Modelling and model checking interactive processes

Defining Basic Processes

We are interested in defining processes that have private state and only interact with other processes via **events**. Process names will start with an upper case letter whereas event names will start with a lower case letter.

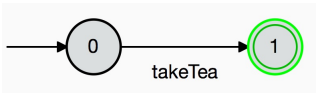
Processes are defined using a simple process algebra $\Sigma = \{Act, ->, |, STOP\}$ the operational semantics of the defined process definition will be rendered as an automata by enclosing the definition in `automata { ... }` as shown in the examples below. We will frequently omit the `automata { ...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

Event prefixing

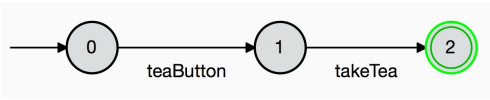
A simple process that performs a single event and stops can be built by prefixing an event `takeTea` to the `STOP` process using the `->` operator by the command:

```
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->STOP).
```

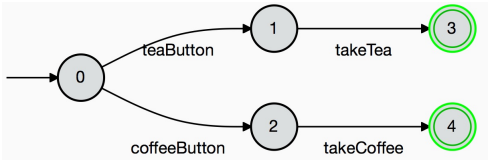


The informal meaning of events will in part be formalised by our definition (to be given later) of parallel composition. Informally we need to think of our events as *hand-shake events*, i.e. event that can be blocked or enabled by the context in which they execute. For example the `teaButton` event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

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

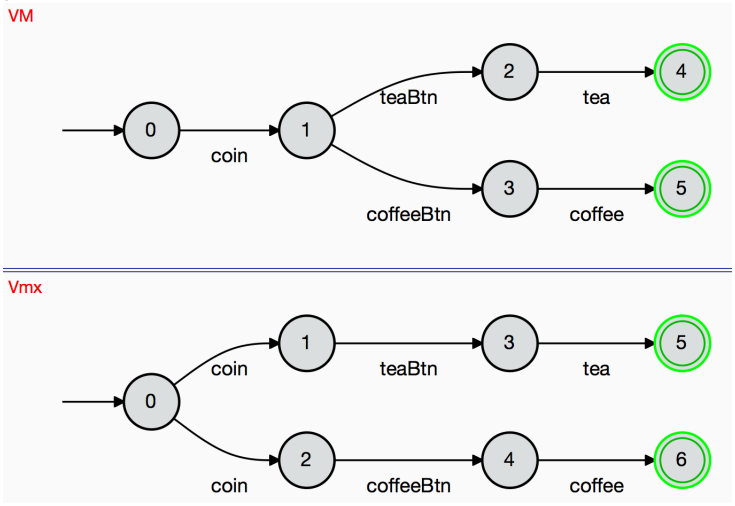


this automata *branches* at the initial node.

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 always 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. It is this notion of indistinguishable by any test that we are interested in here.

Non terminating processes

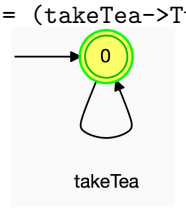
We call the set of know processes the *Process name space*. Initially the *Process name space* is `{STOP}`. Each process definition `P1 = ...` adds the the defined process `P1` to the name space.

Processes consist of a set of states, an initial state and a set of event labelled 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.

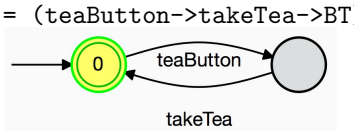
Clearly any state `S` in process `P1` could also be *conceptual identified* with the the process consisting of the same set of states and transitions but with initial state `S`. We use this idea to define non terminating processes. By allowing any valid process to be used where `{STOP}` has been used we can define non terminating or cyclic processes.

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).` The process `Tt` can endlessly perform the `takeTea` event.

