TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VUW VICTORIA UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2015

TRIMESTER 2

SWEN224

Formal Foundations of Programming

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Instructions: Answer all questions
All questions are of equal value

Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use the spare pages.
If further additional space is required you may use a separate answer booklet.

An appendix is provided for the LTSA syntax.

Question	Topic	Marks
1.	Specification & Verification I	30
2.	Model Checking A Door Lock	30
3.	Model Checking II	30
4.	Specification & Verification II	30
	Total	120

Question 1. Specification & Verification I [30 marks]

(a) For each of the following, provide one set of parameter values which meet the precondition and one set which does not.

(i) [2 marks]

```
function invert(int u8) -> (int r)
requires 0 <= u8 && u8 <= 255:
    //...
```

Meets preconditon: `u8=0`

Does not meet precondition: `u8=-1`

(ii) [2 marks]

```
function get(int[] items, int i) -> (int r)
requires 0 <= i && i < |items|:
    //...
```

Meets preconditon: `items=[0],i=0`

Does not meet precondition: `items=[],i=0`

(iii) [3 marks]

```
function find(int[] items, int item) -> (bool r)
requires all { i in 0..|items| | items[i] >= 0 }:
    //...
```

Meets preconditon: `items=[0],item=0`

Does not meet precondition: `items=[-1],item=0`

(iv) [3 marks]

```
function findDuplicate(int[] xs) -> (int r1, int r2)
requires some { i in 0..|xs|, j in 0..|xs| | i != j && xs[i] == xs[j] }:
    //...
```

Meets preconditon: `xs=[0,1,0]`

Does not meet precondition: `xs=[1,2,3]`

(b) For each of the following functions, briefly discuss why it does not meet its specification. In each case, you should provide parameter and return values to illustrate.

(i) [5 marks]

```
function compareTo(int x1, int y1, int x2, int y2) -> (int r)
ensures (r == -1) ==> (x1 < x2 || (x1 == x2 && y1 < y2))
ensures (r == 0) ==> (x1 == x2 && y1 == y2)
ensures (r == 1) ==> (x1 > x2 || (x1 == x2 && y1 > y2)):
    //
    if x1 < x2:
        return -1
    else if x1 > x2:
        return 1
    else:
        return 0
```

Does not meet specification because post condition requires that when `r=0` both `x1==x2` and `y1==y1`, but implementation never examines `y1` and `y2` in this case.

Parameter and Return values: `x1=0,x2=0,y1=0,y2=1` gives `r=0` but should be `r=1`

(ii) [5 marks]

```
function isStriped(bool[] items) -> (bool r)
requires |items| > 0
ensures r ==> all { i in 1 .. |items| | items[i-1] != items[i] }
ensures !r ==> some { i in 1 .. |items| | items[i-1] == items[i] }:
    //
    int i = 1
    bool last = items[0]
    //
    while i < |items|
    where i >= 1
    where all { k in 1 .. i | items[k-1] != items[k] }:
        //
        if items[i] == last:
            return false
        i = i + 1
    //
    return true
```

Implementation never updates `last` variable, meaning that all comparisons are made against first element in array. This goes against the postcondition, which requires adjacent elements are compared.

Parameter and Return values: `items=[true,false,true]` gives **`false`** but should be **`true`**

(c) Consider the following implementation for the function `fill()`.

```
function fill(int[] items, int item, int index) -> (int[] r):
    //
    if index == |items|:
        return items
    else:
        items[index] = item
        return fill(items, item, index+1)
```

(i) [4 marks] Describe in your own words what the function `fill()` does.

The fill function sets all elements of the array `items` from `index` upto the end of the array to `item`

(ii) [6 marks] Provide an appropriate specification for function `fill()`.

```
requires 0 <= index && index <= |items|
ensures |items| == |r|
ensures all { i in index .. |r| | r[i] == item }
ensures all { i in 0 .. index | r[i] == items[i] }
```

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.

Question 2. Model Checking A Door Lock [30 marks]

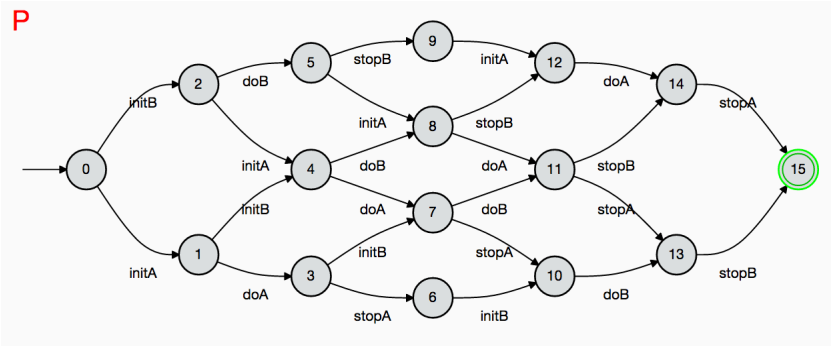
(a) The LTSA language:

(i) [3 marks] Draw the automaton of the following process:

```
||P = (ProgA||ProgB).

ProgA = (initA->doA->stopA->STOP).

ProgB = (initB->doB->stopB->STOP).
```

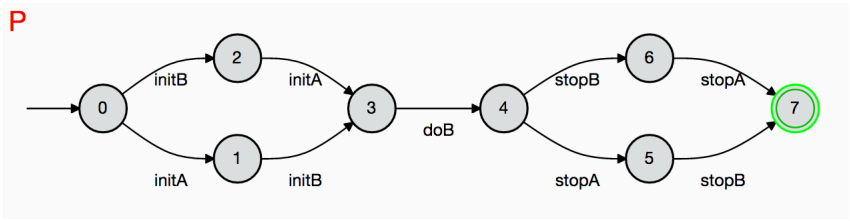


(ii) [2 marks] Briefly explain what is meant by event interleaving.

