TRIAL-EXAM
Software Engineering using Formal Methods
TDA293 (TDA292) / DIT270

Extra aid: Only dictionaries may be used. Other aids are *not* allowed!

Please observe the following:

- This exam has 15 numbered pages, plus two pages of the Spin Reference Card.
  Please check immediately that your copy is complete
- Answers must be given in English
- Use page numbering on your pages
- Start every assignment on a fresh page
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions
- Indicate clearly when you make assumptions that are not given in the assignment

Good luck!

Assignment 1 PROMELA (13p)

The task is to model a simplified version of the TicTacToe game in PROMELA. TicTacToe is a two player game which is played on a 3×3 board as shown below:

0	1	2
3	4	5
6	7	8

(The numbers suggest how to represent the board by a simple array of size 9.)

The game is played turn-by-turn. The player whose turn it is, selects an empty field and marks it with her unique id.

We use here a simplified winning condition: the player who manages to mark one of the two diagonals completely (i.e., all fields belonging to that diagonal) with her unique id wins the game.

The game is over if either one of the players has won or if there is no empty field left which belongs to one of the diagonals and none of the diagonals is owned by one of the two players. In the latter case, we call the game to be a draw.

(a) Modeling TicTacToe in PROMELA.

Below you find a skeleton model of a TicTacToe game modelled in PROMELA:

```
/* message type */
mtype = {move , done}
/* board */
byte board[9];

/* channel to communicate with referee. The second parameter holds the
   player's process id */
chan ch = [0] of {mtype, byte};

/* variable that is 0 if game is ongoing, _pid if player _pid has won,
   255 if game is a draw */
byte over = 0;

proctype player() { /* to be filled in */  }

proctype referee() { /* to be filled in */ }

/* Initialises and starts the players and referee. Ensures that the
 * first player has _pid 1 and the second player has _pid 2 */
init {
  run player();
  run player();
  run referee()
}
```

The PROMELA model realizes the playfield as a global variable called `board`, a byte array of size 9. Initially all components of the board have the value 0.

The message type **mtype** is defined to consist of the two values `move` and `done`. Further, there are two global variables called (i) `ch` of type **chan** which is used for the communication between the referee and the players; (ii) `over` of type byte which indicates whether the game is still ongoing (`over` is 0), or one player has won (`over` carries that player's process id), or the game is a draw (`over` has is 255).

The process **init** initializes the game with two players and one referee.

Your task is now to provide an implementation for the player and referee process according to the following description:

Player: For as long as `over` is 0, the player does the following over and over again:

1. Wait for the referee to inform her that it is her move (by sending a message containing `move` and this player's process id).
2. Select *non-deterministically* a free field (a field is free if its value is 0).
3. Store her process id in that field.
4. Informs the referee that this turn is completed, by sending the message `done` together with her process id.

Afterwards, the player prints "Game Over!" and terminates.

Referee:

1. Initially the referee starts the game by giving the turn to the first player. A turn is assigned to a player by sending the message `move` followed by the player's process id via the channel `ch`.
2. The referee then waits for the player to finish her turn.
3. Upon receiving that the player has finished the turn, the referee checks if *this* player has won the game (i.e., if the player has marked one of the two diagonals completely with her process id). If that is the case, the referee sets the global variable `over` to the player's id and the referee process terminates. Otherwise the referee continues with step 4.
4. The referee checks now if it is a draw, if that is *not* the case, it hands the turn over to the other player by sending the `move` message followed by the corresponding process id of the player. The referee continues then with step 2.
5. Otherwise, it is a draw. The variable `over` is set to 255 and the referee process terminates.