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

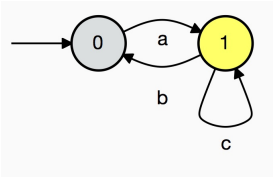



```
BT = (teaButton->takeTea->BT).
```



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).
```

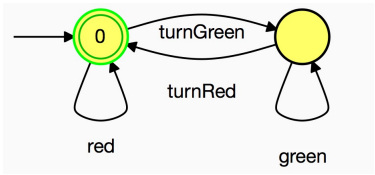


This allows complex processes to be defined without cluttering the *Process name space*.

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 with the behaviour given by the automata. Our tool will automatically generate the automata from the term. The generation of the term from the automata 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.

Event hiding and process simplification

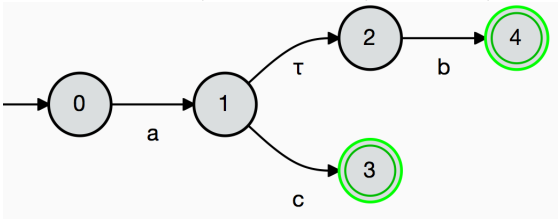
We can make event private by hiding them so they can not be seen. `_\{t}` operator renames the `t` event to a `tau` event and the `abs(_)` operator abstract away the `tau` events.

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

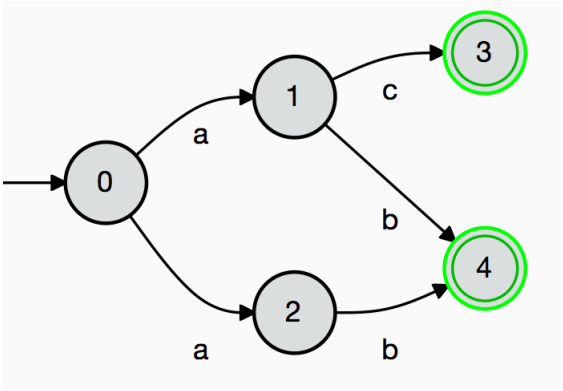
1. there exists `v` such that $x \xrightarrow{a} v$ and $v \xrightarrow{tau} y$ or
2. there exists `v` such that$x \xrightarrow{tau} v$ and $v \xrightarrow{a} y$

See following example:

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




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



Event hiding and non terminating processes

The literature is divided on how to hide τ events that loop. CSP refers to these processes with τ loops as *diverging* and models them as having potentially *chaotic* behaviour. CCS and Discrete Event Systems DES, assumes them to be benign as simply prunes them. Here we offer both options. The CSP option assumes that the system can behave *unfairly* and the CCS option assumes the system behaves *fairly*.

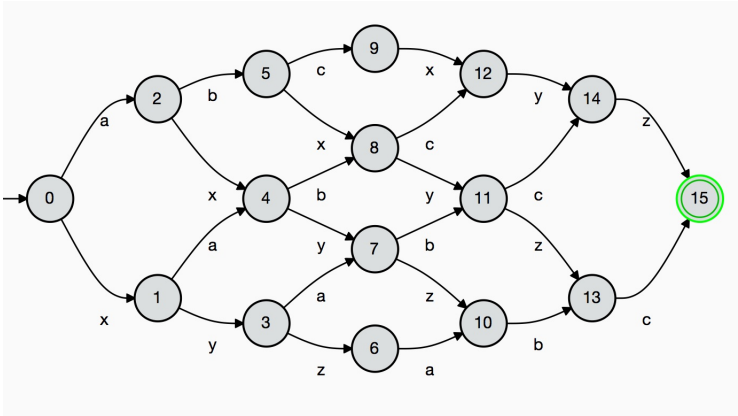
The command `abs(_)` is based on the fair assumption and `abs{unfair}(_)` is based on the unfair assumption.

Defining Concurrent Processes

So far we have have defined sequential processes but now we wish to define how two sequential processes behave when they are both run together. This is modelled using the parallel composition operator $\_ \| \_$. Two processes run in parallel can only interact via event synchronisation and events on only synchronise with other event having the same name.

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.

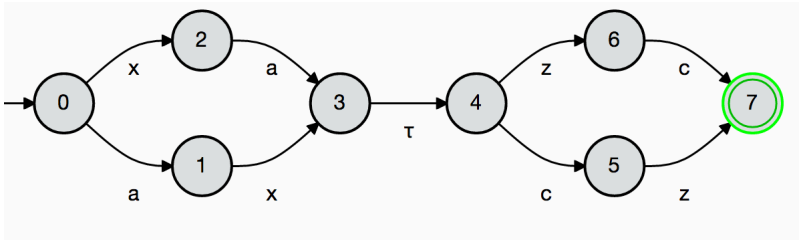
```
P = ((a->b->c->STOP) || (x->y->z->STOP)).
```



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:

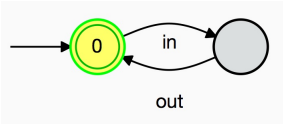
If you can see and event you can synchronize with it and you can block it.

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

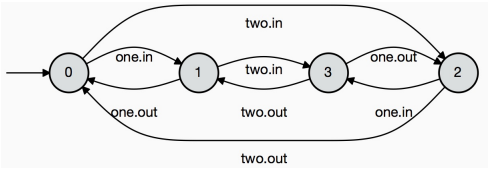
In the following example we make use of a one place buffer `Buf` is a process that when empty can receive some thing `in` and when full can return it `out`.



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).
```



