML specification stating that `getEnd()` is always $\geq$ `getStart()`.

(b) [7p]
In class `IntervalSeq`, the field `size` holds the number of `Interval` objects which have yet been inserted into the `IntervalSeq` object. All inserted `Interval` objects are stored in the beginning of the array. The remaining cells of the array are `null`.
Augment class `IntervalSeq` with JML specification stating the following:

- The `size` field is never negative, and always $\leq$ `contents.length`.
- The `contents` of the array which are stored below index `size` are never `null`.
- If the `size` is strictly smaller than `contents.length`, then all of the following must hold:
  - `insert` terminates normally
  - `insert` increases `size` by one
  - After `insert(iv)`, the interval `iv` is stored in `contents` at some index `i` below `size`. Below index `i`, the array `contents` is unchanged. The elements stored in between `i` and `size` were shifted one index upwards (as compared to the old `contents`).
- If the `size` has reached `contents.length`, `insert` will throw an `IndexOutOfBoundsException`.

Also, add `assignable` clauses where appropriate.

Solution
(a)

```
public class Interval {

    private /*@ spec_public @*/ final int start, end;

    /*@ public invariant getEnd() >= getStart();
      @*/

    public Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public /*@ pure @*/ int getStart() {
        return start;
    }
```

```
    public /*@ pure @*/ int getEnd() {
        return end;
    }
}
```

(b) *Remark:* `nullable` *was not required.*

```
/**
 * Class to represent sequence of intervals.
 */
public class IntervalSeq {

    //@ public invariant size >= 0;

    protected /*@ spec_public @*/ int size = 0;

    //@ public invariant contents.length >= size;
    //@ public invariant (\forall int i;
    //@                            0 <= i && i < size;
    //@                            contents[i] != null);

    protected /*@ nullable spec_public @*/ Interval[] contents
                                              = new Interval[1000];

    /**
     * Insert a new element in the sequence; it is not specified
     * in which place the element will be inserted
     */
    /*@
      @ public normal_behavior
      @   requires size < contents.length;
      @   ensures size == \old(size) + 1;
      @   ensures (\exists int i;
      @              0 <= i && i < size;
      @                 contents[i] == iv
      @              && (\forall int j; 0 <= j && j < i;
      @                    contents[j] == \old(contents[j]))
      @              && (\forall int k; i < k && k < size;
      @                    contents[k] == \old(contents[k-1])));
      @  assignable contents, contents[*], size;
      @
      @ also
      @
      @ public exceptional_behavior
      @   requires contents.length == size;
      @   signals_only IndexOutOfBoundsException;
```

```
      @*/
    public void insert(Interval iv) {
        // ...
    }

    // more methods

}
```

Assignment 6 (Loop Invariants) (12p)

Consider the following program:

```
/*@public invariant
  @  (\forall int i;
  @    (\forall int j;
  @             i>=0 && j>=0 && j<=i && i<arr.length;
  @             arr[j]<=arr[i]));
  @*/

public int[] arr;

/*@public normal_behavior
  @ requires true;
  @ ensures ?
  @*/
public int f(int x) {
  int r=0;
  /*@ loop_invariant ?
    @ assignable ?
    @ decreases ?
    @*/
  while(r<arr.length && arr[r]<x) {
    r++;
  }
  return r;
}
```

(a) [1p] Explain in your own words what `f` does.

(b) [3p] Provide the postcondition for method `f`.

(c) [1p] What fields can `f` modify? Change the specification of `f` accordingly.

(d) [5p] Provide a loop invariant along with an assignable clause that would be sufficient for proving the postcondition of `f`.

(e) [2p] Provide a `decreases` clause that would be sufficient for proving termination of `f`.

Solution

```
class Loop{
```

```
  /*@ public invariant
    @   (\forall int i;
    @     (\forall int j;
    @              i>=0 && j>=0 && j<=i && i<arr.length;
    @              arr[j] <= arr[i]));
    @*/

  public int [] arr;

  /*@ public normal_behaviour
    @ requires true;
    @ ensures \result >= 0 && \result <= arr.length &&
    @         (\forall int i; i >= 0 && i < \result;
    @                         arr[i] < x) &&
    @         (\forall int j; j >= \result && j < arr.length;
    @                         arr[j] >= x);
    @*/
  public int f(int x) {
    int r = 0;
    /*@ loop_invariant
      @   r >= 0 && r <= arr.length &&
      @   (\forall int i; i >= 0 && i < r; arr[i] < x);
      @ assignable r;
      @ decreases arr.length - r;
      @*/
    while (r < arr.length && arr[r] < x) {
      r++;
    }
    return r;
  }
}
```

(total 60p)