When two processes are run in parallel and do not interact in any way then an observer can see any interleaving of the events from the two processes. That is there is no fixed order between any two events from different processes.

(iii) [3 marks] Draw the automaton of the following process:

```
||P = (ProgA||ProgB)/{doB/doA}.
ProgA = (initA->doA->stopA->STOP).
ProgB = (initB->doA->stopB->STOP).
```



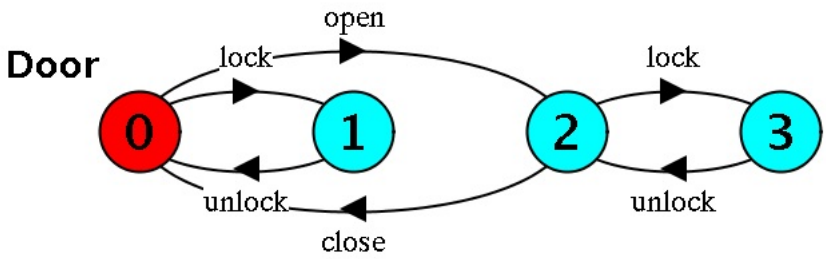
(iv) [2 marks] Briefly explain what is meant by event synchronization.

When two processes are run in parallel an event from one processes can synchronize with an event from the other processes, in LTSA the synchronizing events must have the same name. An event that synchronizes with another is blocked until the other event is ready to be performed and when the synchronizing event is performed it is performed by both processes at the same time.

(b) Simple door specification

A **Door** is initially closed and unlocked. It can **open**, and when open it can **close**. In addition the door can be **lock**ed and then **unlock**ed. Locking the door prevents it from either opening or closing.

(i) [3 marks] Draw the automaton of the **Door** specified above using only the events **open**, **close**, **lock** and **unlock**.

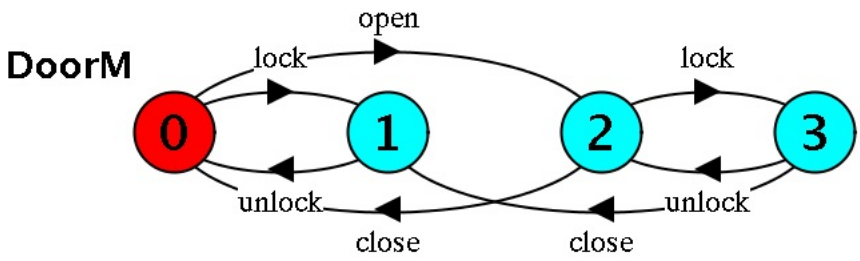


(ii) [3 marks] Specify the **Door** process using the LTSA language.

$$Door = (open- > Dou|lock- > Dcl),$$
$$Dou = (close- > Door|lock- > Dol),$$
$$Dol = (unlock- > Dou),$$
$$Dcl = (unlock- > Door).$$

DoorM is a small amendment to the previous **Door** model. In **DoorM** an open locked door can be closed, and when closed it remains locked.

(iii) [2 marks] Draw the automaton of the **DoorM** specified above using only the events **open**, **close**, **lock** and **unlock**.



(iv) [2 marks] Specify the **DoorM** process using the LTSA language.

$$DoorM = (open- > Dou|lock- > Dcl),$$
$$Dou = (close- > DoorM|lock- > Dol),$$
$$Dol = (unlock- > Dou|close- > Dcl),$$
$$Dcl = (unlock- > DoorM).$$

(c) Door specification

A door has a number pad and a lock. Initially the lock is unset. When the lock is unset the door can not be **open**ed or **close**d. When unset the event **setlock** can occur and will set the lock to a number from one to **N** (inclusive). Once the lock has been set the door only opens after some one **enter**s the number the lock is set to. Entering any other number results in an **error** message occurring after which the lock is returned to the unset state. Once open the door can **close**, and closing the door does not change the number the door has been set to.

Only use following events:

setlock - accepts a number parameter and sets the lock

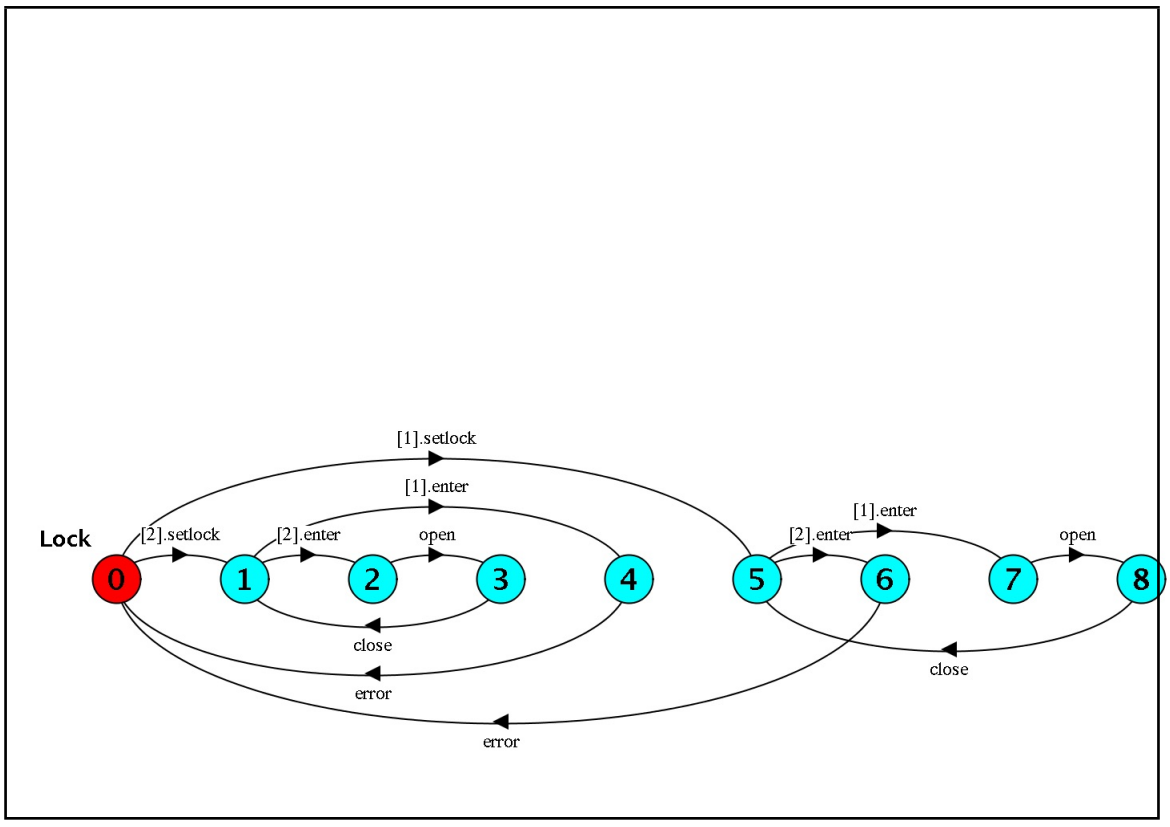
open - opens the door

enter - accepts a number parameter

close - closes the door

error - message

(i) [5 marks] Draw the automaton of the **Door** specified above when the constant **N = 2**.



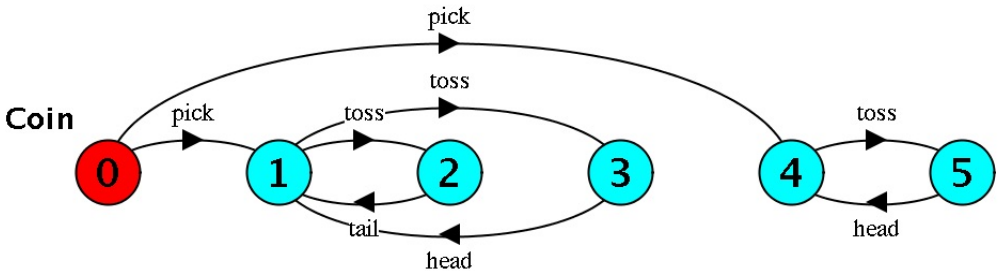
(ii) [5 marks] Specify the **Door** process using the LTSA language.

$$
\begin{aligned}
&Lock = L[0],\\
&\quad L[0] = ([i:1..N].setlock- > L[i]),\\
&\quad L[j:1..N] = ([i:1..N].enter- > Check[i][j]),\\
&\quad Check[i:1..N][j:1..N] = (when(i == j)open- > close- > L[j]\\
&\qquad\qquad\qquad\qquad\qquad\qquad |when(i! = j)error- > Lock).
\end{aligned}
$$

Question 3. Model Checking II [30 marks]

(a) Checking Requirements:

(i) [7 marks] Consider the following automaton.



Which of the following requirements does the process **Coin** satisfy :

	Answer **Yes** (satisfied) or **No** (not satisfied)
`progress T = {tail} .....`	No
`progress T = {head} .....`	Yes
`progress T = {tail,head} .....`	Yes
`progress T = {pick} .....`	No
`property S = (toss->(head->S|tail->S)).` `||Test = (Coin||S).`	Yes
`property Sh = (toss->head->Sh).` `||Testh = (Coin||Sh).` `.....`	No
`property St = (toss->tail->St).` `||Testt = (Coin||St).` `.....`	No

(ii) [1 mark]

	Answer **Yes** or **No**.
Will doing nothing satisfy all Safety requirements?	Yes
Will doing nothing satisfy all Liveness requirements?	No

(b) Write the LTSA commands that are needed for each of the following requirements.

(i) [1 mark] Button B must only be pushed after button A, and after button B is pushed button A must be pushed before button B is pushed a second time. After button A is pushed button B must be pushed before button A is pushed again

$property\ BA = (buttonA- > buttonB- > BA).$

(ii) [1 mark] It is always true that button B must only be pushed directly after button A, but button A can be pushed many times before button B is pushed.

$propertyBAA = (buttonA- > B),$
$B = (buttonB- > BAA|buttonA- > B).$

(iii) [1 mark] It is always true that both button B and button A can be pushed at some time in the future.

