TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA

UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2011

END-OF-YEAR

SWEN 102
Introduction to Software Modelling

Time Allowed: 3 Hours

Instructions: There are 180 possible marks on the exam.
Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use a separate answer booklet.
Some example Alloy code is provided on the last page.
Non-electronic foreign language dictionaries are allowed.
Calculators ARE NOT ALLOWED.
No other reference material is allowed.

Question	Topic	Marks
1.	Use Case Descriptions	30
2.	Object and Class Diagrams I	30
3.	Object and Class Diagrams II	30
4.	Writing Invariants	30
5.	Using Alloy	30
6.	State Machines	30
	Total	180

continued...

Student ID: ....

Question 1. Use Case Descriptions [30 marks]

(a) [3 marks] Perform a *textual analysis* on the following description, to find candidate use cases. You should carefully and neatly underline key verb phrases in the text.

The IntraNet of the University of MiddleEarth contains a Computer Science website that enables students to write reviews of its Courses and Lecturers. Computer Science has 22 courses in Network Engineering, 18 courses in Software Engineering, 12 courses in Hardware Design including 4 in Electronic Testing Techniques, 5 related courses on Research Design and 3 courses in Project Management and Professional Practice.

Any student at the University can register to login to this website and, after authorization, they can search for a course and read reviews. Only students registered for a Computer Science course can write reviews of their courses and lecturers. New reviews are checked by Course Administrators before being posted on the site.

When new courses are made the Course Administrator adds them to the website. They must include a description of the content and assessment details. The Computer Science website is managed by the IT Administrators who delete old information, amend false information and organize the technical issues of registration and login.

(Question 1 continued on next page)

(Question 1 continued)

(b) [9 marks] Draw a **use case diagram** showing at least 3 actors and at least 6 use cases that you would produce to model of this system.

(Question 1 continued on next page)

Student ID: . . . .

(Question 1 continued)

(c) [12 marks] Provide **use case descriptions** for the main success sequence of the following **three** use cases.

Create Course

Write Review

Amend Information

(Question 1 continued on next page)

(Question 1 continued)

(d) [6 marks] Give characteristics for two actors in the system [6 marks]

Actor Name:

Domain Knowledge:

System Knowledge:

Actor Name:

Domain Knowledge:

System Knowledge:

Question 2. Object and Class Diagrams I [30 marks]

(a) [15 marks] Based on the description from Question 1 on Page 2, draw a *well-designed* **class diagram** to model the Course and Lecturer Review system. This should contain at most 6 classes and 10 attributes and use inheritance and associations where appropriate.

Student ID: . . . .

(b) [5 marks] In the box below, draw an *object diagram* consistent with your class diagram which captures the following scenario:

> "The course administrator created an entry for the new course SWEN103. Mel submitted a review, which was placed in a queue to be checked by the course administrator."

(c) [5 marks] Briefly, discuss what *functional requirements* are. You should include at least one example of a functional requirement for the Course and Lecturer Review system.

Student ID: ....

(d) [5 marks] Briefly, discuss what *non-functional requirements* are. You should include at least one example of a non-functional requirement for the Course and Lecturer Review system.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

