TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VUW VICTORIA UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2016

TRIMESTER 2

SWEN224

Software Correctness

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.
Model Checking Documentation is in the Appendix

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 a separate answer booklet.

Question	Topic	Marks
1.	Static Analysis	30
2.	Specification & Verification	30
3.	Model Checking I	30
4.	Model Checking II	30
	Total	120

Question 1. Static Analysis [30 marks]

(a) Consider the following program written in Java, which contains an error:

```
// Return the absolute difference of two integers
int diff(int x, int y) {
    int r;

    if(x < y) { r = x - y; }
    else {
        r = y - x;
    }
    assert r >= 0;

    return r;
}
```

(i) [2 marks] Give values for parameters `x` and `y` which would cause the **`assert`** to fail.

x=1, y=0

(ii) [2 marks] Suggest a simple fix for the above program. You can ignore the possibility of *integer overflow*.

Swap condition from `x < y` to `x > y`

(iii) [5 marks] Java **`assert`**s operate at *runtime*. Briefly, discuss the disadvantages of this, compared with techniques which operate at *compile time*.

- Operating at runtime means that problems can only be found for the limited number of inputs that are tested.
- One cannot possibly check all possible inputs and, hence, problems may still exist for untested inputs.
- Compile time techniques check for problems before the program is run, and can guarantee the absence of certain classes of error. In effect, they check for all possible inputs.

(b) This question is concerned with *non-null* analysis.

(i) [2 marks] Briefly, discuss what the `@NonNull` annotation means.

This means that the annotated variable or field cannot hold the **`null`** value

(ii) [2 marks] Briefly, discuss whether or not `@NonNull` is a *subtype* of `@Nullable`.

Yes, `@NonNull` is a subtype of `@Nullable`. This is because the set of values represented by `@NonNull` is a *subset* of that represented by `@Nullable`

(iii) [6 marks] For each *parameter*, *return* and *field* in the following program, insert `@NonNull` or `@Nullable` annotations (where appropriate) by writing in the box.

```
public class Property {
   private @NonNull String name;       // Every property has a name

   private @Nullable Player owner;      // Some properties have owners

   private boolean mortgaged; // Some properties are mortgaged

   public Property(@NonNull String name) {
       this.name = name;
   }

   public @NonNull String getName() { return name; }

   public @Nullable Player getOwner() { return owner; }

   public void setOwner(@Nullable Player p) { owner = p; }

   public boolean isMortgaged() { return mortgaged; }

   public void setMortgated(boolean m) { mortgaged = m; }
}
```

(c) This question is concerned with Java's *definite assignment* analysis.

(i) [2 marks] Briefly, state what the purpose of checking definite assignment is.

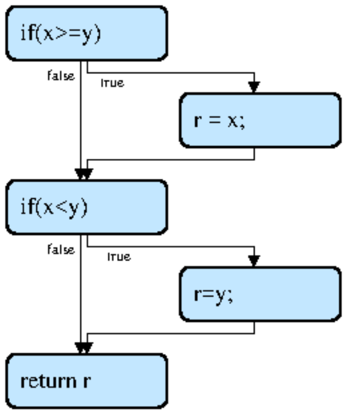
The purpose of definite assignment checking is to ensure that every local variable is defined a value before being used in an expression.

Consider the following program written in Java.

```
int max(int x, int y) {
    int r;
    //
    if(x >= y) { r = x; }
    if(x < y) { r = y; }
    //
    return r;
}
```

(ii) [5 marks] This program fails *definite assignment*. Briefly, discuss why this happens. Your answer should illustrate the program's *control-flow graph*.

The definite assignment checker in Java makes a decision as to whether or not a variable is definitely assigned based purely on the method's control flow graph. In particular, it completely ignores the conditions on the if statements above. When looking at the control-flow graph, the checker observes a path through it where both conditions are false. It simply assumes this path is feasible and, hence, that r may be undefined at the return statement.



(iii) [4 marks] The above program is a *false positive* with respect to definite assignment analysis. Briefly, discuss what this means.

A false positive occurs when a static analysis tool reports an error when, in fact, no such error is actually possible. In the above example, the definite assignment checker is reporting that variable `r` may be undefined a the **`return`** statement. We can see that this is impossible by considering the four possible paths through the program (TT, TF, FT, FF) and observing that both TT and FF are infeasible. False positives reflect the conservative nature of the definite assignment checker, which is preferable to it permitting false negatives (i.e. failing to report errors which actually do exist).

