EXAM
Software Engineering using Formal Methods
TDA293 / DIT270

Day: **28/10/2016** Time: **14:00 – 18:00**

Responsible:	Wolfgang Ahrendt Tel.: 031-7721011
Extra aid:	Only dictionaries may be used. Other aids are *not* allowed!
Grade intervals:	**U**: 0 – 23p, **3**: 24 – 35p, **4**: 36 – 47p, **5**: 48 – 60p, **G**: 24 – 47p, **VG**: 48 – 60p, **Max.** 60p.

Please observe the following:

- This exam has 14 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 your task is to write a Promela program to solve the 15 puzzle. Typically the 15 puzzle is played with a little plastic board with small pieces numbered from 1 to 15, with one slot empty. The only possible move is to slide a piece into the empty slot. The goal is to reach the final state which looks as follows:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Write a PROMELA program which nonderministically chooses a correct move on the puzzle board. After each move, an assertion should check that the final goal state is *not* fulfilled. (This way, a violating path found by SPIN will correspond to a solution of the puzzle.) The move and assertion checking should be done in an infinite loop.

Hint: You can assume that the board is initialized so that it contains all the numbers from 0 to 15. The number 0 is used to represent the free slot. You can also assume that the free slot is in position 0,0 on the board when the game starts. The other numbers, 1 to 15, you can assume to be distributed arbitrarily. Again, this initialization you can *assume*, so do not waste time writing your own code for the initialisation. Also, recall that plain 2-dimensional arrays are not supported by PROMELA. One (but not the only) possibility is that you code everything in a single dimensional field of size 16.

Solution

```
byte board[16];

active proctype solver() {
  byte xpos = 0, ypos = 0;
  do
  :: if
     :: xpos > 0 ->
        board[xpos+4*ypos] = board[(xpos-1)+4*ypos];
        board[(xpos-1)+4*ypos] = 0;
        xpos--;
     :: xpos < 3 ->
        board[xpos+4*ypos] = board[(xpos+1)+4*ypos];
        board[(xpos+1)+4*ypos] = 0;
        xpos++;
     :: ypos > 0 ->
        board[xpos+4*ypos] = board[xpos+4*(ypos-1)];
        board[xpos+4*(ypos-1)] = 0;
        ypos--;
     :: ypos < 3 ->
```

```
        board[xpos+4*ypos] = board[xpos+4*(ypos+1)];
        board[xpos+4*(ypos+1)] = 0;
        ypos++;
     fi;
     assert(!(board[0] == 1 && board[1] == 2 && board[2] == 3 &&
              board[3] == 4 && board[4] == 5 && board[5] == 6 &&
              board[6] == 7 && board[7] == 8 && board[8] == 9 &&
              board[9] == 10 && board[10] == 11 && board[11] == 12 &&
              board[12] == 13 && board[13] == 14 && board[14] == 15));
  od
}
```

Assignment 2 Linear Temporal Logic (LTL) (7p)

Are the following LTL formulas valid, unsatisfiable or neither? In case of valid or unsatisfiable, it is enough to write "valid" or "unsatisfiable", respectively. In case of neither, provide one satisfying and one non-satisfying run. The characters p and q are propositional variables.

1. $(\Diamond\Box p \wedge \Diamond\Box q) \rightarrow \Diamond\Box(p \wedge q)$
2. $(\Box\Diamond p \wedge \Box\Diamond q) \rightarrow \Box\Diamond(p \wedge q)$
3. $(\Box\Diamond p) \vee (\Box\Diamond \neg p)$
4. $(\Box(p \rightarrow \Diamond q)) \rightarrow (p \rightarrow \Box\Diamond q)$
5. $(\Box\text{false}) \rightarrow (\Diamond\text{false})$