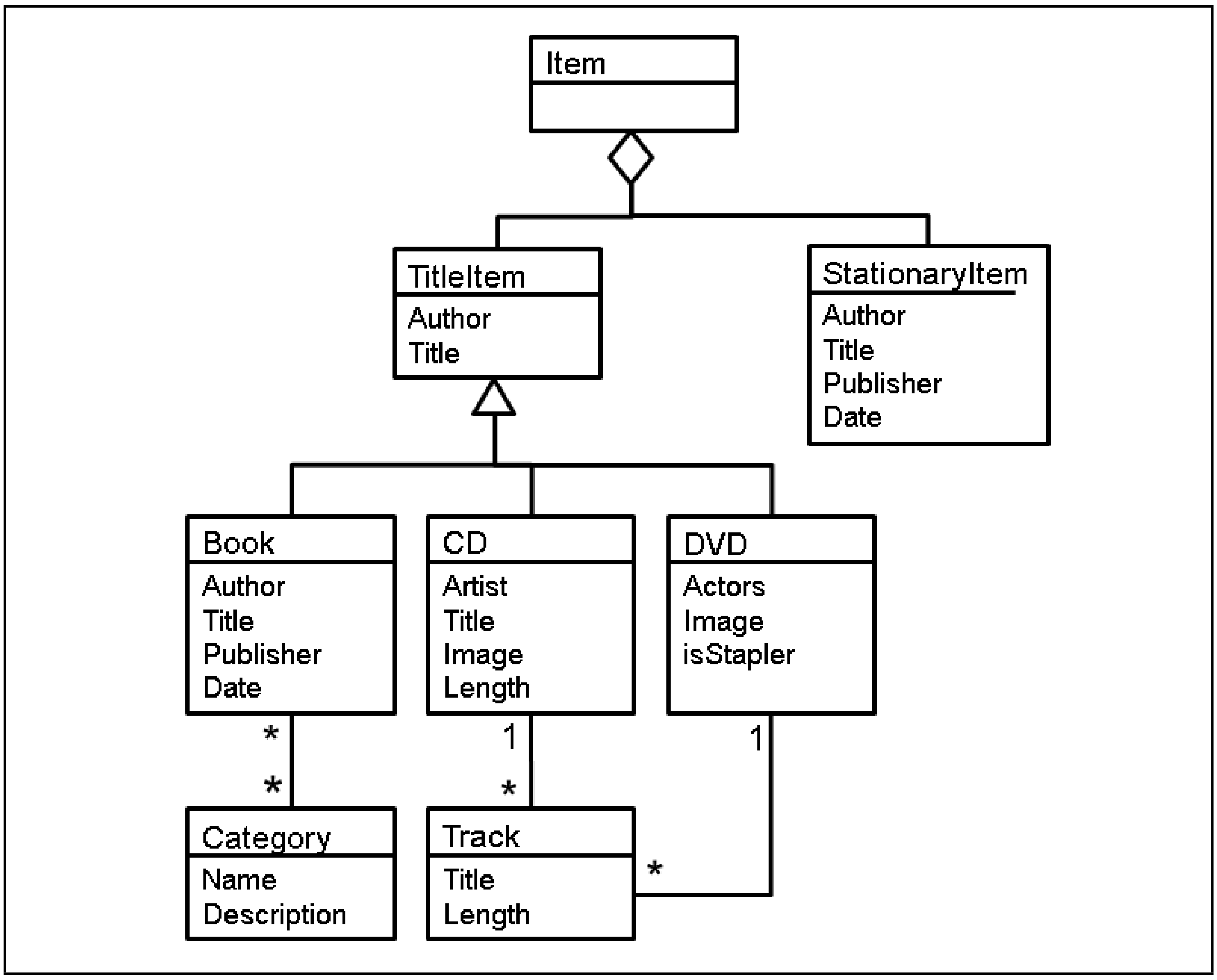
Student ID: . . . .

Question 3. Object and Class Diagrams II [30 marks]

Consider the following description for a stationary shop:

> A shop in the main street sells books, magazines and stationary items. Stationary items include pens, pencils, writing pads, staplers and movies on DVDs.
>
> Books are sold in different parts of the shop depending on their subject. The subject categories include gardening, history, travel, childrens stories and adult fiction. Every book has an author, publisher, date of publication and category.
>
> DVDs are recorded with the movie title, the actors in the movie, a cover image and the length of playing time.
>
> Each CD has a title, an artist, a cover image and the number of tracks each with its own title and running time.
>
> Magazines have a title, issue number, month of publication, publisher, distributer name and address and phone number.

You are given the following class diagram describing the various items sold in the shop:



(Question 3 continued on next page)

Student ID: . . . .

(Question 3 continued)

(a) [15 marks] Circle and number **five distinct** problems in the class diagram on the previous page. Describe why each problem is a problem and how you would solve it.

1)

2)

3)

4)