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. Specification & Verification [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 set(int[] items, int i, int val) -> (int[] r)
requires 0 <= i && i < |items|:
    //...
```

Meets preconditon: `u8=0`

Does not meet precondition: `u8=-1`

(ii) [2 marks]

```
function indexOf(int[] items, int val) -> (int r)
requires some { i in 0..|items| | items[i] == val }:
    //...
```

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

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

(iii) [2 marks]

```
function reverse(int[] bs) -> (int[] r)
requires all { i in 0..|bs| | bs[i] >= 0 && bs[i] <= 255 }:
    //...
```

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

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

(iv) [2 marks]

```
function find(int[] xs, int val) -> (int r)
requires |xs| > 1 ==> all { i in 1..|xs| | xs[i-1] < xs[i] }:
    //...
```

Meets preconditon: `xs=[0,1,3]`

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

(b) Consider the following implementation for the function `cmp()`:

```
function cmp(int[] left, int[] right, int i) -> (int r):
    if i == |left|:
        return 0
    else if left[i] < right[i]:
        return -1
    else if left[i] > right[i]:
        return 1
    else:
        return cmp(left,right,i+1)
```

(i) [2 marks] Briefly, describe in your own words what function `cmp()` does.

The `cmp()` function finds the first position `i` in which both arrays differ and indicates whether `left[i]` is below `right[i]` at that point. If no such position exists, then it returns `0` to indicate they are equal.

(ii) [8 marks] Provide an appropriate specification for function `cmp()`.

```
function cmp(int[] left, int[] right, int i) -> (int r)
requires |left| == |right|

requires i >= 0 && i <= |left|

ensures r >= -1 && r <= 1

ensures r == 0 ==> all { j in i..|left| | left[j] == right[j] }

ensures r < 0 ==> some { j in i..|left| | left[j] < right[j]
                        && all { k in 0..j | left[k] == right[k] } }

ensures r > 0 ==> some { j in i..|left| | left[j] > right[j]
                        && all { k in 0..j | left[k] == right[k] } }:
```

(c) Consider the following implementation for the function `cut()`:

```
function cut(int[] xs) -> (int[] rs)
ensures |rs| == |xs|
ensures all { k in 0..|xs| | xs[k] >= 0 ==> rs[k] == xs[k] }
ensures all { k in 0..|xs| | xs[k] < 0 ==> rs[k] == 0 }:
    //
    int i = 0
    int[] ys = xs
    //
    while i < |xs|:
        if xs[i] < 0:
            ys[i] = 0
        else:
            ys[i] = xs[i]
        i = i + 1
    //
    return ys
```

(i) [2 marks] Briefly, describe in your own words what function `cut()` does.

The cut function takes an array `xs` and returns an updated array where all negative elements are replaced with `0`, and all other elements are left as is.

(ii) [6 marks] Provide an appropriate *loop invariant* for function `cut()`.

```
    while i < |xs|
    where i >= 0 && |xs| == |ys|
    where all {j in 0..i | xs[j] < 0 ==> ys[j] == 0 }
    where all {j in 0..i | xs[j] >= 0 ==> ys[j] == xs[j] }:
```

(iii) [4 marks] Briefly, state the *three rules* of loop invariants.

The three rules of loop invariants are:

1. **Loop invariants must hold on entry**. That is, they must hold before the first iteration of the loop begins.
2. **Loop invariants must be restored after each iteration**. That is, they must hold at the end of each iteration of the loop (assuming they held at the start of each iteration).
3. **Loop invariants must hold on exit**. This follows from rule 2 but, in addition, when the loop condition is false, the loop invariant must still hold.

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 3. Model Checking A Lift [30 marks]

(a) The LTSA language:

(i) [6 marks] Draw the automaton of the following three processes:

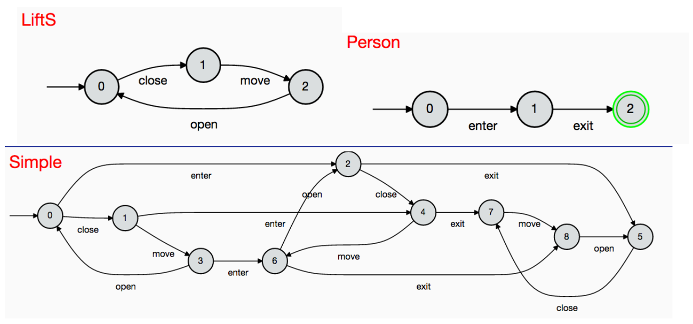
automata {

LiftS = (close →move→open→LiftS).

Person= (enter→exit→STOP).

Simple = LiftS || Person.

}

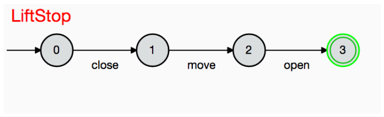


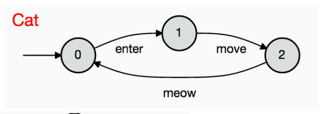
(ii) [2 marks] Briefly explain what is meant by event interleaving.

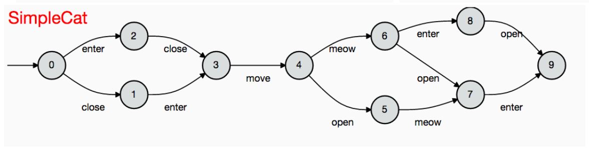
When two processes are run in parallel and do not interact in any way then an observer can see any interleaving of the events from the two processes. That is there is no fixed order between any two events from different processes.

(iii) [6 marks] Draw the automaton of the following three processes:

```
automata {
LiftStop = (close →move→open→STOP).
Cat= (enter→move→meow→Cat).
SimpleCat = LiftStop || Cat.
}
```







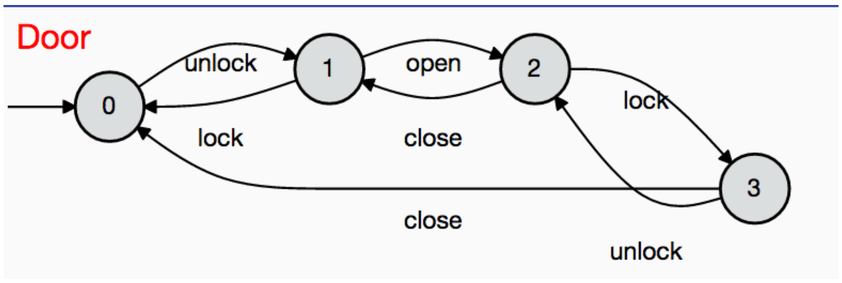
(iv) [3 marks] Briefly explain what is meant by event synchronization.

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 locked. 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 opening but not from closing.

(i) [5 marks] Draw the automaton of the **Door** specified above using only the events **open**, **close**, **lock** and **unlock**.

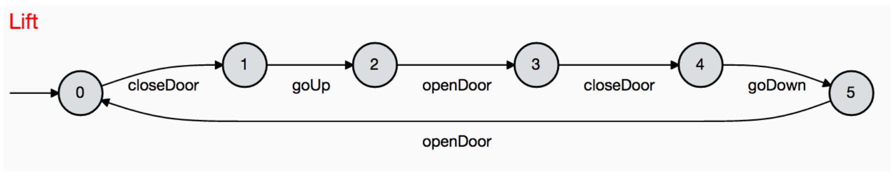


(ii) [3 marks] Specify the **Door** process using the LTSA language.

automata { Door = unlock →CU,
CU = open → OU—lock → Door,
OU = close → CU — lock → OL,
OL = close → Door — unlock →OU .
}

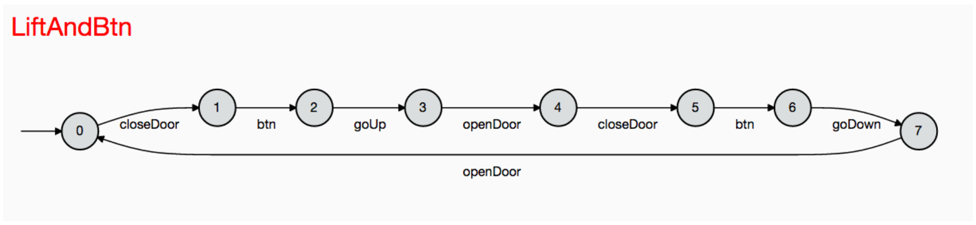
(iii) [5 marks] Process Lift is:

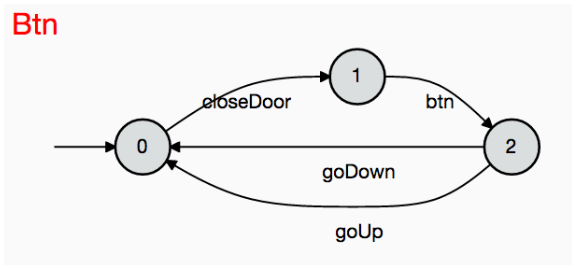
Lift = closeDoor→goUp→openDoor→closeDoor→goDown→openDoor→Lift.



A process Btn may synchronise with any of the events in Lift.

Specify and draw the process Btn **so that** when composed in parallel with Lift they behave like LiftAndBtn = Lift || Btn. shown below.





Btn = closeDoor→ btn →(goDown →Btn|goUp→Btn).

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.

Question 4. Model Checking General Processes [30 marks]

(a) Worker Processes:

(i) [5 marks]

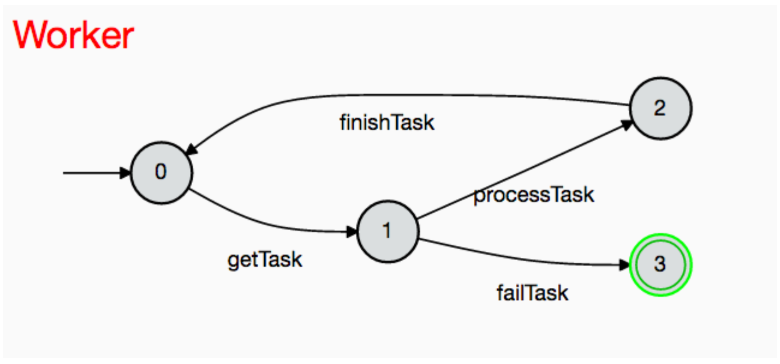
A Worker process gets a task then either fails the task after which the process stops or executes the task. After finishing the task the Worker returns to the start to wait for the next task.

Use only the events: getTask, executeTask, failTask and finishTask

Specify the automaton of the Worker process:

Worker = getTask → (executeTask → finishTask → Worker | failTask → STOP).

Draw the automaton of the Worker process:

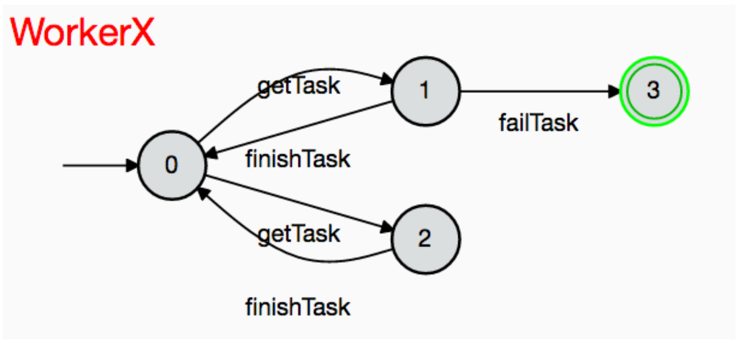


(ii) [5 marks] The WorkerX process that is built from the Worker process above by hiding the executeTask event.

Specify the WorkerX process.

