EECS 4315 3.0 Mission Critical Systems

Midterm

9:00–10:15 on March 1, 2018

Last name:	
First name:	
EECS login:	

Instructions

- No questions are allowed during the test. If a question is not clear, then write down any assumptions made.
- One page of notes (letter size, double sided) may be used during the test.
- A non-electronic dictionary may be used during the test.
- Answer each question in the space provided.
- Make sure that you have answered all questions (test is double sided).
- Manage your time carefully.
- Last page can be used as scrap paper.

1 (4 marks)

Fill in the blanks.

(a) In the course Mission Critical Systems (EECS 4315), you implement listeners in Java to check properties. In the course System Specification and Refinement (EECS 3342), you apply **(i)** and **(ii)** to express properties.

(i) *set theory*
Marking scheme: 1 mark for set theory or sets

(ii) *predicate logic*
Marking scheme: 1 mark for predicate logic or logic

(b) In the course Mission Critical Systems (EECS 4315), you use a model checker. In the course System Specification and Refinement (EECS 3342), you use a(n) **(i)**.

(i) *theorem prover*
Marking scheme: 1 mark for theorem prover

(c) In the course Mission Critical Systems (EECS 4315), you focus on finding bugs. In the course System Specification and Refinement (EECS 3342), you focus on **(i)**.

(i) *designing code that is correct by construction*
Marking scheme: 1 mark for correctness

2 (4 marks)

Fill in the blanks.

(a) "Have you made what you were trying to make?" captures the term **(i)**.

(i) *verification*
Marking scheme: 2 marks for verification

(b) "Have you made the right thing?" captures the term **(i)**.

(i) *validation*
Marking scheme: 2 marks for validation

3 (4 marks)

How did Edger Dijkstra characterize the limitations of testing?

"Program testing can be used to show the presence of bugs, but never to show their absence!"
Marking scheme: 2 marks for "presence of bugs" and 2 marks for "not their absence."

4 (4 marks)

Nondeterministic code is code that, even for the same input, can exhibit different behaviors on different runs, as opposed to deterministic code. Mention two phenomena that give rise to nondeterministic code.

(i) *randomization*
Marking scheme: 2 marks for randomization

(ii) *concurrency*
Marking scheme: 2 marks for concurrency

5 (3 marks)

Fill in the boxes.

$$\begin{tikzcd}
\boxed{\text{(i)}} \arrow[r] & \boxed{\text{JPF}} \arrow[r] & \boxed{\text{(iii)}} \\
 & \boxed{\text{(ii)}} \arrow[u] &
\end{tikzcd}$$

(i) Input to JPF: *Java bytecode*
Marking scheme: 1 mark for Java bytecode, bytecode, Java code, or code

(ii) Input to JPF: *configuration file(s)* Marking scheme: 1 mark for configuration file(s)

(iii) Output of JPF: *report(s)* Marking scheme: 1 mark for report(s), console output, dot file, or state space diagram

6 (4 marks)

Consider the following app.

```
import java.util.Random;

public class App {
  public static void main(String[] args) {
    Random random = new Random();
    int counter = 0;
    while (counter > 0) {
      counter = random.nextInt();
      System.out.print(counter);
      }
   }
}
```

According to the Java API, `nextInt` returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence. The general contract of `nextInt` is that one int value is pseudorandomly generated and returned. All 2^{32} possible int values are produced with (approximately) equal probability.

Note: `counter` should have been initialized to 1. Hence, the answer "1, since the loop is never executed" receives (of course) full marks.

(a) Approximately to how many executions does the above app give rise? Explain your answer (an answer without an explanation does not receive any marks).

Infinitely many. The loop can have any number of iterations. (The probability of some of these executions is very small.)
Marking scheme: 2 marks for the observation that the loop can execute any number of iterations.

(b) Approximately how many states does the JPF model of the above app have? Explain your answer (an answer without an explanation does not receive any marks).

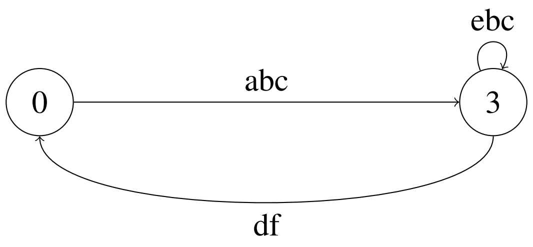
Roughly 2^{32}, as the state is captured by the value of the variable `counter` which is an `int` and, hence, has 2^{32} different values.
Marking scheme: 2 marks for the observation that the variable `counter` can have 2^{32} different values.