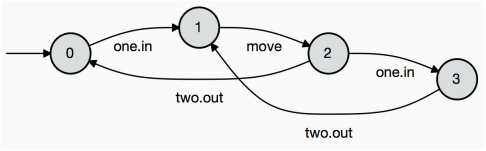
Event renaming

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}).
```

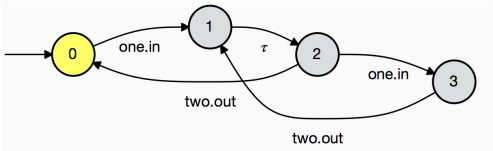


Note that the result is much simpler than the interleaving as the `move` event now can only occur when **both** buffers are able to perform it.

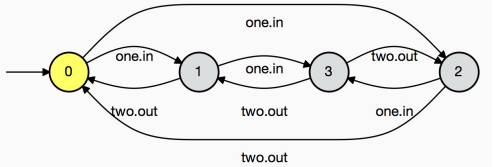
Event hiding and process simplification

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.

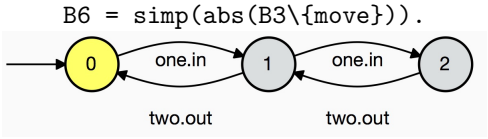
```
B4 = B3\{move}.
```




```
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(_)`.



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 finite set of states. This accurately reflects many situations very well and allows easy and complete push button verification. Alternatively when what you are modelling has infinite state you could use symbolic models but verification frequently requires input from a domain expert and is very time consuming.

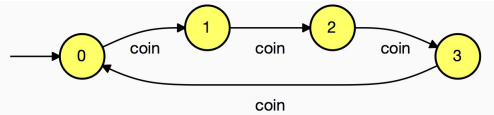
The small world assumption

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 variables with values from the small domain.