```
  W = Worker\ {executeTask}.
WorkerX = abs(W).
```

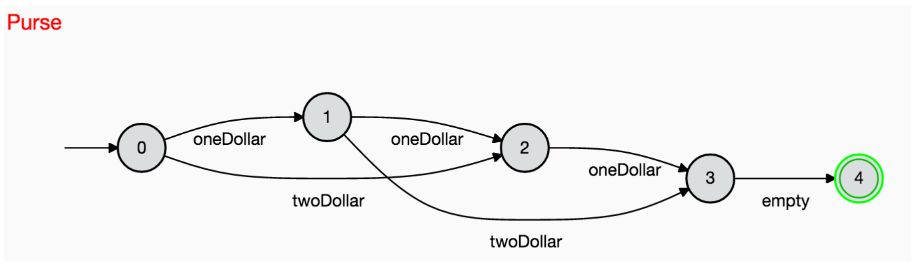
Draw the WorkerX process.



(b) Simple Purse specification

A **Purse** initially has N dollars and when it has the funds is able to pay either one or two dollars. When the Purse is empty it can be refilled.

(i) [4 marks] Draw the automaton of the **Purse** for the special case when N = 3. Use only the events **oneDollar**, **twoDollar**, and **refill**.



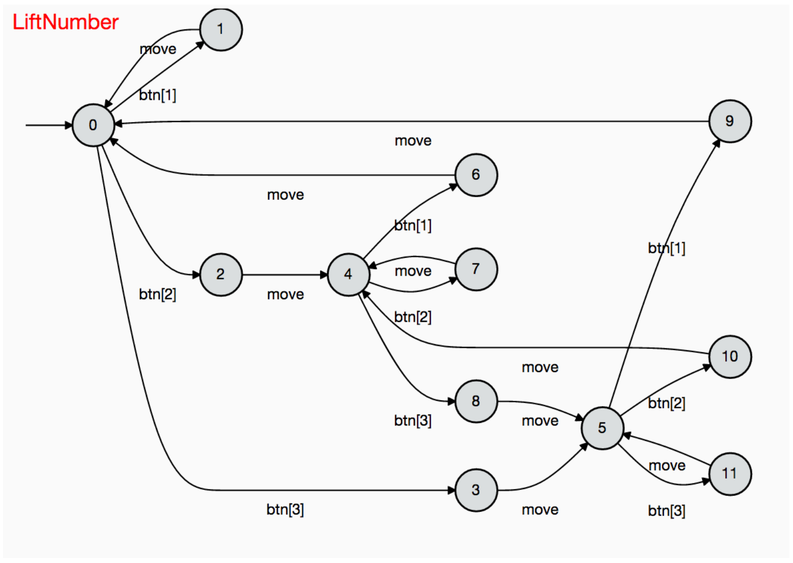
(ii) [5 marks] Specify the **Purse** process using our Process language.

```
  Purse = P[N],
P[i:0..N] = when (i > 0) oneDollar → P[i-1]
        | when (i > 1) twodDollar → P[i-2]
        | when i == 0 empty → STOP.
```

(c) Lift to N floors

A Lift called LiftNumber can be on any one of N floors. There are N buttons, after pushing button j the lift moves to floor j

This figure is an example when N = 3.

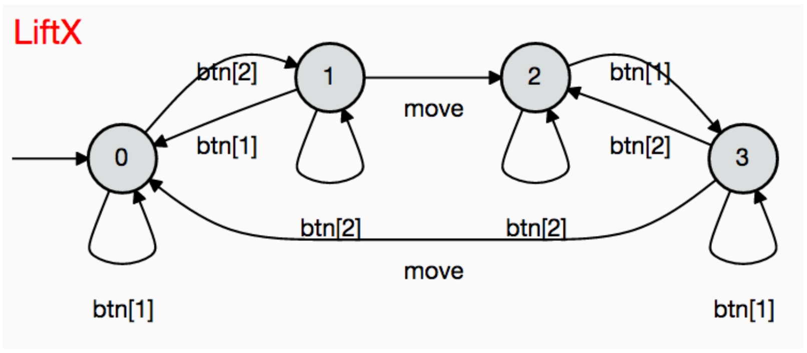


(i) [5 marks] Specify the LiftNumber process using parameter N and only using the events named in the figure.