7 (5 marks)

Consider the following model, where 0 is the initial state.

$$\begin{tikzcd}
0 \arrow[r, "a"] & 1 \arrow[r, "b"] & 2 \arrow[r, "c"] & 3 \arrow[r, "d"] \arrow[ll, bend right, "e"'] & 4 \arrow[llll, bend left, "f"]
\end{tikzcd}$$

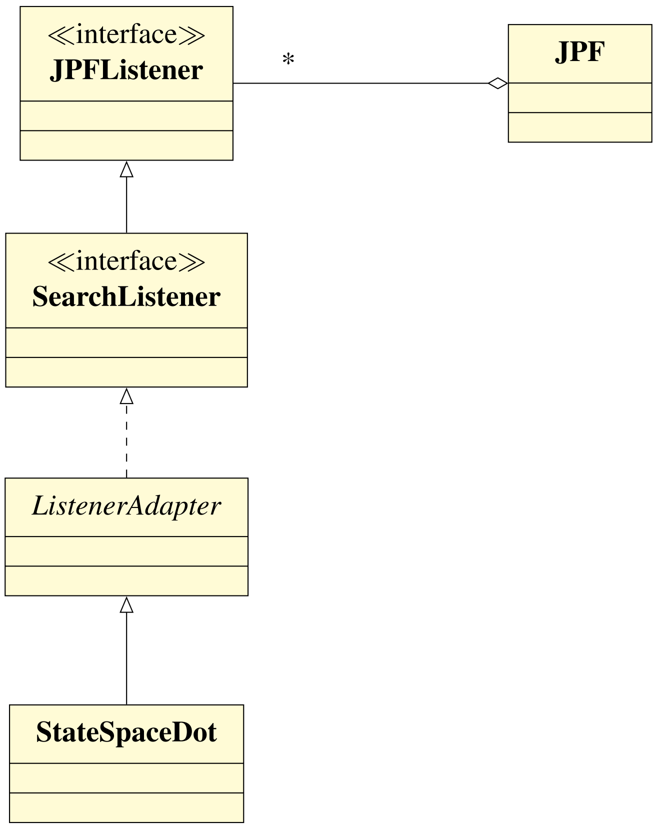
Draw the corresponding mini model. (Relevant definitions can be found at the end of this test.)



Marking scheme: 1 mark for state 0, 1 marks for state 3, 1 mark for transition labelled abc from 0 to 3, 1 mark for transition labelled ebc from 3 to 3, 1 mark for transition labelled df from 3 to 0

8 (4 marks)

Draw the UML diagram consisting of the classes and interfaces `JPF`, `JPFListener`, `SearchListener`, `ListenerAdapter` and `StateSpaceDot`.



Marking scheme: 1 mark for each correct relationship between classes and interfaces (connecting the correct classes/interfaces and using the correct connector)

9 (4 marks)

The `VMListener` interface contains the method declaration

```
void executeInstruction(VM vm, ThreadInfo thread, Instruction instruction);
```

The parameters provide a way to flow information from one object to another.

(a) What is the type of the object from which the information flows?

JPF.
Marking scheme: 2 marks for JPF.

(b) What is the type of the object to which the information flows?

A class implementing VMListener.
Marking scheme: 2 marks for a class implementing VMListener, 1 mark for VMListener

10 (4 marks)

What is a native method?

A method that is implemented in a language other than Java but that is invoked from a Java app.
Marking scheme: 2 marks for "implemented in a language other than Java" and 2 marks for "invoked from a Java app."

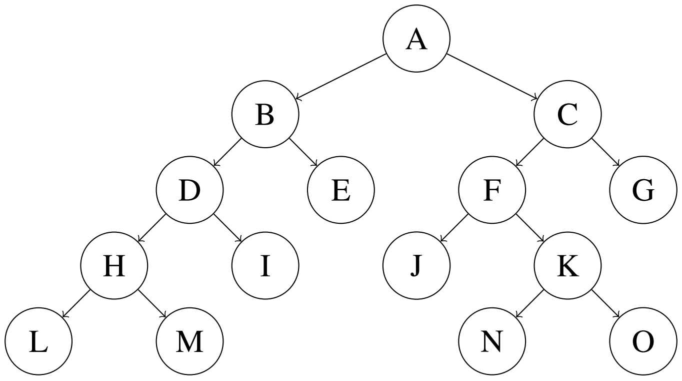
11 (4 marks)

Explain the difference between a peer class and a native peer class.