Indexing introduces the ability to define a process parametrised 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 parametrised size, the second is to model events that input or output data and finally how to model a parametrised 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[cnt:1..N] = (when(cnt<N) coin->C[cnt+1]
              |when(cnt==N) coin->C[1]).
}
```

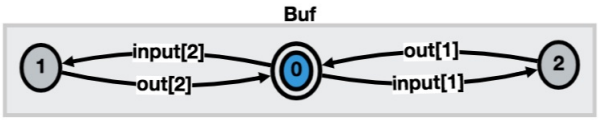
The term `C[cnt:1..N]` on the right of the equality can be thought of as assigning a value to a variable `cnt` and thus as an **input**. The term `C[cnt+1]` as reading the value in the variable and thus as an **output**. Information flows into the process via `C[cnt:1..N]` and out of the process via `C[cnt+1]` and `C[1]`.

On the right hand side of the equality we define guarded events, that is `when(cnt<N) coin->C[cnt+1]` will only add event `coin` that ends at node `C[cnt+1]` when the index is less that the constant `when(cnt<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 that input or output values.

A one place buffer that can accept as input a number from the range `1..N` and then must out put that value is can be represented by an automata with `N+1` states.



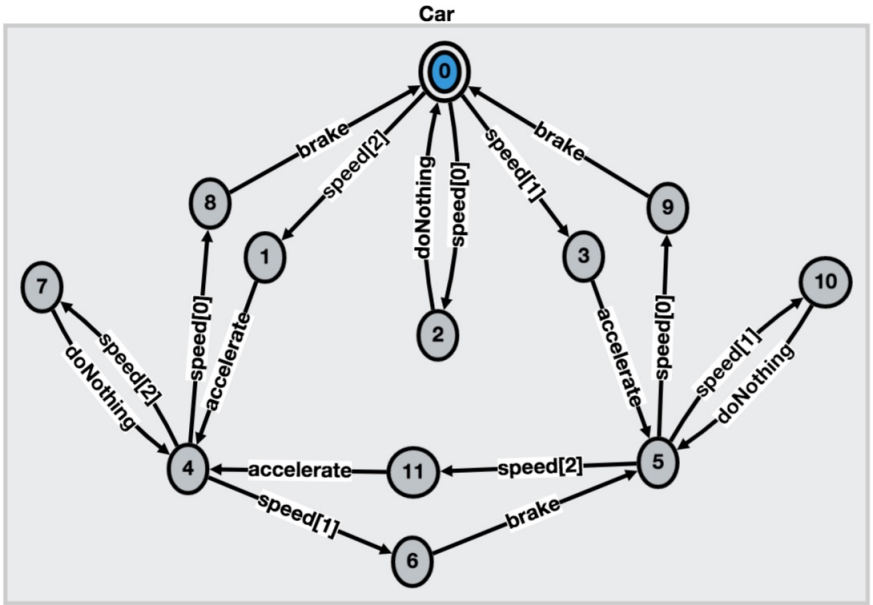
The buffer can be defined by: `Buf = input[v:1..N] -> out[v]->Buf.`

The event `input[v:1..N]` accepts as in put a value that is assigned to the variable `v` and the event `out[v]` outputs the value held in `v`.

Cars Example

Here we mix both state and event indexing. Note in the example the declaration if the index `j` in the event `speed[j:0..N]` becomes before its use in `(when (i< j)`.