Solution

Remark to correctors: Check if `idx` is reset to zero each time before a new field is chosen.

```
mtype = {move, done}

byte board[9];

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

byte over = 0;

proctype player() {

  byte idx;

  do
    :: over != 0 -> break
    :: ch ? move, eval(_pid);
       /* choose field non-deterministically */
       idx = 0;
       do
         :: idx < 8 -> idx++
         :: break
       od;
       /* set player's mark at next free field*/
       do
         :: board[idx] == 0
            ->
            board[idx] = _pid; ch ! done, _pid; break
         :: else
            ->
            idx = (idx + 1) % 9
       od
  od;

  printf("Player␣%d␣says␣good-bye", _pid)
}


proctype referee() {

  byte playerId;

  ch ! move, 1;

  do
    :: ch ? done, playerId;
       if
         :: board[4] != 0 &&
            ((board[0] == board[4] && board[4] == board[8]) ||
             (board[2] == board[4] && board[4] == board[6]))
            ->
            over = board[4];
            printf("Player␣%d␣won.", board[4]); break
         :: else
            ->
            if
              :: (board[0] == 0 || board[2] == 0 || board[4] == 0 ||
                  board[6] == 0 || board[8] == 0)
                 ->
                 ch ! move, (playerId == 1 -> 2 : 1);
                 printf("Next␣move␣by␣player␣%d",
                        (playerId == 1 -> 2 : 1))
              :: else
                 ->
                 over = 255;
                 printf("Draw.");
                 break
            fi
       fi
  od;

  printf("Game␣Over!")
}

init {
  run player();
  run player();
  run referee()
}
```

Assignment 2 Linear Temporal Logic (LTL) (8p)

Tell for each formula whether it is valid or not.
(Each question is worth 1 point, except question 7., which is worth 2 points. Missing answers get no points, wrong answers get -1 point. Still, you will never get a negative total for this assignment.)

1. $\Box p \rightarrow \Diamond p$
2. $\Diamond p \rightarrow \Box p$
3. $\Box\Diamond p \rightarrow \Diamond\Box p$
4. $\Diamond\Box p \rightarrow \Box\Diamond p$
5. $\Diamond(p \vee q) \rightarrow \neg\Box(\neg p \vee \neg q)$
6. $\Box\neg\Diamond p \rightarrow \neg p$
7. $(\Box p \vee \Diamond q) \rightarrow (\Box\neg q \rightarrow \Diamond p)$

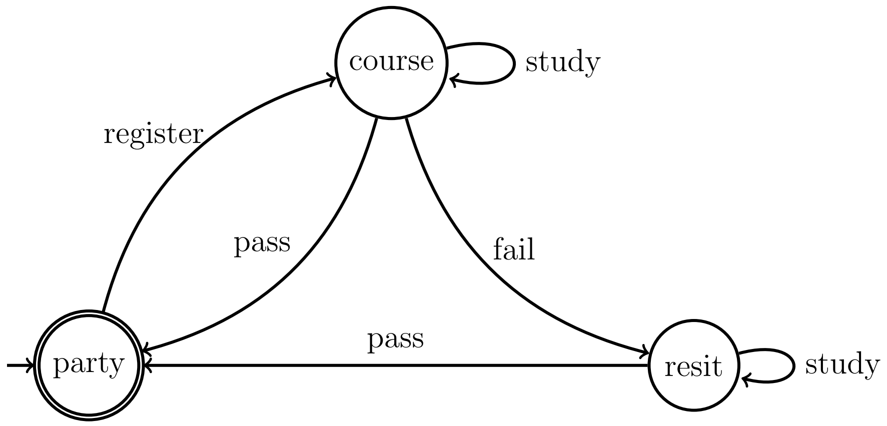
Solution

[1+1+1+1+1+1+2]

1. *valid*
2. *not valid*
3. *not valid*
4. *valid*
5. *not valid*
6. *valid*
7. *valid*

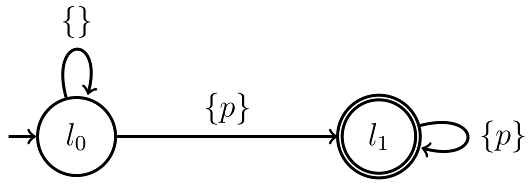
Assignment 3 (Büchi Automata and Model Checking) (9p)

(a) [3p]
Consider the following Büchi automaton.