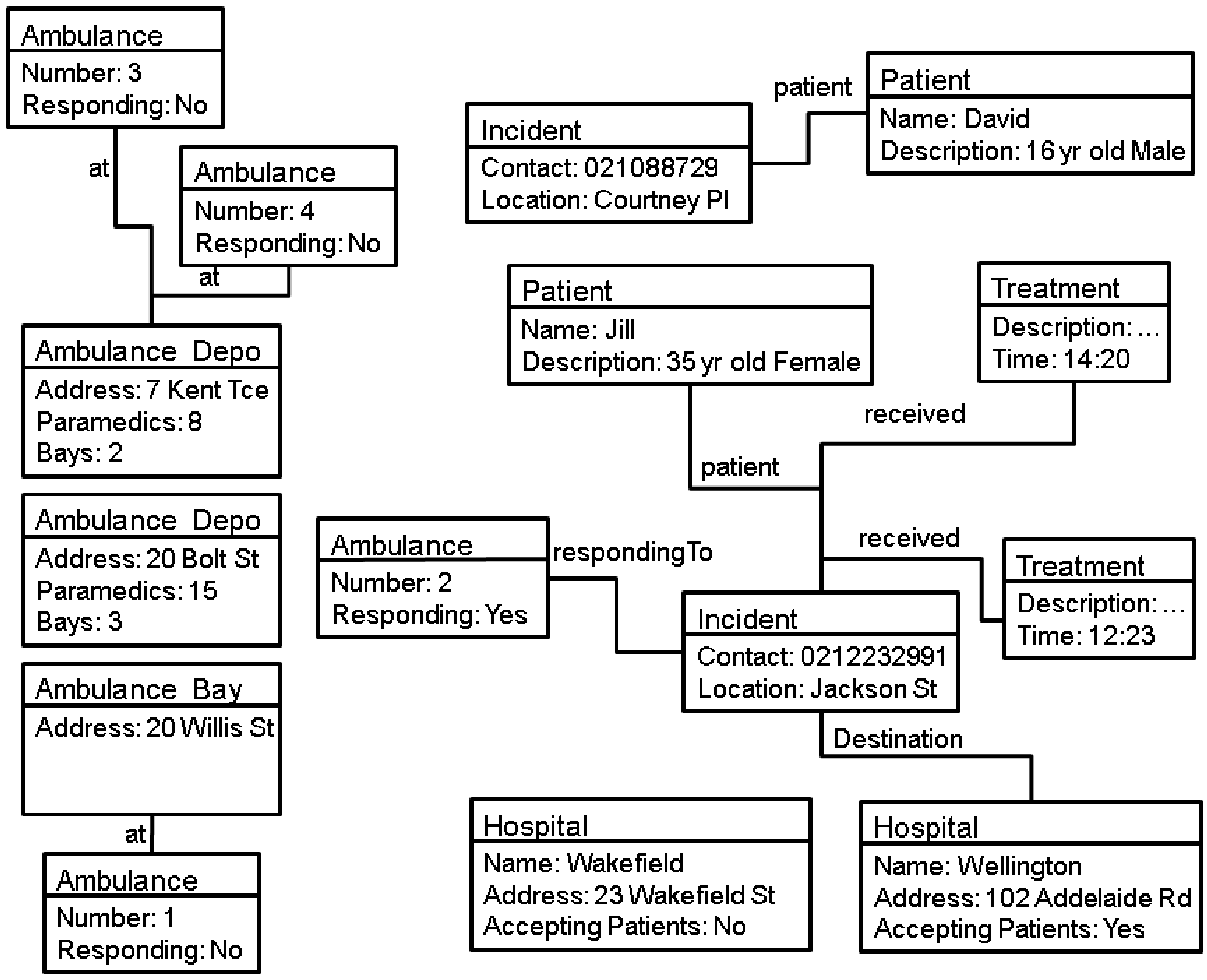
5)

(Question 3 continued on next page)

Student ID: . . . .

(Question 3 continued)

(b) [15 marks] Consider the object diagram below and draw a corresponding class diagram on the facing page.



(Question 3 continued on next page)

Student ID: . . . .

(Question 3 continued)

SPARE PAGE FOR EXTRA ANSWERS

Student ID: . . . .

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

Question 4. Alloy I [30 marks]

Consider the following (incomplete) description of the *tic-tac-toe* game:

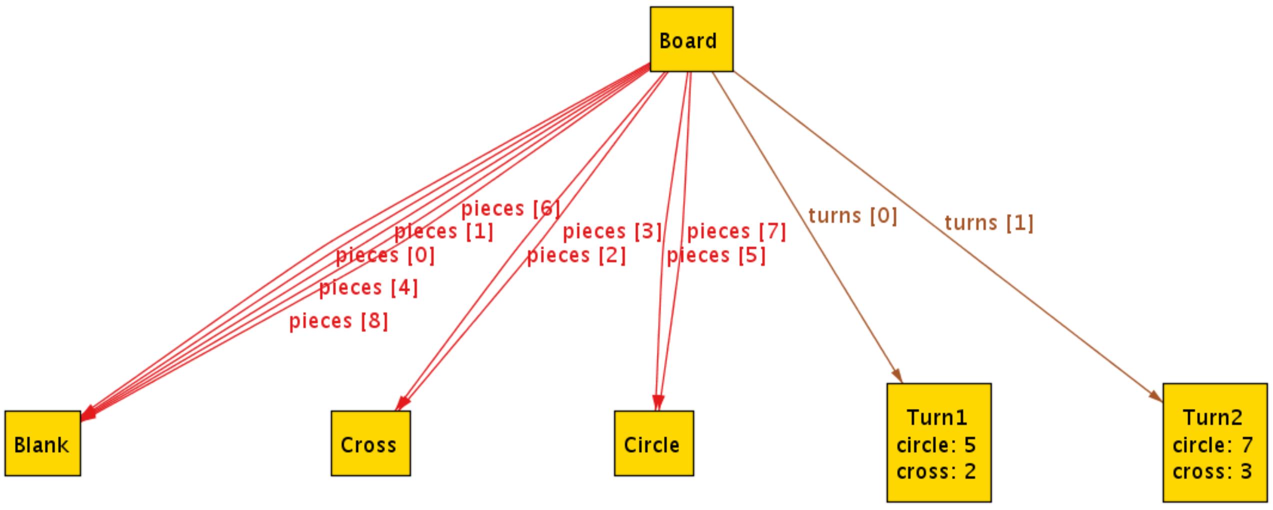
> "A tic-tac-toe game requires two players and a board with 9 squares (which are initially empty). During the game, each player takes it turn to place one of their pieces (either a circle or a cross). The player who starts always uses circles, whilst the other player always uses crosses."