A peer class captures the behaviour of a native method in pure Java. Its code is model checked by JPF. Native peers are the JPF analogue of native code. Its code is executed by the host JVM.
Marking scheme: 2 marks for "model checked by JPF" and 2 marks for "executed by the host JVM."

12 (6 marks)

A bounded search is a depth-first search that starts in a given state and where the depth is bounded. For simplicity, we assume here that the bound is 2.



For example, for the above state space, if started in state B, the states are visited in the following order: B, D, H, I, E.

Recall the following methods.

- `public boolean forward()`: tries to move forward along an unexplored transition and returns whether the move is successful.
- `public boolean backtrack()`: tries to backtrack and returns whether the backtrack is successful.
- `private RestorableVMState getRestorableState()`: returns the current state so that it is restorable.
- `private void restoreState(RestorableVMState state)`: restores the given state.
- `public boolean isNew()`: tests whether the current state is new, that is, it has not been visited before.
- `public boolean isEnd()`: tests whether the current state an end (final) state.

(a) For the above state space, provide the sequence of calls to `forward` and `backtrack` and the value returned by them corresponding to the bounded search started in state B.

forward(true), forward(true), backtrack(true), forward(true), backtrack(true), forward(false), backtrack(true), forward(true), forward(false), backtrack(true), forward(false)
Marking scheme: 2 marks for the correct answer, 1 marks if the sequence contains at most 4 mistakes

(b) Give an algorithm (in terms of pseudocode or Java code) using some of the above methods that gives rise to the sequence of part (a) if started in state B.

```
private void search(RestorableVMState state) {
  this.restoreState(state);
  int depth = 1;
  do {
    depth--;
    while (depth < 2 && this.forward()) {
      depth++;
    }
  } while (depth > 0 && this.backtrack());
}
```

Marking scheme: 2 marks for code that gives rise to the correct sequence (this can be accomplished in many different ways), 1 mark for code that contains the key ingredients (depth, calls to forward and backtrack, loops)

(c) One can invoke the bounded search repeatedly starting in different states so that the whole state space is traversed. One would start with the initial state. That first bounded search may visit states but not explore their outgoing transitions. For these states, bounded search can be invoked subsequently. For example, if such a repeated bounded search is applied to the above state space, the states are visited in the following order: A, B, D, E, C, F, G, H, L, M, I, J, K, N, O.

Give an algorithm (in terms of pseudocode or Java code) that implements this repeated bounded search. You may have to make some modifications to the bounded search you developed in part (b).

```
private void search(RestorableVMState state,
                    Queue<RestorableVMState> queue) {
  this.restoreState(state);
  int depth = 1;
  do {
    depth--;
    while (depth < 2 && this.forward()) {
      depth++;
      if (depth == 2 && this.isNew() && !this.isEnd()) {
        queue.offer(this.getRestorableState());
      }
    }
  } while (depth > 0 && this.backtrack());
}

public void search() {
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    search(queue.poll(), queue);
  }
}
```

Marking scheme: 2 marks for code that gives rise to the correct sequence (this can be accomplished in many different ways), 1 mark for code that contains the key ingredients (use a data structure to store the states that are not fully explored, store states in the data structure and extract them from it)

Definitions

A labelled transition system is a tuple $\langle S, A, \to, s_0 \rangle$ consisting of

- a set S of states,
- a set A of actions,
- a transition relation $\to \subseteq S \times A \times S$, and
- a start state $s_0 \in S$.

The set $\mathit{succ}(s)$ of successors of the state s is defined by

$$\mathit{succ}(s) = \{\, t \in S \mid \exists a \in A : s \xrightarrow{a} t \,\}.$$

The set $\mathit{pred}(s)$ of predecessors of the state s is defined by

$$\mathit{pred}(s) = \{\, t \in S \mid \exists a \in A : t \xrightarrow{a} s \,\}.$$

The labelled transition system $\langle S^+, A^+, \to^+, s_0 \rangle$ consists of

- $S^+ = \{s_0\} \cup \{\, s \in S \mid |\mathit{succ}(s)| \neq 1 \,\}$,
- A^+ is the set of nonempty and finite sequences of actions,
- $s_1 \xrightarrow{a_1 \ldots a_n}{}^{+} s_{n+1}$ if

$$\begin{array}{l} s_1 \in S^+ \wedge s_{n+1} \in S^+ \wedge \\ \exists s_2, \ldots, s_n \in S \setminus S^+ : \forall 1 \leq i < n : s_i \xrightarrow{a_i} s_{i+1} \wedge \\ \forall 1 \leq i, j \leq n : s_i = s_j \Rightarrow i = j \end{array}$$