Give the ω-expression describing the language accepted by this automaton.
Solution
(register (study)* (pass + (fail (study)* pass))$^\omega$

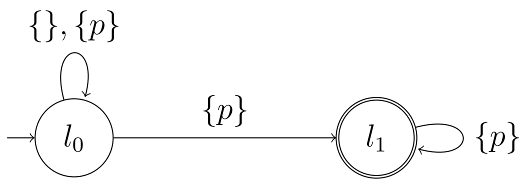
(b) [2p]
The following Büchi automaton does *not* exactly accept the runs that satisfy the LTL formula $\Diamond\Box p$:



Demonstrate this mismatch, by giving a run which is accepted by the automaton and not by the LTL formula, or vice-versa.
Solution
The Büchi automaton rejects the following run, though it satisfies the LTL formula: $\{p\}, \{\}, \{p\}, \{p\}, \{p\}, \{p\}, \ldots$

(c) [2p]
Give an LTL formula that is satisfied exactly by accepted runs of the automaton from (b).
Solution
$\neg p \mathcal{U} \Box p$

(d) [2p]
The Büchi automaton from (b) requires just a small modification to accept exactly those runs satisfying $\Diamond\Box p$. Give this modified automaton.
Solution



Assignment 4 (First-Order Sequent Calculus) (10p)

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 first-order logic formula is valid using the *first-order sequent calculus*. For each step name the rule you have applied and for the quantification rules also the side-conditions like substitution or introduction.

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

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

$$\begin{array}{l}\forall x; \forall y; (p(x,y) \rightarrow \neg p(y,x)),\\ \forall x; \forall y; \forall z; ((p(x,y) \wedge p(y,z)) \rightarrow p(x,z)),\\ p(c,d) \Rightarrow\\ \forall z; (p(d,z) \rightarrow \neg p(z,c))\end{array}$$

Hint: You may abbreviate formulas, but only if you clearly describe your abbreviations.

Solution

1. apply allRight to succedent formula introduce new constant z_0
2. instantiate transitivity formula in antecedent by successive application of 3 allLeft rules instantiating with d, z_0, c subsequently
3. impRight on formula in succedent
4. notRight on formula in succedent
5. impLeft on formula in antecedent
6. two subgoals
   (a) first subgoal: apply andRight on formula in succedent; the two created subgoal can be closed immediately by applications of the close rule
   (b) second subgoal: instantiate antisymmetry formula by applying allLeft twice subsequently with d, c. Apply impLeft: Two goals opened: first apply close rule, second: apply notLeft followed by close

Assignment 5 (Java Modeling Language) (12p)

Consider the class `Hashtable`:

```
public class Hashtable {

  private Object[] h;
  private int capacity;
  private int size = 0;

  Hashtable (int capacity) {
     h = new Object[capacity];
     this.capacity = capacity;
  }

  /*@ public normal_behavior
    @ requires val > 0;
    @ ensures \result >= 0 && \result =< capacity;
    @*/
  private int hash_function (int val) { ..... }

  public void add (Object obj, int key) {
     if (size < capacity) {
        int i = hash_function(key);
        if (h[i] == null) {
           h[i] = obj;
           size++;
        }
        else {
           while (h[i] != null){
                 if (i == capacity-1) {i = 0;} else {i++;}
           }
           h[i] = obj;
           size++;
        }
        return;
    } else
        throw new FullHashtableException();
  }
}
```

This class represents an open addressing hash table with linear probing as collision resolution. Within a hash table, objects are stored into a fixed array `h`. Besides, in order to have an easy way of checking whether or not the capacity of `h` is reached (i.e. the array `h` is full), a field `size` keeps track of the number of stored objects and a field `capacity` represents the total amount of objects that can be added to the hash table.

The method `add`, which is used to add objects to the hash table, first tries to put the corresponding object at the position of the computed hash code. However, if that index is occupied, then `add` searches upwards (modulo the array length) for the nearest following index which is free. A position is considered free if and only if it contains a **`null`** object.

Augment class `Hashtable` with JML specification stating the following:

- The `size` field is never negative, and always ≤ `capacity`.
- The `capacity` should be the same value as `h.length`.
- The array `h` cannot be `null`.
- There should be space for at least one element in the hash table.
- The number of elements stored in array `h` (i.e., the number of array cells whose content is not **`null`**) is `size`.
- If the `size` is strictly smaller than `capacity`, then all of the following must hold:
  - `add` terminates normally
  - `add` increases `size` by one
  - After `add(obj,key)`, the object `obj` is stored in `h` at some index `i`.
- If the `size` has reached `capacity`, `add` will throw an `FullHashtableException`, and the state does not change.

