Testexam
Software Engineering using Formal Methods

The reasonable person adapts him-/herself to the world; the unreasonable one persists in trying to adapt the world to him-/herself. Therefore all progress depends upon the unreasonable (wo-)man.

—George Bernard Shaw

Assignment 1 (Propositional Logic) (5p)

Indicate for each of the following propositional formula if it is valid, contradictory (i.e., not satisfied by any interpretation) or neither.

(a) $p \vee \neg p \vee q$

(b) $(p \wedge \neg q) \vee \neg(p \wedge q)$

(c) $\neg(\neg p \vee q)$

(d) $((p \wedge \neg p) \vee q) \wedge \neg((p \wedge \neg p) \vee q))$

(e) $(p \wedge q) \vee (\neg p \wedge q)$

You get one point for each correct answer and one point deducted for each wrong answer.

Solution
[1p, 1p, 1p, 1p, 1p]
(a) Valid
(b) Neither
(c) Neither
(d) Contradiction
(e) Neither

Assignment 2 (First-Order Logic) (4p)

We work in untyped first-order logic, i.e., all terms have the trivial type *any* which is omitted to improve readability. In your solution you can omit it as well.

The signature consists of two unary predicate symbols p, q.

Your task is to prove that the following sequent is valid using the sequent calculus:

$$(\forall x; \exists y; (p(x) \to q(y))) \to ((\exists r; p(r)) \to (\exists t; q(t)))$$

Provide the name of each rule used in your proof. For the quantifier rules, justify that the side conditions on substituted terms are fulfilled.

To save space and time you can introduce abbreviations for some formulas.

Solution
Apply rules 2 times impRight, then exLeft (introducing skolem constant c) use then c to instantiate the universal quantifier on the left, then exLeft (introducing skolem constant d) and impLeft, close one goal immediately and the second one after instantiating the existential quantifier on the right side using d then close the proof.

Assignment 3 (PROMELA) (10p)

In this problem you are given an implementation of a simple repository server. Your goal is to write a complementary implementation of a client that asserts correctness of the server as specified below.

Server and channels The server provides two basic operations for the clients. A client may either

- *commit* new data into the server's repository, or
- *update* the client's local version from the repository

The server accepts messages on two respective channels — `commit` and `update`. For simplicity, we model possible content with mtype values `msgX` and `msgY`. The state of the repository is maintained in two variables. Variable `current` stores the latest committed value and variable `version` keeps track of the repository version number.

Channel `commit` carries two parameters. The first is an mtype for the payload, and the second is a *callback* channel. The callback channel provides the client with the current version number in the repository as well as re-confirms the submitted content. Channel `update` has only one argument which is the callback channel for returning the latest version number and the value stored in the repository.

Client Your task is to write the client code according to the following requirements.

- First, every client randomly chooses between `msgX` and `msgY` as the content it is going to commit to the repository. Once it is committed the client waits for the confirmation on its callback channel. After receiving the confirmation it *asserts* that the confirmed content matches what it has submitted when committing.
- Next, the client performs one update and subsequently awaits for the new updated version number and data on its callback channel. The client performs two assertions depending on the version number:
  - if the version number is exactly the same as the one that the client has received after the commit, then the client *asserts* that the content from the update matches the content it has committed itself.
  - otherwise, the client *asserts* that the new version number is strictly higher than the version it has committed.

Note that our initialization code creates several instances of clients, so we assume that clients may interleave in their execution. Also we provide just the header for the client `proctype` so you will need to specify your local variables as well.

```
mtype { msgX, msgY}          /* the two possible content values */

chan commit = [0] of {mtype, chan}   /* channel declarations   */
chan update = [0] of {chan }

active proctype repository() {       /* server implementation  */
  mtype current;
  byte version = 0;

  chan cb = [0] of { byte, mtype }   /* temporary channel variable
                                        to store callback channels */
end:
  do
    :: commit ? (current, cb)             /* handling commits  */
               -> version ++;
                  cb ! version, current
    :: update ? (cb)                      /* handling updates  */
               -> cb ! version, current
  od
}

proctype client() {
  /* your implementation here */
}

init {              /* initialization code - start several clients */
   run client();
   run client();
   run client();
}
```

