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 9 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 (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).

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:

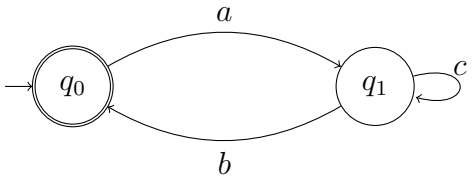
1. `b` will be **true** at some point.
2. `x` will always be ≥ 10.
3. At some point, `x` will be 10.
4. At some point, `x` will be 11.
5. From some point on, `x` will always be ≥ 10.
6. `x` will infinitely often be 11.
7. If `b` will never be **true**, then `x` will infinitely often be 11.

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

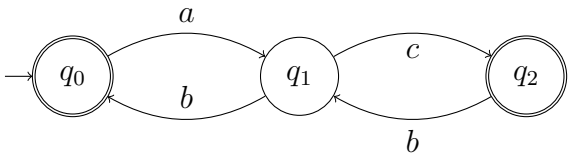
(b) [4p]
For each of the properties 1. - 7., tell whether or not the property is valid in the transition system given by the above PROMELA model. (You don't need to explain your answer.)

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

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



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



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

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

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]

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

(b) [4p]

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

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.

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`.

(total 60p)

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
```