Consider the following Alloy model of the above description, and corresponding object diagram:

```
abstract sig Piece {}
one sig Blank extends Piece {}
one sig Cross extends Piece {}
one sig Circle extends Piece {}

sig Turn { circle: Int, cross: Int }
{ circle >= 0 && circle < 9 && cross >= 0 && cross < 9 }

sig Board { pieces : seq Piece, turns : seq Turn }
{
 #pieces = 9
 all t : turns.elems | pieces[t.circle] in Circle
 all t : turns.elems | pieces[t.cross] in Cross
 all disj i1, i2 : turns.inds | turns[i1].cross != turns[i2].cross
 all disj i1, i2 : turns.inds | turns[i1].circle != turns[i2].circle
 all i : pieces.inds | pieces[i] in Circle implies some t : turns.elems | t.circle = i
 all i : pieces.inds | pieces[i] in Cross implies some t : turns.elems | t.cross = i
}
```

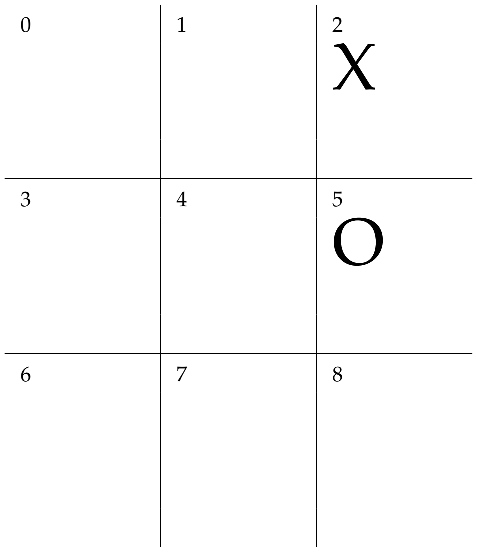


The object diagram was generated from the given alloy model using an appropriate `run` command.

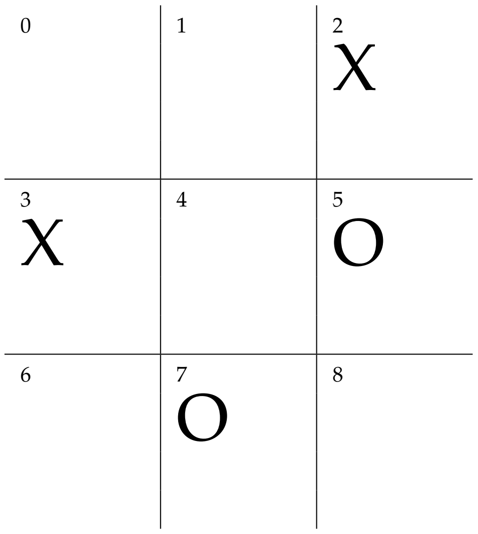
(Question 4 continued on next page)

(Question 4 continued)

(a) [5 marks] By adding circles and crosses to the following tic-tac-toe boards, illustrate the game shown in the object diagram given on the previous page. You should follow the mapping of indices to squares indicated in the diagrams.