Solution

[10p]

```
proctype client() {
   mtype content, content2;
   byte  version, version2;
   chan cb = [0] of {byte, mtype};
   if
       :: content = msgX
       :: content = msgY
   fi;
   commit ! content, cb;
   cb ?   (version, content2 );

   assert (content == content2);
   update ! cb;
   cb ?   (version2, content2);
   if
       :: version == version2 -> skip; assert (content == content2)
       :: else -> assert (version2 > version)
   fi
}
```

Assignment 4 (Temporal Logic) (10p)

Consider the following PROMELA model:

```
byte flag = 0;
byte count = 0;
bool end = false

active proctype P() {
  if
    :: flag = 1
    :: flag = 2
  fi;
  if
    :: flag == 1 ->
       do
         :: (count == 0) -> break
         :: count++
         :: count--
       od
    :: else ->
       do
         :: (count == 0) -> break
         :: (count != 0) ->
            if
              :: count++
              :: count--
            fi
       od
  fi;
  end = true
}
```

Take your time to understand the behavior of `P`. (Note that `byte` is an unsigned 8-bit type, where 255 + 1 is `0`.) Then consider the following properties, each of which *might or might not* hold:

1. `end` will be **true** at some point.
2. Whenever `flag` is 1, `end` will become **true** at some point thereafter.
3. Whenever `flag` is 2, `end` will become **true** at some point thereafter.
4. If `count` is infinitely often 5, then it is infinitely often 4 or infinitely often 6.
5. `count` can only finitely often be 0.

(a) Formulate each of the properties 1. - 5. in Temporal Logic.

(b) For each of the properties 1. - 5., 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
[6p, 4p]
(a)

1. `<>end`
2. `[]((flag == 1) -> <>end)`
3. `[]((flag == 2) -> <>end)`
4. `[]<>(count == 5) -> ([]<>(count == 4) || []<>(count == 6))`
5. `![]<>(count == 0)`

(b)

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

Assignment 5 (Modeling with JML) (10p)

The scenario we consider here is inspired by the game Sudoku. (If you don't know Sudoku, that does not matter, as the description here is self contained.)

Our version of Sudoku is a table (two dimensional array) of parametric `size` (given in the constructor). An entry in the table is considered (still) empty if it has the value 0.

For instance, a Sudoku of `size` 4 will initially look like the leftmost table below. During the course of the game, numbers in the range of `[1...size]` are filled in, one by one, *with the restriction that each number (other than 0) can appear only once per line, and only once per column.* (Unlike in real Sudoku, we have no such restrictions on the four quarters of the table.)

After a sequence of legal `put` operations, we might arrive for instance at the table depicted in the middle. The game is won if all entries are filled with numbers (other than 0), like in the table to the right (and all operations since the start of the game have obeyed the restriction mentioned above).

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

0	2	3	0
0	3	0	1
3	4	1	0
0	0	2	3

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

The game is implemented in the class `Sudoku`, the relevant part of which is given at the next page.

(Please turn.)

```
public class Sudoku {

    private final int size;

    private int[][] s;

    public Sudoku(int size) {
        this.size = size;
        s = new int[size][size];
    }

    /**
     * Check whether entering a the number num at
     * position (x,y) would be legal in the current
     * state of s.
     * Return true if yes, and false otherwise.
     */
    private boolean legal(int x, int y, int num) {
        // ...
    }

    /**
     * Assumes that entering num at (x,y) is legal
     * in the current state of s.
     * Enters num at (x,y) and leaves the remaining table
     * unchanged.
     * Returns true if the Sudoku game is won after
     * the operation, and false otherwise.
     */
    private boolean put(int x, int y, int num) {
        // ...
    }

    // other methods
}
```

(We left out other methods, in particular `public` methods to be used by user interface.)

Your task is to rewrite the class such that it includes JML specifications for the methods `legal` and `put`. Your specification should cover all aspects described in the above text and Java comments. (You are not asked to implement any method.) If you use additional methods in the specification, those need to be specified as well.

Remark: You can assume that `s` is only ever modified through legal `put` operations.

Solution
[10p]

In the following, we present two alternative solutions.