In addition:

- Write `assignable` clauses where appropriate.
- Add JML modifiers where necessary.

Solution

```
public class Hashtable {

// Open addressing Hashtable with linear probing
// as collision resolution.

/*@ invariant h != null;
  @
  @ invariant h.length == capacity;
  @
  @ invariant capacity > 0;
  @
  @ invariant size >= 0 && size <= capacity;
  @
  @ invariant
  @ (\sum int i; 0 <= i && i < capacity && h[i] != null; 1) == size;
  @*/
  private /*@ spec_public nullable @*/ Object[] h;
  private /*@ spec_public @*/ int capacity;
  private /*@ spec_public @*/ int size = 0;

// Specifying the constructor was not required
/*@ public normal_behaviour
  @ ensures this.capacity == capacity;
  @ ensures size == 0;
  @ assignable h[*], capacity, size;
  @*/
  Hashtable (int capacity) {
     h = new Object[capacity];
     this.capacity = capacity;
  }

/*@ public normal_behaviour
  @ requires val > 0;
  @ ensures \result >= 0 && \result < capacity;
  @*/
  private /*@ pure @*/ int hash_function (int val) {
     .....
  }

// Add an element to the hashtable.
/*@ public normal_behaviour
  @ requires size < capacity && key > 0;
  @ ensures size == \old(size)+1;
  @ ensures (\exists int i; i>= 0 && i < capacity; h[i] == obj);
  @ assignable size, h[*];
  @
  @ also
  @
  @ public exceptional_behaviour
  @ requires size == capacity && key > 0;
  @ signals_only FullHashtableException;
  @ assignable \nothing;
  @*/
  public void add (Object obj, int key) {
     if (size < capacity) {
        int i = hash_function(key);

        if (h[i] == null) {
           h[i] = obj;
           size++;
        }
        else {
           while (h[i] != null){
                 if (i == capacity-1) {i = 0;} else {i++;}
           }
           h[i] = obj;
           size++;
         }
         return;
    } else
         throw new FullHashtableException();
  }
}
```

Assignment 6 (Loop-Invariant) (4p)

Consider the following `Arrays` class with utility methods for copying portions of **`int`** arrays from some source array `src` to some destination array `dest`. The top level method `arrayCopy` is capable of copying data even within the same array (when `src == dest`) in which case the data is first copied from the source region to a temporary array and then from the temporary array to the destination region. This way data corruption is avoided when the two regions overlap. The private helper method `arrayCopyHelper` works under the assumption that the provided arrays are non-null and different by reference, and that all other data (offsets and number of elements to be copied) is well-defined. This is reflected in the method's precondition. *Note* that the postcondition specification for `arrayCopyHelper` does not use the **`\old`** operator, because the source array will never be overwritten by this method. This is not the case for the top-level method `arrayCopy`, where the `src` array may change after the method is finished (when `src == dest`).

```
public class Arrays {

  private static /*@ spec_public @*/ IllegalArgumentException iae =
    new IllegalArgumentException();

  /*@ public normal_behavior
    @   requires numElems >= 0;
    @   requires src != null && dest != null;
    @   requires srcOffset >= 0 && destOffset >= 0;
    @   requires srcOffset + numElems <= src.length;
    @   requires destOffset + numElems <= dest.length;
    @   ensures (\forall int i; i >= 0 && i < numElems;
    @     dest[destOffset + i] == \old(src[srcOffset + i]));
    @   assignable dest[*];
    @
    @ also
    @
    @ public exceptional_behavior
    @   requires src == null || dest == null || numElems < 0 ||
    @     srcOffset < 0 || destOffset < 0 ||
    @     srcOffset + numElems > src.length ||
    @     destOffset + numElems > dest.length;
    @   signals_only IllegalArgumentException;
    @   assignable \nothing;
    @*/
  public void arrayCopy(/*@ nullable @*/ int[] src, int srcOffset,
                        /*@ nullable @*/ int[] dest, int destOffset,
                        int numElems) throws IllegalArgumentException {
    if(src == null || dest == null || numElems < 0 ||
       srcOffset < 0 || destOffset < 0 ||
       srcOffset + numElems > src.length ||
       destOffset + numElems > dest.length) {
      throw iae;
    }
    if(src == dest) {
      int[] temp = new int[numElems];
      arrayCopyHelper(src, srcOffset, temp, 0, numElems);
      arrayCopyHelper(temp, 0, dest, destOffset, numElems);
    } else {
      arrayCopyHelper(src, srcOffset, dest, destOffset, numElems);
    }
  }

  /*@
    @ public normal_behavior
    @   requires numElems >= 0;
    @   requires src != dest;
    @   requires srcOffset >= 0 && destOffset >= 0;
    @   requires srcOffset + numElems <= src.length;
    @   requires destOffset + numElems <= dest.length;
    @   ensures (\forall int i; i >= 0 && i < numElems;
    @     dest[destOffset + i] == src[srcOffset + i]);
    @   assignable dest[*];
    @*/
  private void arrayCopyHelper(int[] src, int srcOffset,
                               int[] dest, int destOffset,
                               int numElems) {
    int i = 0;
    while(i < numElems) {
      dest[destOffset + i] = src[srcOffset + i];
      i++;
    }
  }
}
```