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

1. valid
2. neither:
   - satisfying run: $\emptyset\,\emptyset\,\emptyset\ldots$
   - non-satisfying run: $\{\texttt{p}\}\{\texttt{q}\}\{\texttt{p}\}\{\texttt{q}\}\{\texttt{p}\}\{\texttt{q}\}\ldots$
3. valid
4. neither:
   - satisfying run: $\{\texttt{p}\}\,\emptyset\,\emptyset\,\emptyset\ldots$
   - non-satisfying run: $\{\texttt{p}\}\{\texttt{q}\}\,\emptyset\,\emptyset\,\emptyset\ldots$
5. valid

Assignment 3 (First-Order Sequent Calculus) (8p)

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

$\exists\, x; \forall\, y; p(y, x),$
$\forall\, x; \forall\, y; (p(x, y) \rightarrow q(y, x)),$
$\Rightarrow$
$\exists\, x; q(x, x)$

You are only allowed to use the rules presented in the lecture.

Provide the name of each rule used in your proof as well as the resulting sequent. (If you do not use a tree notation, make clear on which sequent you have applied the rule.) Every time you use a rule that has a side condition, justify that the side condition is fulfilled.

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

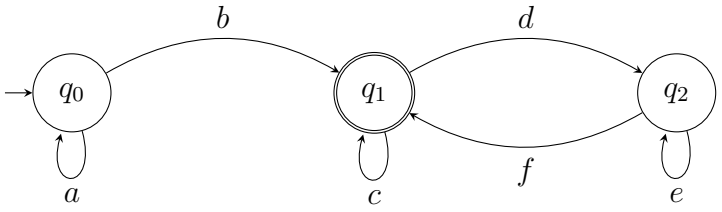
Solution
In this document, we only provide the names of the used proof rules. In the exam, you were requested to give more information, including the resulting sequents, for each proof step. See the question text.

Several solutions are possible. One is:

`exLeft` *on* $\exists\, x; \forall\, y; p(y, x)$*, with* $x \mapsto c$*, where* c *is a new constant*
`allLeft` *on* $\forall\, y; p(y, c)$*, with* $y \mapsto c$*,*
`allLeft` *on* $\forall\, x; \forall\, y; (p(x, y) \rightarrow q(y, x))$*, with* $x \mapsto c$*,*
`allLeft` *on* $\forall\, y; (p(c, y) \rightarrow q(y, c))$*, with* $y \mapsto c$*,*
`exRight` *on* $\exists\, x; q(x, x)$*, with* $x \mapsto c$*,*
`impLeft`*, get two branches*
`close`*, on both branches*

Assignment 4 (Büchi Automata and Model Checking) (9p)

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



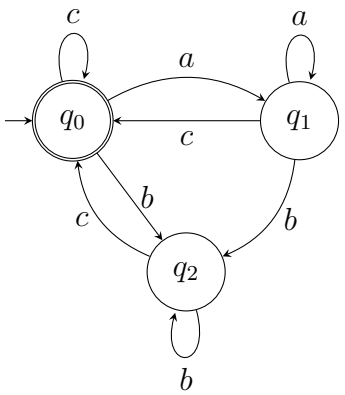
Solution

$$a^*b(c + (de^*f))^\omega$$

(b) [3p]
Give a Büchi automaton accepting exactly the ω-language given by the following ω-expression:

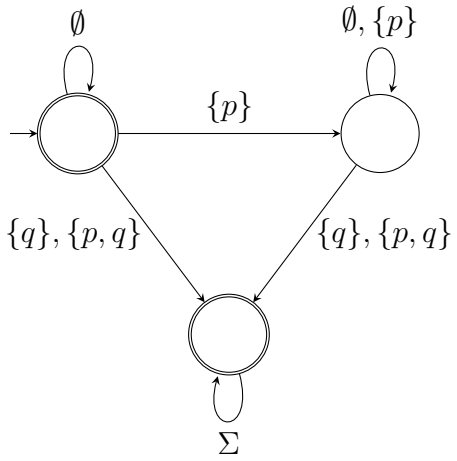
$$(a^*b^*c)^\omega$$

Solution



(c) [3p]
Give an LTL formula that is satisfied by exactly those runs which are accepted by the following Büchi automaton:

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



Solution

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

Assignment 5 (Java Modeling Language) (12p)

We consider a small part of an e-book application. Each book is represented by an instance of a class `Book`, which is sketched below (see next page). A book always has an ISBN number and a number of pages. A book has a current page and a number of bookmarks (stored in an array). A reader can flip a page, jump to a marked page, and mark the current page. In the following we describe the properties and constraints for the books:

1. No two books have the same ISBN.
2. A book has at least one page.
3. Pages are numbered from 1 to the last page, and the current page must be within this range.
4. The bookmarks are initialized to the value 0, and can be assigned to valid page numbers.
5. There is no duplicate in bookmarks (i.e., no two assigned bookmarks point to the same page).
6. A reader can try to flip a page (i.e., `flip`) to move to the next page. If he/she is already at the last page, flipping throws a `BookException`, and keeps the book (i.e., ISBN and number of pages) and the current reading state (i.e., current page and all bookmarks) unchanged.
7. A reader can jump directly to a marked page (i.e., `jump`) by providing the array index of the bookmark.
8. A reader can mark the current page (i.e., `mark`) by storing the page number in the bookmark array, without changing existing bookmarks other than the location that is used for the new one. (*Hint*: We do not care about which array location is used for the new bookmark.)

Your assignment is to specify the `Book` class using JML to reflect the above properties plus any other properties that you deem necessary. In particular, specify:

- Class invariants to reflect the stated consistency properties (*Hint:* recall the difference between static and instance invariants),
- Method specifications for the `flip`, `jump`, and `mark` methods.
- Assignable clause for the methods.

You only have to *specify* the class. No implementation (neither for the constructor nor for any methods) is expected from you. Keep the pre- and postconditions as minimal as possible. In particular, avoid stating postconditions that can be inferred from the other method conditions or the class invariant.

```
public class Book {
   private int lastPage;
   private int currentPage;
   private int isbn;
   private int[] bookmarks = new int[10];


   public Book(int numberOfPages, int isbn) {...}

   public void flip() {...}

   public void jump(int index) {...}

   public void mark() {...}
}
```

Solution

```
public class Book {

        private /*@ spec_public @*/ int lastPage;

        /*@ public invariant lastPage >= currentPage &&
          @                  currentPage > 0;
          @*/
        private /*@ spec_public @*/ int currentPage;


        /*@ public static invariant
          @ (\forall Book b1, b2;
          @   b1 != b2 ==> b1.isbn != b2.isbn);
          @*/
        private /*@ spec_public @*/ int isbn;

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

```
    @   0 <= i && i < bookmarks.length &&
    @   0 <= j && j < bookmarks.length;
    @   i != j ==>
    @   (bookmarks[i] != bookmarks[j] || bookmarks[i] == 0));
    @*/
  private /*@ spec_public @*/ int[] bookmarks = new int[10];

  /*@ public normal_behavior
    @   requires currentPage < lastPage;
    @   ensures currentPage == \old(currentPage) + 1;
    @   assignable currentPage;
    @ also
    @ public exceptional_behavior
    @   requires (currentPage == lastPage);
    @   signals_only BookException;
    @   assignable \nothing;
    @*/
  public void flip() {
  }

  /*@ public normal_behavior
    @   requires index >= 0 && index < bookmarks.length;
    @   requires bookmarks[index] != 0;
    @   ensures currentPage == bookmarks[index];
    @   assignable currentPage;
    @*/
  public void jump(int index) {
  }

  /*@ public normal_behavior
   @   ensures
   @   (\exists int i;
   @     0 <= i && i < bookmarks.length;
   @     bookmarks[i] == currentPage);
   @   ensures
   @   (\forall int i;
   @     0 <= i && i < bookmarks.length;
   @     bookmarks[i]==currentPage ||
   @     bookmarks[i]==\old(bookmarks[i]));
   @   assignable bookmarks[*];
   @*/
  public void mark() {
  }
}
```

Assignment 6 (Contract Fulfillment) (8p)

Look at the following JML contract for a method **int** `m()` of a class `C`:

```
public int x, y;

/*@ public normal_behavior
  @  requires x>=0 && y>=0;
  @  ensures  \result == (x>=y ? x : y);
  @*/
public int m() { ... }
```

"`(x>=y ?  x :  y)`" is a conditional expression, returning `x` if `x>=y`, and `y` otherwise.

(a) [5p]
Which of the following four implementations of `m()` respect the contract? Justify your answer in each case. You don't need to give a formal correctness proof. You can assume that the integers are unbound and no overflow occurs.

```
public int m() { // Implementation I1
  if (y >= 0) {
    if (x >= y) { return x;
    } else { return y; }
  } else return 0;
}

public int m() { // Implementation I2
  x = x - y;
  if (x >= 0) { return x + y;
  } else { return y; }
}

public int m() { // Implementation I3
  int t = x - y;
  if (t >= 0) { return t + y;
  } else { return y; }
}

public int m() { // Implementation I4
  if (x >= y) { return x;
  } else {
    if (y >= 0) { return y;
    } else { return x++; }
  }
}
```

(b) [3p]
Choose one of the implementations of `m()` that does not respect its contract, and suggest a patch of the *contract* such that the (unchanged) implementation respects that contract.

Solution

(a)

I1: Returns the correct result for a weaker precondition than required which is fine. Contract is fulfilled

I2: The method changes `x` *and, since the postcondition doesn't use* `\old`*, this can lead to a wrong result. For example, if initially* `x=3` *and* `y=2` *then* `x` *is changed to* `1`*. With that, the specified result is* `(1>=2 ? 1 : 2)`*, which is* `2`*. But the implementation returns* `3`*.*

I3: Looks similar as I2, but does not suffer from the same problem, as no field is overwritten. Contract is fulfilled

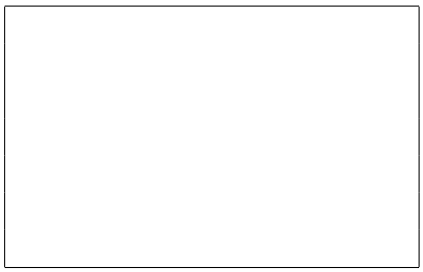
I4: Like in I2, the field `x` *is changed, but only when the requires clause is not satisfied. This is no harm, and the contract holds.*

(b) The only implementation that violates the contract is I2. One easy way to patch the contract is to enclose the conditional in the ensures clause with `\old`*. In this case, the side effect on* `x` *is not carried into the contract.*

Assignment 7 (Proof Obligations) (4p)

Take the *original* JML contract from the previous problem (i.e., not the one you wrote to answer question 6b). The following dynamic logic formula is the proof obligation, generated by KeY, for the correctness of the implementation of `m()` with respect to that contract. However, we have left out (indicated by the empty rectangle) some part of the DL formula. Your task is to write the part of the proof obligation formula which should be there instead of the rectangle, such that the whole formula is the right proof obligation generated from the given contract.

```
      wellFormed(heap)
    & !self = null
    & self.<created> = TRUE
    & C::exactInstance(self) = TRUE
    & measuredByEmpty
    & (self.x >= 0 & self.y >= 0 & self.<inv>)
 ->
```



```
        (  exc = null
         &   result
           = \if (self.x >= self.y)  \then (self.x)  \else (self.y)
         & self.<inv>
         & \forall Field f;
             \forall java.lang.Object o;
               (  (o, f) \in allLocs
                | !o = null
                & !o.<created>@heapAtPre = TRUE
                | o.f = o.f@heapAtPre))
```

Solution

It does not matter whether you write mathematical or KeY notation (in particular what concerns the diamond modality).

```
    {heapAtPre := heap}
      \<{
          exc = null;
          try {
            result = self.m()@C;
          }
```

```
      catch (Throwable t) {
        exc = t;
      }
    }\>
```

(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
```