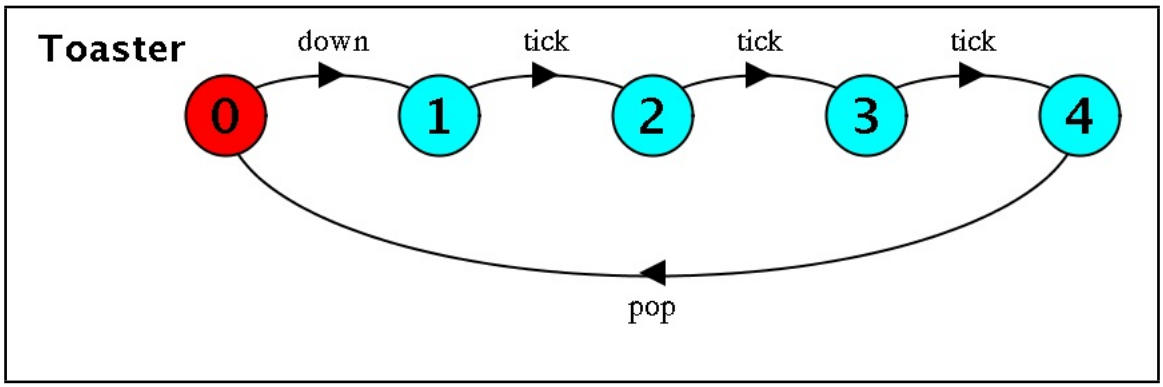
$progressA = \{buttonA\}$
$progress\ B = \{buttonB\}$

(iv) [1 mark] It is always true that either button B or button A can be pushed at some time in the future.

$progress\ AB = \{buttonA, buttonB\}$

(c) After a slice of bread is added to a **Toaster** a lever needs to be pushed **down**. **N** seconds after the lever has been pushed down the toast pops up (**popup**). The toaster is again ready for use after the toast has popped up. Each second is marked by a **tick** event. Use the events **down**, **tick** and **popup**.

(i) [3 marks] Draw the automaton of the **Toaster** process when **N=3**.

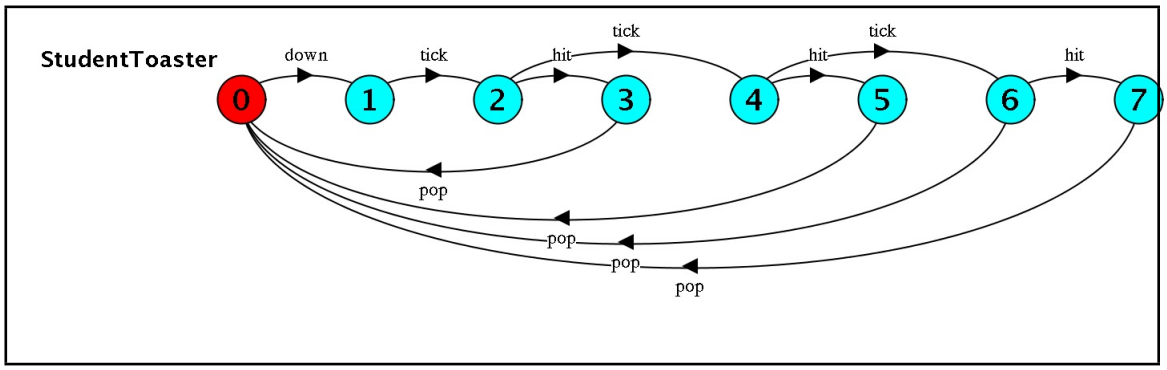


(ii) [3 marks] Specify the **Toaster** process using the LTSA language.

$$Toaster = (down- > Wait[0]),$$
$$Wait[t : 0..Time] = (when(t < Time)tick- > Wait[t+1]$$
$$|when(t == Time)pop- > Toaster).$$

(d) Hungry students soon discover that if they wait only one second they can **hit** their **StudentToaster**. Hitting the toaster makes the bread pop up immediately and the toaster is not broken.

(i) [2 marks] Draw the automaton of the **StudentToaster** process when **N=3**.



(ii) [2 marks] Specify the **StudentToaster** process using the LTSA language.

$$Toaster = (down- > Wait[0]),$$
$$Wait[t : 0..Time] = (when(t < Time)tick- > Wait[t+1]$$
$$|when(t == Time)pop- > Toaster$$
$$|when(t > 0)hit- > pop- > StudentToaster).$$

(e) Software engineering students decide to improve the toaster by reprogramming it to give them control of the time the bread is in the toaster.

Design a **TimedToaster** that can be set to toast bread for anything from 1 to **N** seconds (inclusive). If the time is unset the toaster defaults to one second. Your toaster must remember the time set and keep using this time until the time is set to a different value.