Assignments:

- Provide a strong enough loop invariant, variant (**`decreases`** clause), and **`assignable`** clause for the loop in the `arrayCopyHelper` method, so that the postcondition of this method is provable.

  Solution *4pt*

  ```
      /*@ loop_invariant i >= 0 && i <= numElems &&
        @    (\forall int j; j >= 0 && j < i;
        @         dest[destOffset + j] == src[srcOffset + j]);
        @ decreases numElems - i;
        @ assignable i, dest[*];
        @*/
  ```

Liveness:

```
spin -a file
gcc -o pan pan.c
pan -a -f or ./pan -a -f
spin -t -p -l -g -r -s file
```

Spin arguments

`-a`	generate verifier and syntax check
`-i`	interactive simulation
`-I`	display Promela program after preprocessing
`-nN`	seed for random simulation
`-t`	guided simulation with trail
`-tN`	guided simulation with Nth trail
`-uN`	maximum number of steps is N
`-f`	translate an LTL formula into a never claim
`-F`	translate an LTL formula in a file into a never claim
`-N`	include never claim from a file
`-l`	display local variables
`-g`	display global variables
`-p`	display statements
`-r`	display receive events
`-s`	display send events

Compile arguments

`-DBFS`	breadth-first search
`-DNP`	enable detection of non-progress cycles
`-DSAFETY`	optimize for safety
`-DBITSTATE`	bitstate hashing
`-DCOLLAPSE`	collapse compression
`-DHC`	hash-compact compression
`-DMA=n`	minimized DFA with maximum n bytes
`-DMEMLIM=N`	use up to N megabytes of memory

Pan arguments

`-a`	find acceptance cycles
`-f`	weak fairness
`-l`	find non-progress cycles
`-cN`	stop after Nth error
`-c0`	report all errors
`-e`	create trails for all errors
`-i`	search for shortest path to error
`-I`	approximate search for shortest path to error
`-mN`	maximum search depth is N
`-wN`	2^N hash table entries
`-A`	suppress reporting of assertion violations
`-E`	suppress reporting of invalid end states

Caveats

- Expessions must be side-effect free.
- Local variable declarations always take effect at the beginning of a process.
- A `true` guard can always be selected; an `else` guard is selected only if all others are false.
- Macros and `inline` do *not* create a new scope.
- Place labels before an `if` or `do`, *not* before a guard.
- In an `if` or `do` statement, interleaving can occur between a guard and the following statement.
- Processes are activated and die in LIFO order.
- Atomic propositions in LTL formulas must be identifiers starting with lowerase letters and must be boolean variables or symbols for boolean-valued expressions.
- Arrays of `bit` or `bool` are stored in bytes.
- The type of a message field of a channel cannot be an array; it can be a `typedef` that contains an array.
- The functions `empty` and `full` cannot be negated.