(After Turn 1)



(After Turn 2)

(b) [5 marks] Evaluate each of the following Alloy expressions on the object diagram given on the previous page:

```
Board.pieces[2]     ={Cross}
```

```
#(Board.pieces.Circle)     =2
```

```
Board.turns     ={0->Turn$1,1->Turn$0}
```

```
Circle.~(Board.pieces)     ={5,7}
```

```
some t :  Board.turns.elems | t.circle > t.cross     =true
```

(Question 4 continued on next page)

(Question 4 continued)

(c) [8 marks] For each of the following statements from the Alloy model, describe (in English) what it achieves:

```
circle >= 0 && circle < 9 && cross >= 0 && cross < 9
```

For each turn, both players must place a piece within the bounds of (i.e. on) the board.

```
all t : turns.elems | pieces[t.circle] in Circle
```

Every piece placed by the circle player must, in fact, be a Circle

```
all disj i1, i2 : turns.inds | turns[i1].cross != turns[i2].cross
```

The cross player cannot place a cross in a position already occupied by a cross.

```
all i : pieces.inds | pieces[i] in Circle implies some t : turns.elems | t.circle = i
```

For every Circle on the board, there must have been a turn where the circle player placed that circle.

(Question 4 continued on next page)

(Question 4 continued)

(d) Consider the following additional requirement for a tic-tac-toe game:

> "The board is full when a piece has been placed on every square. There will always be more circles than crosses on a full board."

(i) [3 marks] Briefly, discuss why this must be true.

Because there are nine squares, and the circle player always begins. After 4 turns there are an even number of crosses and circles placed. Thus, after the final move, there will be 5 circles and 4 crosses on the board.

(ii) [4 marks] The Alloy model given on page 16 does not permit instances corresponding to a full board. Briefly, discuss what the problem is with the model.

The model requires that, on every turn, both a circle and a cross is placed. Therefore, there are always an even number of pieces on the board and, hence, the final turn cannot occur because it would require placing a cross over an existing piece — which is not allowed.

(iii) [5 marks] Briefly, outline how you would modify the Alloy model to permit instances corresponding to a full board.

The model could be modified so that each turn only places a single piece, which would mean 9 turns were possible. Constraints would need to be added to ensure that turns alternate between players, starting with circle.

Student ID: . . . .

Question 5. Alloy II [30 marks]

A *linked list* is a chain of zero or more *links* that begins with a special *header* link. Links cannot be shared between lists. Consider the following simple model for linked lists:

```
sig Link {
 next : lone Link,
 data : Int
}

sig Header extends Link { length : Int }
```

(a) [12 marks] Translate each of the following statements into an Alloy fact.

1) Every `data` item in a `Link` is a positive integer.

```
fact { all l :  Link | l.data > 0 }
```

2) Every chain of `Links` begins with an instance of `Header`.

```
fact { all l :  Link | no l.~next implies l in Header }
```

3) Every instance of `Header` begins a chain of `Links`.

```
fact { all h :  Header | no h.~next }
```

4) Field `length` counts the number of `Links` reachable from the `Header` (including itself).

```
fact { all h :  Header | h.length = #(h.*next) }
```

5) Every `Link` is reachable from exactly one `Header` (i.e. no sharing).

```
fact { all l :  Link | one h :  Header | l in h.^next }
```

6) No `Link` is reachable from itself (i.e. no cycles).

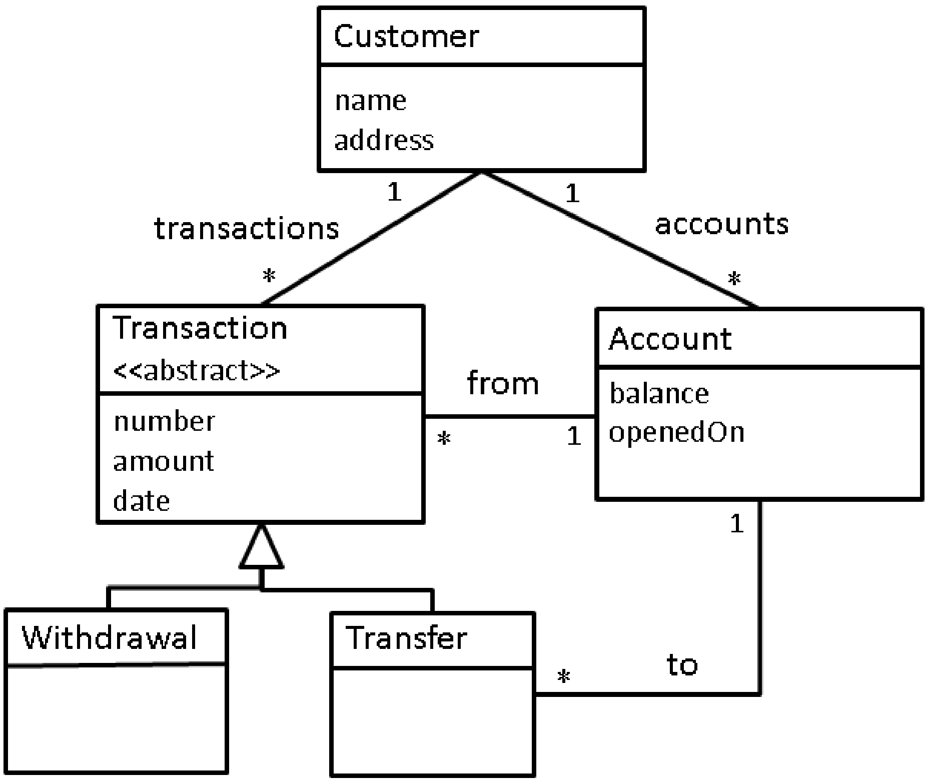
```
fact { no l :  Link | l in l.^next }
```