```
  LiftNumber = Ln[1],
Ln[i:1..N] = btn[j:1..N] → move → Ln[j].
```

(ii) [6 marks] Specify LiftX a Lift that is like LiftNumber above except when you push the button for the floor that you are on then the lift will not move.

The figure is for the case when N = 2



```
Lift X= L[1][1],
        L[i:1..N][f:1..N] = btn[j:1..N] → L[i][j]
                       | when (i!=f) move → L[f][f].
```

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

Defining Basic Processes

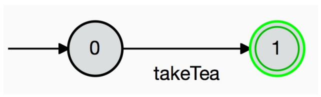
Processes are defined using a simple process algebra $\Sigma = \{Act, - >, |, STOP\}$ the operational semantics of the defined process definition will be rendered as an automata by enclosing the definition in `automata { ... }` as shown in the examples below. We will frequently omit the `automata { ... }` as any number of process definitions may be included with in the brackets `{...}`.

A simplest process is `STOP` the process that dose nothing. The more interesting but very basic processes that we discuss consist of a finite state space and transitions labeled with atomic events.

Event prefixing

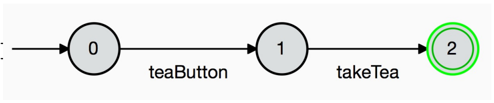
A simple process that performs a single event and stops can be built by prefixing an event `takeTea` to the `STOP` process using the `->` operator by the command:

```
automata {
Simple = (takeTea->STOP).
}
```



Every process we define can be represented by a transition labeled automata. The events, like `takeTea`, have an informal meaning (semantics) given by relating them to some real world event. We can prefix a second event

```
Two = (teaButton->takeTea->STO
```

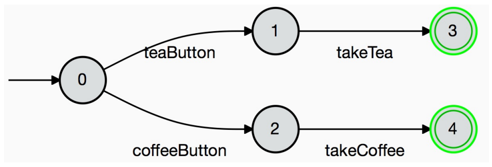


The informal meaning of events will in part be formalised by our definition (to be given later) of parallel composition. Informally we need to think of our events as *hand-shake events*, i.e. event that can be blocked or enabled by the context in which they execute. For example the `teaButton` event of a vending machine can only occur when some agent actually pushes the button, which can also be modelled by a `teaButton` event.

Event choice

A vending machine that has two buttons one for coffee the other for tea offers the user the *choice* to push either button. This we formalise by intrducing the *choice operator* _|_.

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

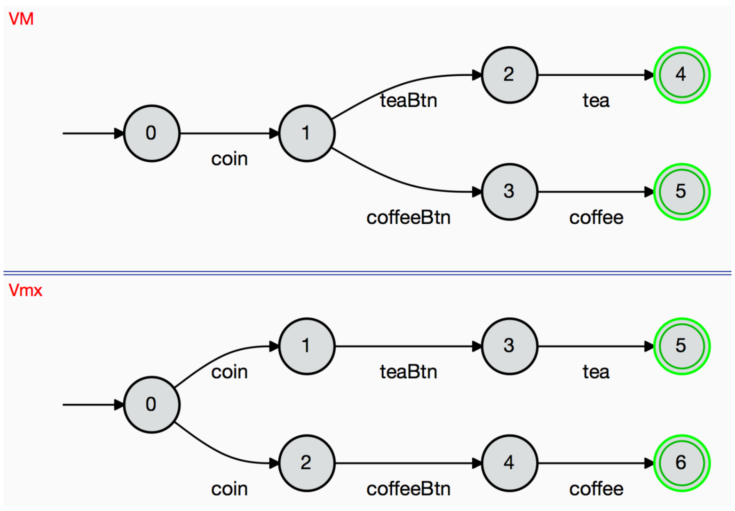


this automata *branches* at the initial node.

Non deterministic processes

The two processes VM and VMx both represent a vending machine that offers two drinks, tea and coffee after a coin is inserted. The two terms are different and they are represented by different automata.