```
Car = L[0],
L[i:0..N] =  speed[j:0..N] -> (when (i< j)  accelerate -> L[j]
                             | when (i>j)  brake -> L[j]
                             | when (i ==j) doNothing->L[i]).
```



From Natural Language to indexed process model

Natural languages are expressive but ambiguous. Added to which we are interested in describing event based models and there is no one universal way to describe such systems. This leads to many problems, some of which can be overcome by breaking the task of formalising these informal specifications into some simple steps.

Step 1 find indexes and indexed states
Step 2 find indexed events
Step 3 find all events
Step 4 build automata either sketch and code or code and view.
Step 5 inspect the automata and validate it against specification

We will demonstrate this with a simple example of a lockable door.

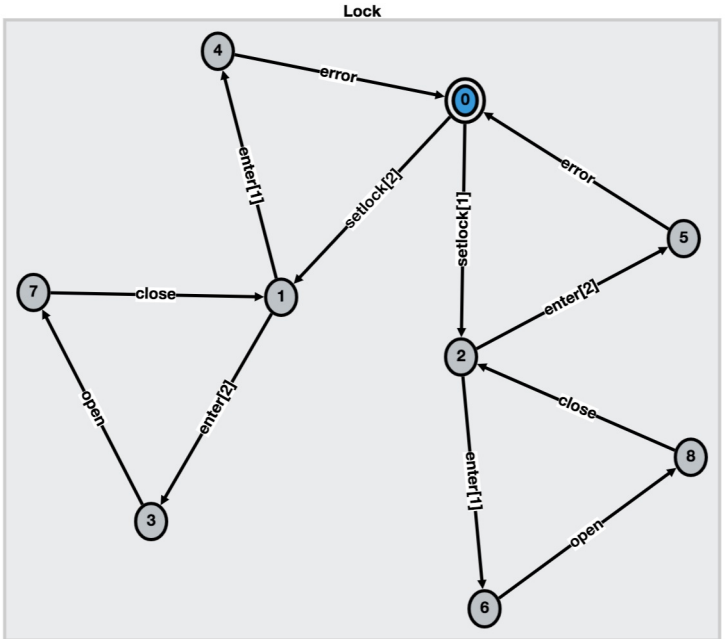
> *Closed doors are always locked. The door starts closed. The lock can hold any of a number of codes. To open you need to input the correct code and after opening the door can only close. Inputting the wrong code is an error and the door returns its start state. Before using the door the code must be set.*
>
> *You may assume that only an administrator can set the code where as any one may use the door by entering codes but such distinctions are not part of the model.*

Step 1 When the numbers of states or events is not fixed but is dependent upon some parameter then you need to build what we call an indexed process. The parameter is an index and in our example this is the `code` the lock uses. As the code needs to be stored by the Lock we need indexed states `L[j:1..N]`.

Step 2 There are two indexed events `setlock[k:1..N]` to set the state of the Lock and `enter[ji:1..N]` to enter a code when trying to open the door.

Step 3 The list of all events: `open,close,error,enter[]` and `setlock[]`

Step 4 Automata when `N==2`



This is defined by:

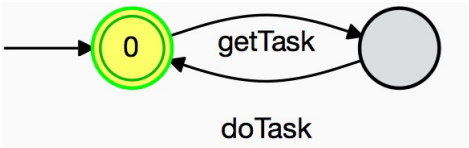
```
Lock = (setlock[k:1..N] -> L[k]),
L[j:1..N] = (enter[i:1..N] ->
                    ( when (i==j) open ->close->L[j]
                    | when(i!=j) error->Lock)).
```

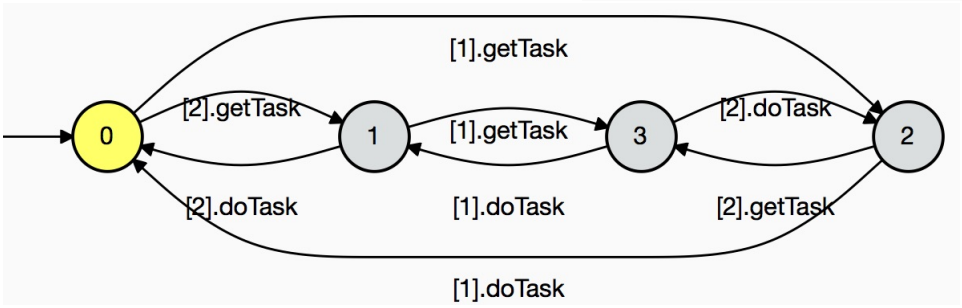
Step 5 Note the value input in the `setlock[i:1..N]` event is stored in the state of the process `L[i]` for subsequent comparison with the value input in the `enter[i:1..N]` event.

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]:Worker)).
```

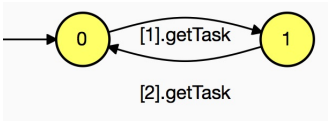


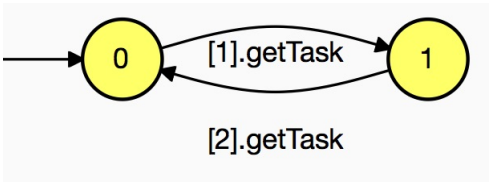


We can add a `Farmer` process to hand out the `Tasks` to the `Workers` in order. Then build a `Farm` composed of the `Farmer` and the `Workers`.