(Question 5 continued on next page)

Student ID: ....

(Question 5 continued)

Consider the following class diagram for an *account system*, and the additional rules given below.



- The amount on a transaction must be a positive integer.
- A transaction for a customer must occur on one of his/her accounts.
- A transaction cannot occur on an account before it is open.
- Every transaction has a different number.
- A transfer cannot be made from / to the same account.

Student ID: . . . .

(b) [18 marks] In the box below, provide an Alloy model for the account system. You should use `Ints` to represent times, and you may assume unlimited bit-width.

```
sig Customer {
        name : String,
        address : String
        accounts : set Account
        transactions : set Transaction
}

sig Account {
        balance : Int,
        openedOn : Int
}{
        one ~accounts
}

abstract sig Transaction {
        number : Int
        amount : Int
        date : Int
        from : one Account
}{
        one ~transactions
        amount >= 0
        date > from.openedOn
        this.~transactions = from.~accounts || this in Transfer
}

sig Withdrawal extends Transaction {
}

sig Transfer extends Transaction {
        to : one Account
}{
        from != to
        this.~transactions = from.~accounts || this.~transactions = to.~accounts
}

fact { all disj t1,t2 : Transaction | t1.number != t2.number }
```

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

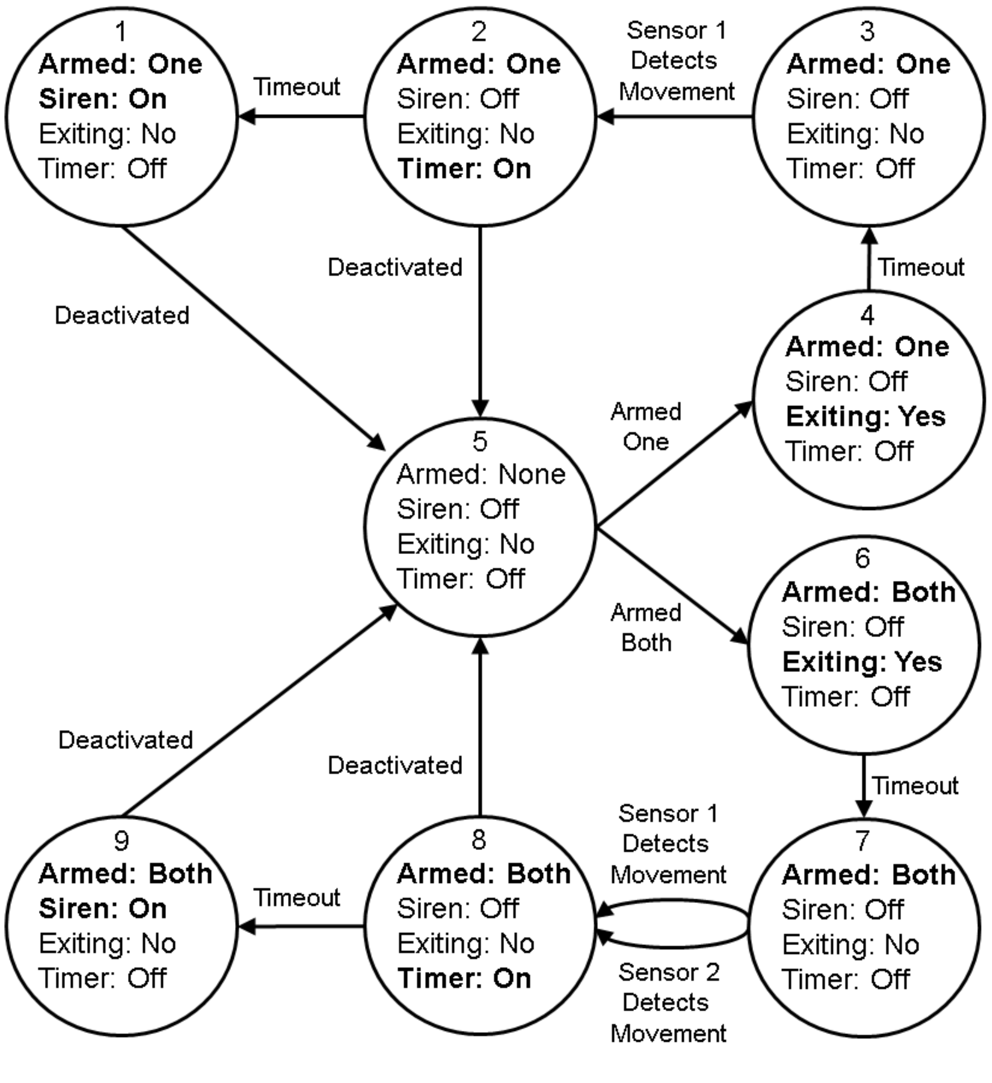
Student ID: . . . .