```
automata
{
VM = coin->((teaBtn->tea-> STOP)|(coffeeBtn->coffee->STOP)).
Vmx = (coin->teaBtn->tea-> STOP)|(coin->coffeeBtn->coffee->STOP).
}
```



Are VM and VMx equivalent processes? Before you can answer this you must decide what it means for two processes to be equivalent and there is many reasonable answers to this. If we assume either that the processes generate events or that they ae used to recognise a sequence of events then the processes can reasonable be viewed as equivalent as both generate (recognise) the same two event sequences:

1. `coin,teaBtn,tea`
2. `coin,coffeeBtn,coffee`

But what if you were interacting with these processes and you wanted `coffee` then with the first machine you could allways insert a `coin` than push the `coffeeBtn` and you would be able to get your `coffee`. In contrast with the second machine after inserting the `coin` you would not be able to push the `coffBtn`. Hence you would be able to distinguish the two processes.

Event hiding and process simplification

We can make event private by hiding them so they can not be seen. `_\{t}\}` operator renames the t event to a tau event and the `abs(_)` operator abstract away the tau events.

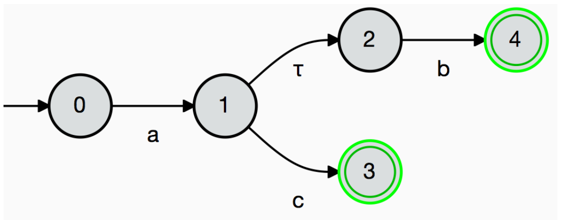
Abstraction works by adding observable events $x \xrightarrow{a} y$ whenever:

1. there exists v such that $x \xrightarrow{a} v$ and $v \xrightarrow{tau} y$ or

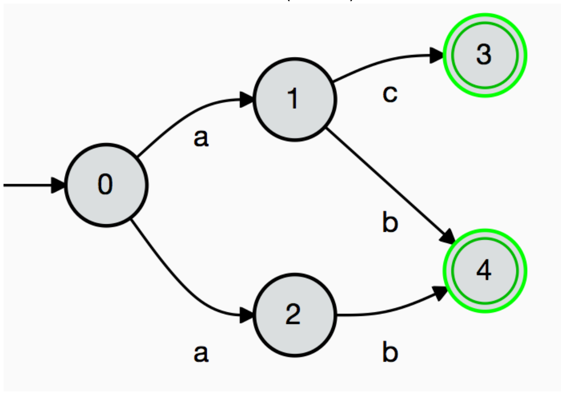
2. there exists v such that $x \xrightarrow{tau} v$ and $v \xrightarrow{a} y$

See following example:

`Basic = a->(t->b->STOP | c->STOP).`



```
Bas = Basic\{t}.
 B = abs(Bas).
```



Nonterminating processes

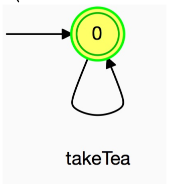
We call the set of know processes is called the *Process name space*. Initially the *Process name space* is `{STOP}`.

Processes consist of a set of states, an initial state and a set of event labeled state transitions. Given a process has a set of states and a set of transitions it is reasonable that the process can be *conceptual identified* with its initial state. Each process definition `P1 = ...` adds the the defined process `P1` to the name space.

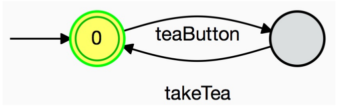
Clearly any state `S` could also be *conceptual identified* with the the process consisting of the same set of states and transitions but with initial state `S`. This we use to define nonterminating processes simply by allowing any valid process to be used where `{STOP}` has been used:

To build events that do not terminate we can replace `STOP` with the name of the process we are defining thus `T = (takeTea->STOP).` becomes `Tt = (takeTea->Tt).` and the new process `Tt` endlessly performs the `takeTea` event.

`Tt = (takeTea->Tt).`

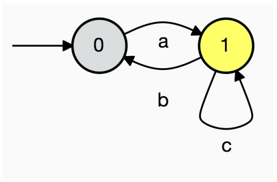


`BT = (teaButton->takeTea->BT).`



We allow *local process* or states to be defined within a process definition by separating definitions with a comma. The local process do not appear in the Process name space.

```
P = (a->Q ),
  Q = (b->P|c->Q)
```

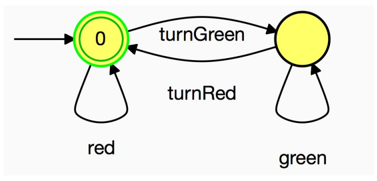


This allows processes to be defined without cluttering the *Process name space*.

Translating any finite state automata into a process term

It is often easy to sketch your understanding of a processes behaviour as an automata. Then from any automata we can construct the process term from which represents the automata and from which our tool will generate the automata. This can be achieved quite mechanically as follows:

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

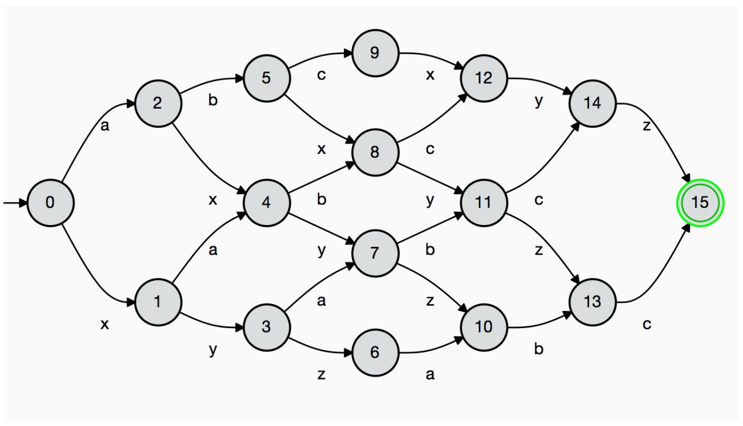
```
TrRed = (red->TrRed | turnGreen ->TrGreen),
  TrGreen = (green->TrGreen|turnRed->TrRed).
```

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

Defining Concurrent Processes

Below 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.

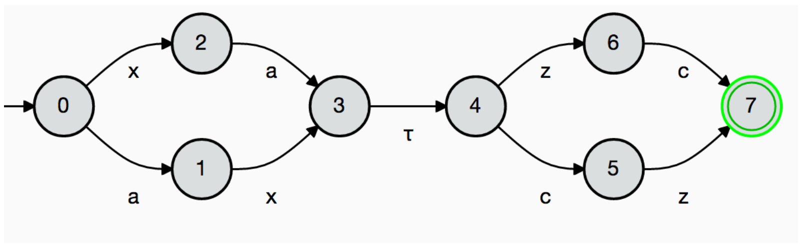
```
P = ((a->b->c->STOP) || (x->y->z->STOP)).
```



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.

In the Process 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 only differs from the previous process in that the second event in both processes has the same name and hence must synchronize and the resulting `m` event is then hidden (renamed τ).