Use only the following events:

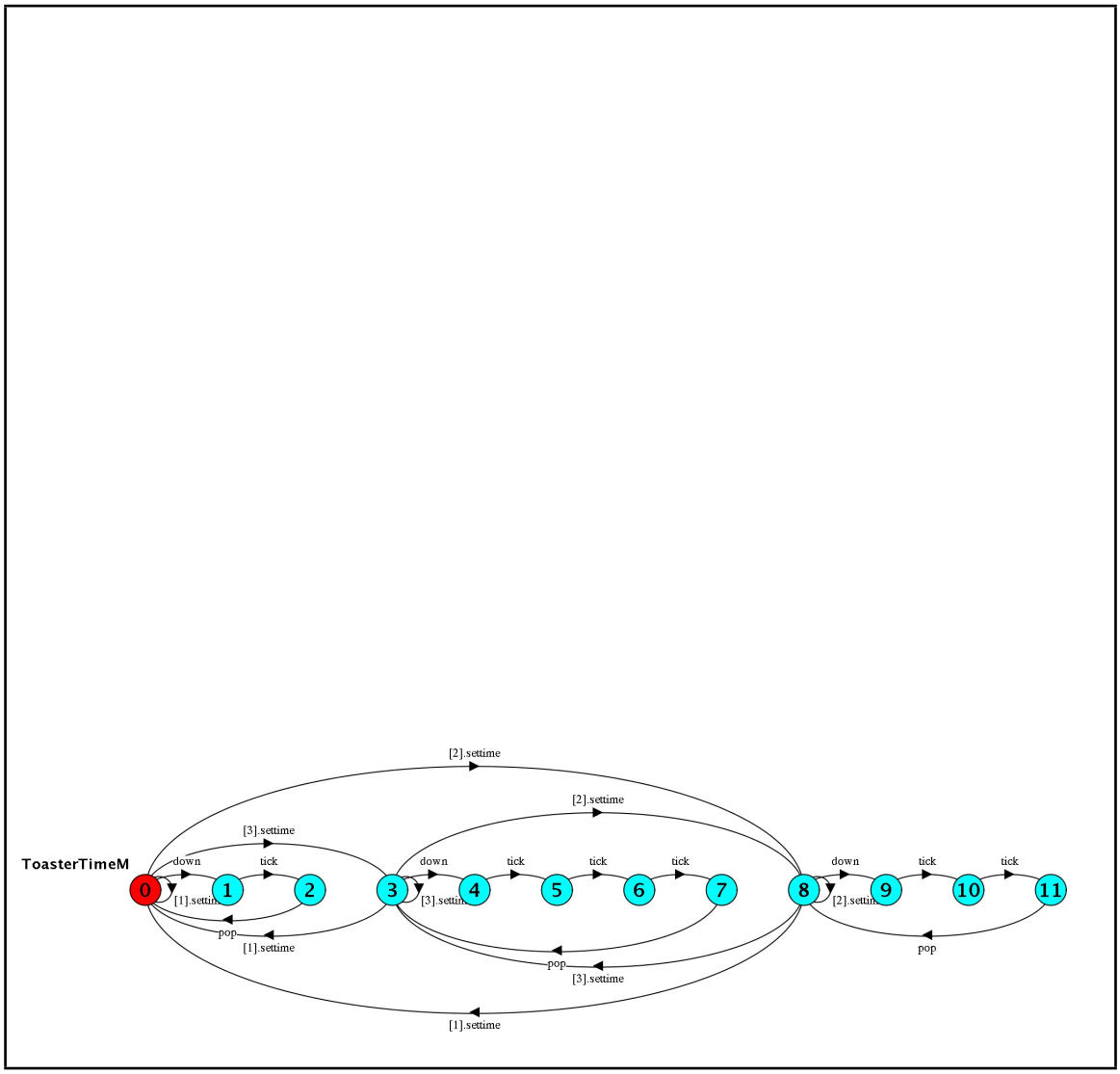
down - toast is put into the toaster

pop - toast is returned from toaster

tick - toast is cooked for one second

settime - number of seconds toast is cooked for is set

(i) [4 marks] Draw the automaton of the **TimedToaster** process when **N=3**.



(ii) [4 marks] Specify the **TimedToaster** process using the LTSA language.

$$
\begin{aligned}
&ToasterTimeM = S[1],\\
&\quad S[x:1..X] = ([y:1..X].settime- > S[y]\\
&\qquad |down- > Wait[x][X-x]),\\
&\qquad Wait[x:1..X][t:0..X] = (when(t < X)tick- > Wait[x][t+1]\\
&\qquad\quad |when(t == X)pop- > S[x]).
\end{aligned}
$$

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.

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.

Question 4. Specification & Verification II [30 marks]

(a) [6 marks] Next to each numbered comment in the box below, carefully and neatly write appropriate logical conditions which are true at that point in the program.

```
function max(int x8, int y8) -> (int r)
requires 0 <= x8 && x8 <= 255
requires 0 <= y8 && y8 <= 255
ensures r == x8 || r == y8
ensures r >= x8 && r >= y8:
    //
    // (1) 0 ≤ x8 ∧ x8 ≤ 255 ∧ 0 ≤ y8 ∧ y8 ≤ 255
    //
    if x8 < y8:
        //
        // (2) 0 ≤ x8 ∧ x8 < y8 ∧ y8 ≤ 255
        //
        x8 = y8
        //
        // (3) 0 ≤ x8' ∧ x8' < y8 ∧ y8 ≤ 255 ∧ x8 == y8
        //
    else:
        //
        // (4) 0 ≤ y8 ∧ y8 ≤ x8 ∧ x8 ≤ 255
        //
    //
    // (5) (0 ≤ x8' ∧ x8'< y8 ∧ y8 ≤ 255 ∧ x8 == y8) ∨ (0 ≤ y8 ∧ y8 ≤ x8 ∧ x8 ≤ 255)
    //
    return x8
```

(b) [4 marks] Briefly, describe what a *loop invariant* is.