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

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





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

Syntax

Summary of event syntax:

	symbol	meaning
handshake	a	a synchronises with a both must be ready
non blocking send	a!	need not wait - can not be blocked
receive	a?	waits for a! synchronises to become a!

For indexed automata first but a bound on the size of any index `Max` with
`const  Max = 2`

Summary of the syntax used to define processes. process `A` may be defined: `A = <def>` or if you wish to suppress the rendering of the automata then you can use: `A* = <def>` Definitions of automata must be enclosed within:

`automata  \{  ...  \}`

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).`
Labeling	`lab:P`	see below
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

For processing automata use the following operations within:

`operation  \{  ...  \}`

equality	$A \sim B$	A and B are bisimular
inequality	$A! \sim B$	A and B are not bisimular
equality	$A \# B$	A and B are complete trace equivalent
inequality	$A! \# B$	A and B are not complete trace equivalent
fair divergence		remove all τ loops
not fair divergence		replace τ loops with deadlock

Semantics (only for interest)

Giving the appropriate detailed meaning, semantics, to event based processes is far form easy. Consequently there are many hundreds to choose from.

One way to give confidence that you have chosen an "appropriate" semantics is to construct a testing semantics where the tests closely follow how the processes under consideration actually interact.

Processes can be modelled by an automata but some different automata actually model the same process. Consequently it is more accurate to say that a process is modelled by a set of "equivalent"

automata. The notion of process equivalence that is induced by constructing a testing semantics based on how our finite processes interact is Failure Equality

Here our notion of non terminating process equality (for unfair abstraction) is NDFD semantics. This allows for the removal of all τ events including τ loops (divergence). If you view the automata with τ events, `P\{x}` as no more than a step in the computation of `abs(P\{x})` then divergence is replaced by deadlock and all you need consider is the failure semantics of any process.

Where as `simp` only simplifies process up to bisimulation. This is safe to do but means that some simplification steps our tool can not perform.

The use of *fair abstraction* treats τ loops in the same way as in observational bisimulation and fair failures semantics.

Tool Development

This tool started by a 2015 - SWEN302 Agile project and extended during a summer scholarship and currently is being worked in a 2016 - SWEN489 project. It is the basis of a paper comparing Event-B and Model checking.

Extensions TO DO

Below is a list of interesting projects that could be SWEN302/489 or even MSc they are given in particular order. Each project has interest both academically and pragmatically how easy they are to implement depends upon the state of the code base and no extension is of interest unless backed up by extensive executable tests.

1. **Add interrupts.**
2. **Add support for Event B style reasoning.**
3. **Add probabilistic choice.**
4. **Add the ability to model check algebraic rules by generating "all" process models of a fixed size and verifying the rules against these processes.**
5. **Code generate from the models**
   (a) **The Go programming language has Go routines that can communicate via the CSP style event synchronisation that we use here. Consequently might be an easy target language for code generation.**
   (b) **Java code - and apply the specification mining tool to rebuild the automata**
   (c) **Ada code - and use SPARK Ada theorem prover to symbolically verify the indexed models for all values of the index**
6. **Add hierarchical processes (where the state of one process becomes a whole process) the result will be include adding signals that will better model interrupts.**
7. **Add event refinement**
8. **Add δ events - unobservable and blocked + model known examples.**
9. **Make wiring processes together, currently done by renaming events, easier to do by making the GUI interactive.**

Theory Development

We have a theory of handshake and broadcast events and are just completing a theory of probabilistic choice. Adding time or mobility would be interesting but very hard (PhD hard). Smaller steps (MSc) might be to construct a theory:

1. of models that include signals and probability
2. of event refinement that could be applied to models that contain a mixture of handshake event and events from one of broadcast events, signals, probabilistic events.