```
P = ((a->m->c->STOP) || (x->m->z->STOP))\{m}.
```



Event synchronization is the only mechanism for concurrent process interaction and because of event synchronisation we know:

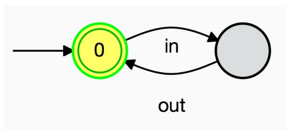
If you can see and event you can synchronize with it and you can block it.

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

Another effect of synchronization 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** dose the **b** event actually get performed.

Labeling Processes

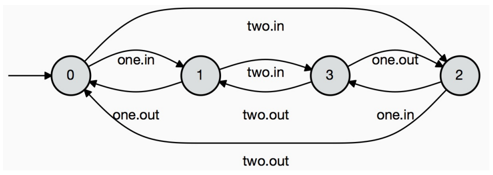
In the following example we make use of a one place buffer `Buf` is a process that when empty can receive some thing `in` and when full can return it `out`.



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.

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



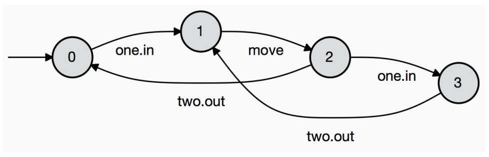
Event renaming

If two events from processes run in parallel have the same name they, and only they, must synchronise.

Pragmatically when you compose two processes in parallel you should check the name of events you want to synchronise and where necessary rename them to enforce the desired synchronisation.

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