A loop invariant is a condition which, intuitively, holds true at each step of a loop. In particular: (1) the loop invariant must hold on entry; (2) it must be restored after each iteration (assuming it held at the start of the iteration); (3) and, finally, it must hold when the loop terminates.

(c) Consider the following function `f()`

```
1  function f(int[] items, int item) -> (int r)
2  ensures r >= 0 ==> items[r] == item
3  ensures all { i in r+1 .. |items| | items[i] != item }:
4      //
5      int i = |items|
6      while i > 0:
7          i = i - 1
8          if items[i] == item:
9              return i
10     //
11     return -1
```

(i) [5 marks] In your own words, describe what the function `f()` does.

The function determines the largest index `i` where `items[i] == item`, or returns `-1` if no such index exists.

(ii) [4 marks] Provide an appropriate loop invariant for the function `f()`.

```
where i <= |items|
where all { k in i .. |items| | items[k] != item }
```

(d) [5 marks] In the box below, give the *weakest precondition* for the `trim()` function. You should include any loop invariants determined as part of this process.

```
function trim(int[] items, int n) -> (int[] result)
requires n >= 0 && n <= |items|


ensures |result| == n
ensures all { k in 0..n | items[k] == result[k] }:
    //
    int[] nitems = [0; n]
    int i = 0
    //
    while i < n
    where i >= 0 && |nitems| == n
    where all { k in 0..i | nitems[k] == items[k] }:


        nitems[i] = items[i]
        i = i + 1
    //
    return nitems
```

(e) Consider the following `add()` function:

```
function add(int[] items, int n) -> (int[] r)
requires true
ensures all { k in 0..|items| | r[k] == items[k] + n }:
    //
    int i = 0
    //
    while i < |items|:
        items[i] = items[i] + n
        i = i + 1
    //
    return items
```

(i) [2 marks] Provide one set of parameter and return values which are permitted by the postcondition, but which would never arise for any execution of the function.

`items=[0], n=1` gives `r=[1,2,3]`

(ii) [4 marks] Briefly, discuss why the postcondition given for the `add()` function is not the *strongest postcondition*.

The postcondition is not strongest because there are additional properties which the implementation guarantees that could be specified in the postcondition. In particular, `|r| == |items|`.

* * * * * * * * * * * * * * *

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.

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.

Appendix — LTSA

Defining Processes

The most Basic process is `STOP` the process that does nothing. The simplest process that does something is a single event is built by prefixing an event `takeTea` to the `STOP` process by the command:

```
Simple = (takeTea->STOP).
```

Every process can be represented by a transition labelled automata.

We can prefix a second event `Two = (teaButton->takeTea->STOP).`

In addition we can add the ability to choose one of two events:

```
A = (teaButton->takeTea->STOP|coffeeButton->takeCoffee->STOP).
```

this automata *branches* at the initial node.

To build events that do not terminate we can replace `STOP` with the name of the process we are defining:

```
Tt = (takeTea->Tt).
```

This endlessly performs the `takeTea` event, while:

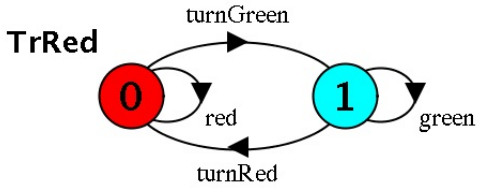
```
BT = (teaButton->takeTea->BT).
```

This endlessly performs `teaButton` followed by `takeTea`.

How to define a process

Given any automata we can always construct a process that is represented by the automata as folows:

1. name all nodes (or all nodes with more than one in and one out event) with a process name
2. define each of the processes and the choice of events leaving them
3. end each process definition with a comma except for the last process that must end with a full stop.



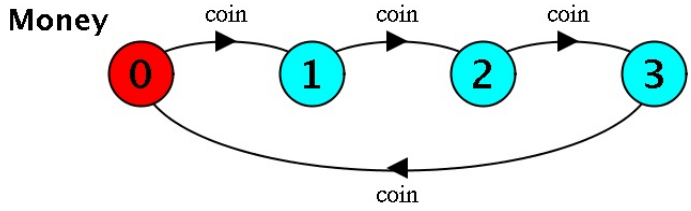
For the above automata node 0 we name `TrRed` and node 1 we name `TrGreen`. Then we define the events leaving these nodes

```
TrRed = (red->TrRed | turnGreen ->TrGreen),
  TrGreen = (green->TrGreen|turnRed->TrRed).
```

The result of this construction is the definition of the first process `TrRed`, all other processes, in this case just `TrGreen`, are *local* definitions.

Indexed Process

We can define a process consisting of an an unknown number of events all in a loop. To do this we must index both the (local) processes and the events.