Alternative A

```
public class Sudoku {

    private final /*@ spec_public @*/ int size;
    private /*@ spec_public @*/ int[][] s;

    public Sudoku(int size) {
        this.size = size;
        s = new int[size][size];
    }

    /*@ public normal_behavior
      @   ensures
      @     \result
      @     ==
      @     (   0 <= x && x < size
      @      && 0 <= y && y < size
      @      && 0 < num && num <= size
      @      && s[x][y] == 0
      @      && (\forall int i; 0<=i && i<size;
      @                          s[i][y] != num && s[x][i] != num));
      @*/
    private /*@ spec_public pure @*/
            boolean legal(int x, int y, int num) {
        // ...
    }

    /*@ public normal_behavior
      @   requires legal(x, y, num);
      @   ensures s[x][y] == num;
      @   ensures
      @     \result
      @     ==
      @     (\forall int i,j; 0<=i && i<size && 0<=j && j<size;
      @                       s[i][j] != 0);
      @   assignable s[x][y];
     @*/
    private /*@ spec_public @*/ boolean put(int x, int y, int num) {
        // ...
    }

    // other methods, in particular public ones for the user.
}
```

Alternative B

```
public class Sudoku {

    private final /*@ spec_public @*/ int size;
    private /*@ spec_public @*/ int[][] s;

    public Sudoku(int size) {
        // ... as before
    }

    /*@ public normal_behavior
      @   requires 0 <= x && x < size;
      @   requires 0 <= y && y < size;
      @   requires 0 < num && num <= size;
      @   ensures
      @     \result
      @     ==
      @     (   s[x][y] == 0
      @      && (\forall int i; 0<=i && i<size;
      @                          s[i][y] != num && s[x][i] != num));
      @*/
    private /*@ spec_public pure @*/
            boolean legal(int x, int y, int num) {
        // ...
    }

    /*@ public normal_behavior
      @   requires 0 <= x && x < size;
      @   requires 0 <= y && y < size;
      @   requires 0 < num && num <= size;
      @   requires legal(x, y, num);
      @   ensures s[x][y] == num;
      @   ensures
      @     \result
      @     ==
      @     (\forall int i,j; 0<=i && i<size && 0<=j && j<size;
      @                       s[i][j] != 0);
      @   assignable s[x][y];
      @*/
    private /*@ spec_public @*/ boolean put(int x, int y, int num) {
        // ...
    }

    // other methods, in particular public ones for the user.
}
```

Assignment 6 (Specifying Exceptional Behavior) (7p)

Consider the following JML contract for method `arraySwap()`:

```
/*@ public normal_behavior
  @  requires a!=b && a.length >= 0 && a.length <= b.length;
  @  ensures
  @   (\forall int j; 0<=j && j<a.length; b[a.length-1-j]==\old(a[j]));
  @  assignable b[*];
  @*/
  public static void arraySwap(int[] a, int[] b) {
    // ...
  }
```

Let us now say that the method should throw an `IllegalArgumentException` in all cases which are not covered by the 'normal behavior' requires clause. Moreover, if an `IllegalArgumentException` is thrown, neither the arrays `a` and `b` nor their contents should change.

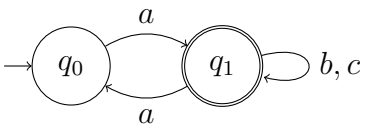
Your task is to extend the JML *contract* of `arraySwap` accordingly. For that, write the full contract (not only the extension). In order to increase the level of difficulty a little bit, the usage of the `assignable` clause is disallowed in the new part of the contract.

Solution

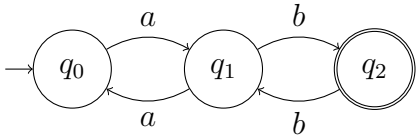
```
/*@ public normal_behavior
  @  requires a!=b && a.length >= 0 && a.length <= b.length;
  @  ensures
  @   (\forall int j; 0<=j && j<a.length; b[a.length-1-j]==\old(a[j]));
  @  assignable b[*];
  @
  @ also
  @
  @ public exceptional_behavior
  @  requires a==b || a.length < 0 || a.length > b.length;
  @  signals_only IllegalArgumentException;
  @  signals (IllegalArgumentException)
  @      a==\old(a)
  @   && b==\old(b)
  @   && (\forall int i; 0 <= i && i<a.length; a[i]==\old(a[i]))
  @   && (\forall int i; 0 <= i && i<b.length; b[i]==\old(b[i]));
  @*/
```