Question 6. State Machines

Consider the following description of a simple *alarm system* with timers:

> "The alarm system protects houses against break-ins. The system has two *sensors* which detect movement in the home. The owner can *arm* one or both of the sensors. When no-one is home, both sensors should be armed. At night, the downstairs sensor should be armed so the owner can move around upstairs. When one or more sensors is armed, the system begins an *exit timer*. Whilst the exit timer is active, the *siren* will not sound. Once exiting is complete, the system is fully armed. When an armed sensor detects movement, an *alarm timer* begins. This gives time for the owner to deactivate the system when returning home. However, if the alarm is not deactivated before time runs out, the siren will sound and continue until the system is deactivated."

A *state machine diagram* for the alarm system has been provided:



1 Armed: One Siren: On Exiting: No Timer: Off
2 Armed: One Siren: Off Exiting: No Timer: On
3 Armed: One Siren: Off Exiting: No Timer: Off
4 Armed: One Siren: Off Exiting: Yes Timer: Off
5 Armed: None Siren: Off Exiting: No Timer: Off
6 Armed: Both Siren: Off Exiting: Yes Timer: Off
7 Armed: Both Siren: Off Exiting: No Timer: Off
8 Armed: Both Siren: Off Exiting: No Timer: On
9 Armed: Both Siren: On Exiting: No Timer: Off
Timeout
Sensor 1 Detects Movement
Sensor 2 Detects Movement
Deactivated
Armed One
Armed Both

(Question 6 continued on next page)

Student ID: . . . .

(Question 6 continued)

(a) For each of the following statements, indicate whether it is a true or false statement based on the state machine diagram.

(i) [2 marks] Sensor 2 may be armed whilst sensor 1 is not.

False

(ii) [2 marks] The siren may sound when the exit timer is active.

False

(iii) [2 marks] When the siren is deactivated, the system remains armed.

False

(iv) [2 marks] A *timeout* does not always result in the siren being activated.

True

(b) Provide a suitable *execution trace* for the following scenarios. Your execution trace may start from whichever state you chose.

(i) [2 marks]

> *"Before going to bed, John arms the downstairs sensor. Later, a sensor is activated and the siren sounds. Luckily, it was John's cat and he quickly deactivates the system."*

$5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 5$

(ii) [2 marks]

> *"Sue arms both sensors and leaves the house. But, she forgot her bag. She goes back into the house and immediately deactivates the system before the alarm sounds."*

$5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 5$

(Question 6 continued on next page)

Student ID: . . . .

(Question 6 continued)

(c) Consider the following *incomplete* Alloy model of the alarm system:

```
enum Bool {True, False}
enum Sensor {None, Both}

sig AlarmState {
 armed : Sensor, siren : Bool, exiting : Bool, timer : Bool
}

pred init(s : AlarmState) {
 s.armed = None and s.siren = False and s.exiting = False and s.timer = False
}

pred armSystem(s,s' : AlarmState) {
 s.armed = None
 s'.armed = Both and s'.siren = False and s'.timer = False and s'.exiting = True
}

pred sensorActivated(s,s' : AlarmState) {
 s.armed = Both and s.siren = False  and s.timer = False and s.exiting = False
 s'.armed = Both and s'.siren = False and s'.timer = True and s'.exiting = False
}

pred finished(s,s' : AlarmState) {
 s.exiting = True
 s'.armed = Both and s'.siren = False and s'.timer = False and s'.exiting = False
}

pred timeOut(s,s' : AlarmState) {
 s.timer = True
 s'.armed = s.armed and s'.siren = True and s'.timer = False and s'.exiting = False
}

pred deactivated(s,s' : AlarmState) {
 s.armed = Both
 s'.armed = None and  s'.siren = False and  s'.timer = False and s'.exiting = False
}

pred transition(s1, s2: AlarmState) {
  armSystem[s1,s2] or finished[s1,s2] or sensorActivated[s1,s2] or timeOut[s1,s2] or deactivated[s1,s2]
}

sig ExecutionTrace { states: seq AlarmState }{
  init[states[0]] and all i: states.inds | i > 0 implies transition[states[i−1], states[i]]
}
```

(Question 6 continued on next page)

Student ID: . . . .