The first thing we do is define a constant to be used for the size of the automata to be constructed:

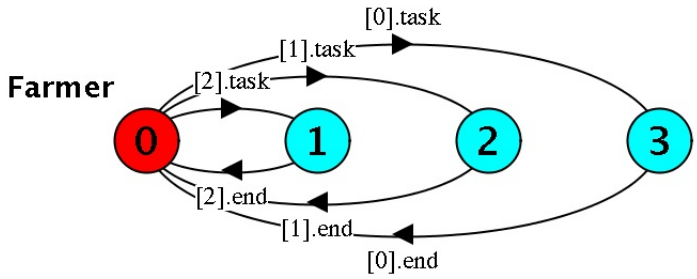
```
const N = 4
```

Next the definition `C[i:1..N] =` defines the `N` processes `C[1]`,`C[2]`,`C[3]` and `C[4]`

```
Money = C[1],
C[i:1..N] = (when(i<N) coin->C[i+1]
            |when(i==N) coin->C[1]).
```

The right hand side of the definition defines guarded events, that is `when(i<N) coin->C[i+1]` will only add event `coin` that ends at node `C[i+1]` when the index is less than constant `when(i<N)`. **Note a guard only applies to one event.** Each time you add a choice you need to add any required guard.

You can also index events and choice. Consider a `Farmer` that has to handout one of `N tasks` and then wait for the task to `end`.



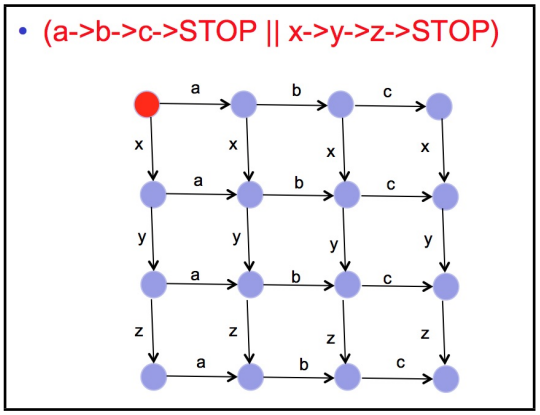
This is defined by:

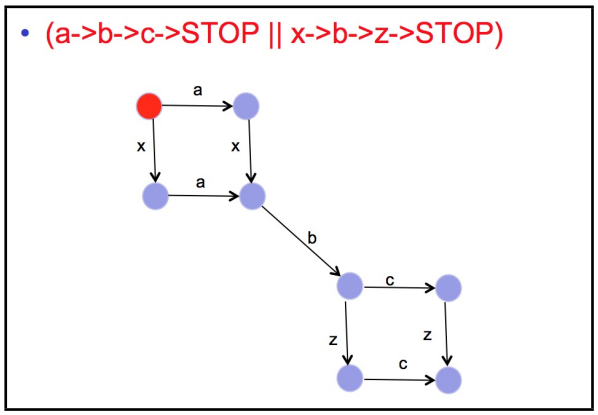
```
Farmer = ([i:0..N].task ->W[i]),
   W[i:0..N] = ([i].end->Farmer).
```

Defining Concurrent Processes

Below left we have two processes each with three events and no two event have the same name hence the event from each process can be **interleaved** in any way.

In the LTSA tool events from different concurrent processes that have the same name must synchronize and only these events synchronize. That is neither process can execute the synchronising event on its own. These synchronising events are only executed when both processes are ready to execute them. Below right only differs from below left in that the second event in both processes has the same name and hence must synchronize.





Event synchronization is the only mechanism for concurrent processes to interact and because of event synchronisation we have:

> **If you can see an event you can synchronize with it and you can block it.**

Without synchronization two processes are independent and hence their events interleave and the state space of the composition of the processes is the product of the state space of the constituent processes.

Hence the only way to control the order of two events from concurrent processes is to introduce a synchronizing event. In above the `a` event and the `z` event are from different concurrent processes in the left hand interleaving example either could occur first. Whereas in the right hand example the synchronization of the `b` event forces the `a` event to occur before the `z`.

The effect of the synchronization of the **b** events in the right hand processes is to reduce the size of the reachable state space of the automata. Note the first two events **a** and **x** can be performed in either order but only when both **a** and **x** have been performed and both processes are ready to perform **b** does the **b** event actually get performed.

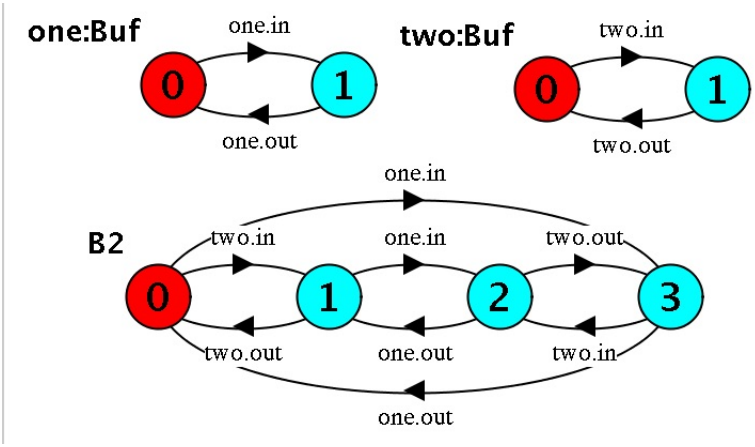
Labelling Processes

In the following example we make use of a one place buffer `Buf` is a process that when empty can receive some thing `in` and when full can return it `out`. By labelling processes `one:Buf` the tool labels all events in the process `one.in` and `one.out`.

Using process labelling we can make two differently label copies of a process and compose them in parallel to build the interleaving of the two copies.