Assignment 7 (Buechi Automata and LTL) (8p)

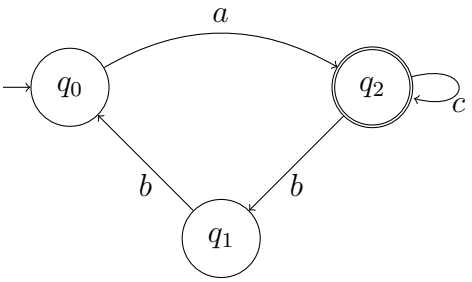
(a) Give the omega expression representing exactly the language recognised by the *Buechi* automaton below.



(b) Give the omega expression representing exactly the language recognised by the *Buechi* automaton below.



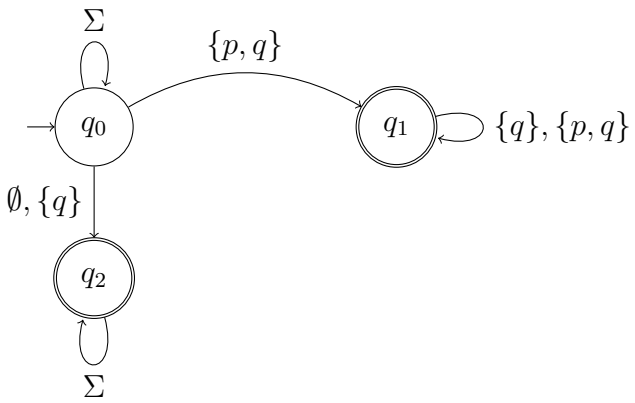
(c) Give the omega expression representing exactly the language recognised by the *Buechi* automaton below.



(d) Give a Buechi automaton that accepts exactly those runs satisfying the LTL formula:

$$\Diamond(p \rightarrow \Box q)$$

Solution
(a) $a(aa + b + c)^\omega$
(b) $a((aa)^*bb)^\omega$
(c) $a(bba + c)^\omega$
(d) $\Sigma := 2^{\{p,q\}}$



Assignment 8 (Model Checking) (6p)

Fill in the holes in the text below.

Given a transition system $\mathcal{T}$ (e.g., a Promela program) and an LTL-formula ϕ which expresses a desired property over $\mathcal{T}$.

To model check whether ϕ is satisfied in all possible runs of $\mathcal{T}$ (i.e. $\mathcal{T} \models \phi$) the following steps have to be performed:

Starting from the transition system $\mathcal{T}$ a Buechi automaton B_1 is constructed that accepts _____. Then a Buechi automaton B_2 is constructed accepting all runs satisfying the _____ formula ϕ.

Finally, the _____ automaton B_3 of B_1 and B_2 is constructed. If the language recognised by B_3 is empty then formula ϕ is _____ in any possible run of $\mathcal{T}$. Otherwise any run accepted by B_3 is a _____ for ϕ.

To test if the language recognised by B_3 is empty or not, the automaton is analysed to determine whether an accepting location can be reached from an initial one and whether that accepting location lies on a _____.

Solution
Given a transition system $\mathcal{T}$ (e.g., a Promela program) and an LTL-formula ϕ which expresses a desired property over $\mathcal{T}$.

To model check whether ϕ is satisfied in all possible runs of $\mathcal{T}$ (i.e. $\mathcal{T} \models \phi$) the following steps have to be performed:

Starting from the transition system $\mathcal{T}$ a Buechi automaton B_1 is constructed that accepts exactly those runs that are possible. Then a Buechi automaton B_2 is constructed accepting all runs satisfying the negated formula ϕ.

Finally, the intersection automaton B_3 of B_1 and B_2 is constructed. If the language recognised by B_3 is empty then formula ϕ is satisfied in any possible run of $\mathcal{T}$. Otherwise any run accepted by B_3 is a counterexample for ϕ.

To test if the language recognised by B_3 is empty or not, the automaton is analysed to determine whether an accepting location can be reached from an initial one and whether that accepting location lies on a cycle.

(total 60p)