```
B3 = (one:Buf/{move/one.out}||two:Buf/{move/two.in}).
```

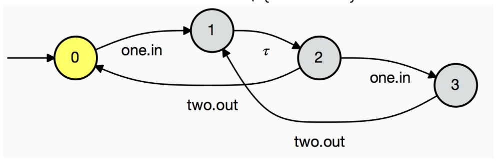


Note that the result is much simpler than the interleaving as the `move` event now can only occur when **both** buffers are able to perform it.

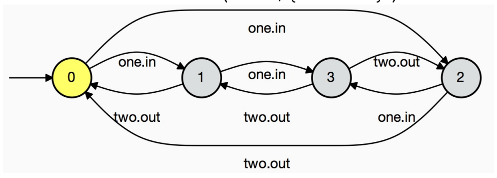
Event hiding and process simplification

We can go further and hide the `move` event by applying `_\{move}` The `move` event becomes a `tau` event that can neither be synchronized with nor blocked.

`B4 = B3\{move}.`

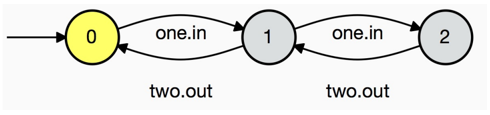


`B5 = abs(B3\{move}).`



The `tau` events can be removed by **abstraction**, (the application of `abs(_)`) otherwise known as building the *observational* semantics. With a little effort nodes, 1 and 2 in `B5` can be seen to be essentially the same. They are actually bisimular but we will not be going into details here. These nodes can be identified to produce a simpler but equivalent automata by the application of `simp(_)`.

`B6 = simp(abs(B3\{move})).`



Event hiding is commonly, but not exclusively, used to model private communication.

Indexed Process definitions

Basic process definitions you have seen so far a fixed bounded set of states. This suits some situations very well and allows easy and complete push button model checking. When what you are modelling has infinite state you could use symbolic model checking but this frequently requires input from a domain expert and is very time consuming.

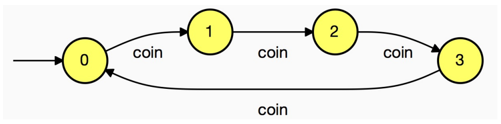
The small world assumption

Most program bugs can be found while restricting variables to range over a small domain. Using this assumption we model processes with variables by indexing the processes and restricting the indexes to range over a small domain. Having done this the variables in the state can be removed by instantiating the varaibles with values from the small domain.

Indexing introduces the ability to define a process parameterised by one or more index. Once the indexes are fixed you are back to a basic process with a fixed set of states. Processes can be indexed in different ways to achieve conceptually different things. The first we consider is how to build a process of parameterised size, the second is to model events that input or output data and finally how to model a parameterised number of concurrent processes.

State indexing

We can define a process consisting of an an unknown number of states. To do this we must index the local states (or local processes).



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]`

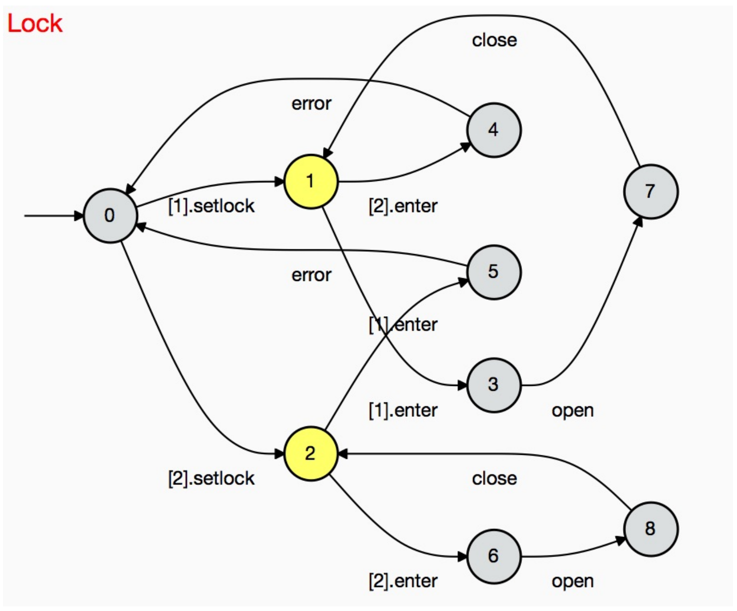
```
automata {
Money = C[1],
C[i:1..N] = (when(i<N) coin->C[i+1]
            |when(i==N) coin->C[1]).
}
```

On the right hand side of the indexed definition we define 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 that the 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.

Event indexing

An indexed event can be used to model events with data I/O.

You can also index events and choice. Consider a `Lock` that has to be set to a value between `1` and `N` and once set the door only opens if the same value is entered by user. If the wrong value is entered the lock needs to be reset. To help make sense of this design you can think of the `setlock` event as needing admin privileges (not modeled).



This is defined by:

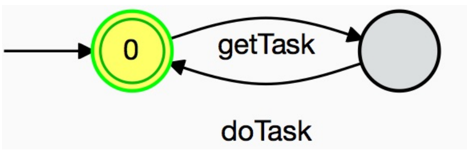
```
Lock = ([i:1..N].setlock -> L[i]),
L[j:1..N] = ([i:1..N].enter ->
                              ( when (i==j)open ->close->L[j]
                               |when(i!=j) error->Lock)).
```

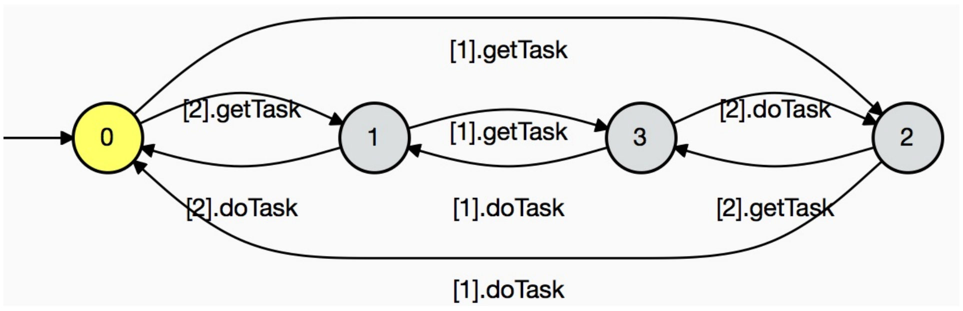
Not the value input in the `[i:1..N].setlock` event is stored in the state of the process `L[i]` for subsequent comparison with the value input in the `[i:1..N].enter` event.

Indexing concurrent processes.

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

```
Worker = (getTask -> doTask -> Worker
Workers = (forall [i:1..N] ([i]:Worke
```

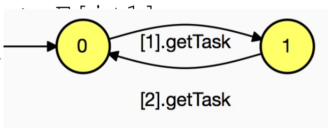


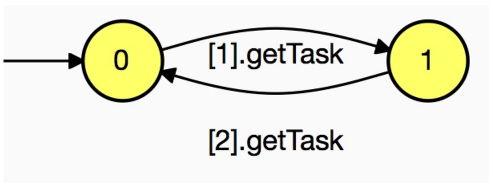


We can add a `Farmer` process to hand out the `Task`s to the `Worker`s in order. Then build a `Farm` composed of the `Farmer` and the `Workers`.

```
Farmer = F[1],
  F[i:1..N] = (when (i < N) [i].getTask
              | when (i>= N) [i].getTask

Farm = (Farmer || Workers).
```





The `Farmer` process is far from ideal in some regards.

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`	`Fmr = F[0],` `F[w:0..W] = when (w< W) [w].wtask -> F[w+1]|` `when (w == W) ([w].wtask ->Fmr).`
Labeling	`lab:P`	see Buff example above
Parallel	`A = (P||Q)`	`Workers = (forall [i:0..N] ([i]:Worker)).`
Relabeling	`P/{new/old}`	`P/{new[i:0..N]/old[i]}`
Hiding	`P\{act}`	`P\{act[i:0..N]}`

For processing automata:

abstraction	`abs(P)`	the removal of τ events
simplification	`simp(P)`	for the simplification of automata
equality	$\_ \sim \_$	compute if two automata are bisimular
fair divergence		remove all τ loops
not fair divergence		replace τ loops with deadlock

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