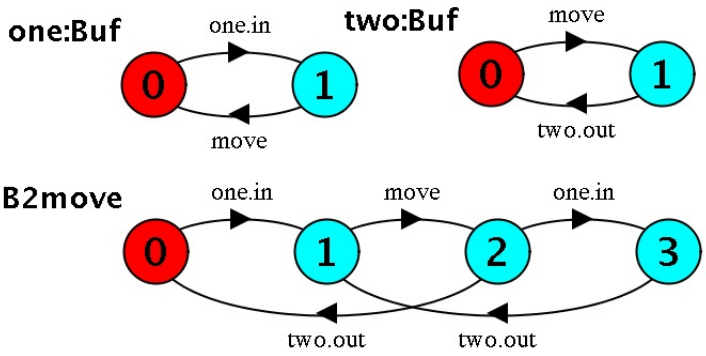
When, in the LTSA tool, we build a process by concurrently composing processes the name of a concurrent process must begin with `||`

```
||B2=(one:Buf||two:Buf).
```



We can force the synchronisation of the output from buffer `one` with the input to buffer `two` by event relabelling

```
||B2=(one:Buf||two:Buf)/{move/one.out,move/two.in}.
```



Note that the result is much simpler than the interleaving as the `move` event now can only occur when **both** buffers are able to perform it. We can go further and hide the `move` event

```
||B2=(one:Buf||two:Buf)/{move/one.out,move/two.in}\{move}.
```

The `move` event becomes a `tau` event that can not be synchronized with nor can be blocked. The `tau` events can be removed by **minimizing**. Hiding is commonly used to make communication private.

> **Pragmatically when you compose two processes in parallel you should look at both automata and check the names of the events you want to synchronise.**

Indexing concurrent processes.

If you want `N Worker` processes, each labeled with `[1],[2],...[N]`

```
||Workers = (forall [i:1..N] ([i]:Worker)).
```

Safety and Liveness Properties

Processes, when small, can be understood when visualized as an automata but as the size of the process grows this becomes increasingly difficult. We can automatically model check some simple process requirements. These requirements can be divided into two groups:

Safety requirements - **nothing bad happening** (LTSA- `property`)

Liveness requirements - **some thing good happening** (LTSA- `progress`)

The LTSA tool can check both types of requirements and will return a trace to the state in which the error occurs. Doing nothing satisfies any safety requirements but fails liveness requirements. So if you are not sure if a requirement is a safety or a liveness requirement:

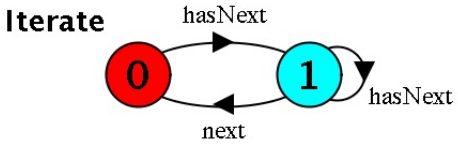
> **ask yourself will this requirement be satisfied by doing nothing?**

The LTSA tool we are using provides two commands `property`, to specify safety requirements and `progress` to specify liveness requirements. The dropdown menu *Check* has options to evaluate the requirements.

Safety is a process, over a restricted alphabet, that defines the desired safe behaviour. Any other behaviour, over the restricted alphabet, will lead to an ERROR.

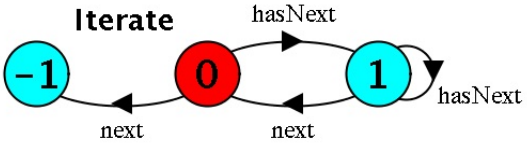
In Java the safe use of an iterator is to never call `next` before calling `hasNext`. This safe behaviour we can define as behaving as the automata:

```
Iterate = (hasNext -> X),
      X = (hasNext -> X | next ->
```



The `property` command below changes the automata by adding the error state (-1) along with any event needed to stop the automata from blocking any event in any state, other than the error state.

```
property Iterate = (hasNext -> X),
      X = (hasNext -> X | next ->
```



To check if a process `UnderTest` makes use of iterators safely you must compose it with the property and then run the check on the combination:

```
||TestIterator = (UnderTest||Iterate).
```

A safety `property` is a process that acts like an observer. That is it does not block any event of the process under test until it has violated the safety property and then it reports the error.

Progress command defines a set of events, one of which must always be reachable, no matter what state you are in.

The command

```
progress HeadOrTail = {head,tail}
```

defines the liveness requirement that: it is always possible for one of `head` or `tail` to eventually occur. Both a fair coin or a double headed coin with satisfy `HeadOrTail`. If you want to distinguish fair coins from double sided coins you need the two commands:

```
progress Head = {head}
```

to detect the double tailed coin and

```
progress Tail = {tail}
```

to detect the double headed coin.

Syntax

There are always many ways to define any interesting automata but some simple examples should help.

	atomic	indexed
Prefixing	`A = act->P`	`if (i<N) then (act[i]->P[i+1]) else P[0]`

		`Money = C[1],`
		`C[i:1..N] = (when(i<N) coin->C[i+1]`
		`|when(i==N) coin->C[1]).`

Choice	`A = a->P|b->Q`	`Farmer = ([i:0..N].task ->W[i]),` `W[i:0..N] = ([i].end->Farmer).`
Labelling	`lab:P`	see below
Parallel	`||A = (P||Q)`	`||Workers = (forall [i:0..N] ([i]:Worker)).`
Relabelling	`P/{new/old}`	`P/{new[i:0..N]/old[i]}`
Hiding	`P\{act}`	`P\{act[i:0..N]}`

For requirements checking we have:

Safety	`property Iterate = P.`
Liveness	`progress HeadOrTail = {head,tail}.`