(Question 6 continued)

(i) [2 marks] Give an instance of AlarmState which corresponds to state eight from the diagram on page 24.

armed: Both
siren: false
timer: true
exiting: false

(ii) [2 marks] Give an instance of ExecutionTrace which corresponds to the execution trace "5 → 6 → 7" from the diagram on page 24.

armed: None siren: false timer: false exiting: false	⟶	armed: Both siren: false timer: false exiting: true	⟶	armed: Both siren: false timer: false exiting: false

(iii) [2 marks] What key functionality is provided in the state machine diagram but not by the Alloy model?

The state machine diagram permits none, one or both sensors to be activated, whilst the alloy model only permits none and both

(iv) [2 marks] What transition in the state machine diagram does the predicate finished correspond to?

$6 \longrightarrow 7$

(Question 6 continued on next page)

Student ID: . . . .

(Question 6 continued)

(v) [2 marks] Can the system described by the Alloy model be deactivated whilst exiting is in progress?

Yes, there is nothing in the predicate deactivated which prevents this

(vi) [2 marks] How many different instances of ExecutionTrace are possible with exactly *three* states?

2 instances are possible.
($5 \longrightarrow 6 \longrightarrow 7$ and $5 \longrightarrow 6 \longrightarrow 5$)

(vii) [2 marks] How many different instances of ExecutionTrace are possible with exactly *four* states?

3 instances are possible.
($5 \longrightarrow 6 \longrightarrow 7 \longrightarrow 8$, $5 \longrightarrow 6 \longrightarrow 7 \longrightarrow 5$ and $5 \longrightarrow 6 \longrightarrow 5 \longrightarrow 6 \longrightarrow 5$)

(viii) [4 marks] Suppose we add "exiting = True **implies** armed = Both" as an invariant on AlarmState. Briefly, discuss whether or not this changes the number of valid instances of the model.

No, because it's already impossible to reach a state where existing = True and armed = None

.

****

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

(This page may be detached)

Appendix

Example Alloy code for the Monopoly board is given here for reference.

```
sig Name, Colour {}

abstract sig Building {}{
  some buildings.this
}
sig House extends Building {}
sig Hotel extends Building {}

fact {
  // There can be no more than 32 houses and 12 hotels at one time
  // For simplicity, we lower these values since they're too big for Alloy!
  #House <= 7
  #Hotel <= 2
}

sig Player {}

sig Property {
  name: Name,
  owner: lone Player,
  buildings: set Building,
  colourGroup : one ColourGroup,
}{
 // Up to four houses or one hotel in buildings
 #buildings <= 4
 some h : Hotel | h in buildings
 all h : Hotel | h in buildings implies #buildings = 1

 // built properties must have an owner
 some buildings implies one owner
}

sig Mortgaged in Property {}{ no buildings }

sig ColourGroup {
 colour: Colour,
}{
 // Every ColourGroup has a unique colour
 all cg : ColourGroup | cg.@colour = colour implies cg = this
```

continued...

```
 // ColourGroup Arity
 #~colourGroup >= 2 && #~colourGroup <= 3

 // You must own a whole ColourGroup in order to build
 all p : this.~colourGroup | (some p.buildings) implies
   (all p' : this.~colourGroup | p.owner = p'.owner)

 // buildings must be equally distributed
 all disj p, p' : this.~colourGroup, h: Hotel |
   (h in p.buildings) implies ((some h' : Hotel | h' in p'.buildings) ||
                               #p'.buildings = 4)

 all disj p, p' : this.~colourGroup |
   (no h : Hotel | h in p.buildings) implies (
   #p.buildings = #p'.buildings ||
   #p.buildings = #p'.buildings + 1 ||
   #p.buildings = #p'.buildings - 1 ||
   (#p.buildings = 4 && some h : Hotel | h in p'.buildings))
}

run {} for 4 but 2 ColourGroup, 8 Building, 6 int, exactly 1 Hotel
```