References

- G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004. `http://spinroot.com`.
- M. Ben-Ari. *Principles of the Spin Model Checker*, Springer, 2008. `http://www.springer.com/978-1-84628-769-5`.

Spin Reference Card

Mordechai (Moti) Ben-Ari

October 1, 2007

Datatypes

`bit` (1 bit)
`bool` (1 bit)
`byte` (8 bits unsigned)
`short` (16^* bits signed)
`int` (32^* bits signed)
`unsigned` ($\leq 32^*$ bits unsigned)
* - for a 32-bit machine.
`pid`
`chan`
`mtype` = { name, name, ... } (8 bits)
`typedef` typename { sequence of declarations }

Declaration - type var [= initial value]
Default initial values are zero.
Array declaration - type var[N] [= initial value]
Array initial value assigned to all elements.

Operators (descending precedence)

```
()   []   .
!    ~    ++   --
*    /    %
+    -
<<   >>
<    <=   >    >=
==   !=
&
^
```

|
&&
||
(... -> ... : ...) conditional expression
=

Predefined
Constants - true, false
Variables (read-only except _):
  _ - write-only hidden scratch variable
  _nr_pr - number of processes
  _pid - instantiation number of executing process
  timeout - no executable statements in the system?

Preprocessor
#define name (arguments) string
#undef, #if, #ifdef, #ifndef, #else, #endif
#include "file name"
inline name (arguments) { ... }

Statements
Assignment - var = expression, var++, var--
assert(expression)

printf, printm - print to standard output
  %c (character), %d (decimal), %e (mtype),
  %o (octal), %u (unsigned), %x (hex)
scanf - read from standard input in simulation mode

skip - no operation
break - exit from innermost do loop
goto - jump to label
Label prefixes with a special meaning:
  accept - accept cycle
  end - valid end state
  progress - non-progress cycle

atomic { ... } - execute without interleaving
d_step { ... } - execute deterministically (no jumping in or out; deterministic choice among true guards; only the first statement can block).

{ ... } unless { ... } - exception handling.

Guarded commands
if :: guard -> statements :: ... fi
do :: guard -> statements :: ... od
else guard - executed if all others are false.

Processes
Declaration - proctype procname (parameters) { ... }
Activate with prefixes - active or active[N]
Explicit process activation - run procname (arguments)
Initial process - init { ... }
Declaration suffixes:
  priority - set simulation priority
  provided (e) - executable only if expression e is true

Channels
chan ch = [capacity] of { type, type, ... }

ch ! args	send
ch !! args	sorted send

ch ? args	receive and remove if *first* message matches
ch ?? args	receive and remove if *any* message matches
ch ? <args>	receive if *first* message matches
ch ?? <args>	receive if *any* message matches
ch ? [args]	poll *first* message (side-effect free)
ch ?? [args]	poll *any* message (side-effect free)

Matching in a receive statement: constants and mtype symbols must match; variables are assigned the values in the message; eval(expression) forces a match with the current value of the expression.

len(ch) - number of messages in a channel
empty(ch) / nempty(ch) - is channel empty / not empty?
full(ch) / nfull(ch) - is channel full / not full?

Channel use assertions:
  xr ch - channel ch is receive-only in this process
  xs ch - channel ch is send-only in this process

Temporal logic

!	not
&&	and
||	or
->	implies
<->	equivalent to
[]	always
<>	eventually
X	next
U	strong until
V	dual of U defined as pVq <-> !(!pU!q)

Remote references
Test the control state or the value of a variable:
  process-name @ label-name
  proctype-name [expression] @ label-name
  process-name : label-name
  proctype-name [expression] : label-name

Never claim
never { ... }.
Predefined constructs that can only appear in a never claim:
  _last - last process to execute
  enabled(p) - is process enabled?
  np_ - true if no process is at a progress label
  pc_value(p) - current control state of process
  remote references
See also trace and notrace.

Variable declaration prefixes
hidden - hide this variable from the system state
local - a global variable is accessed only by one process
show - track variable in Xspin message sequence charts

Verification
Safety:
```
spin -a file
gcc -DSAFETY -o pan pan.c
pan or ./pan
spin -t -p -l -g -r -s file
```