SWEN224 2015 - LTSA CribSheet

Defining Basic Processes

The simplest process is `STOP` the process that dose nothing. The basic processes that we discuss are finite state processes containing state transitions with atomic events.

Event prefixing

The simplest process that dose something is a single event is built by prefixing an event `takeTea` to the `STOP` process by the command:

```
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. But the events in our formal model have part of the semantic story fixed by how we will define parallel composition. Informally we need to think of our events as `hand shake` events, i.e. can be blocked or enabled by the context in which they execute. For example the `takeTea` event of a vending machine can only occur when some agent is there to also perform a `takeTea` event.

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

Event choice

The choice operator adds the ability to choose between 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).` endlessly performs the `takeTea` event and `BT = (teaButton->takeTea->BT).` endlessly performs `teaButton` followed by `takeTea`.

Nonterminating processes

Processes consist of a set of states, an initial state and a set of event labeled state transitions. Given the sets of states and transitions the process can be identified with the initial state. Further any state `S` can be identified with the the process consisting of the same sets of states and transitions but with initial state `S`.

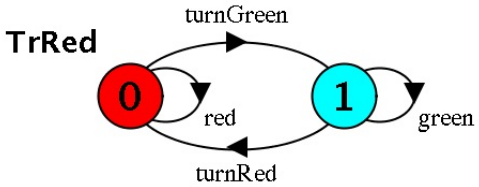
We can use this conceptual identification to define nonterminating processes simply by adding the process name to `{STOP}` the set of valid processes. Hence `P=a->P.` is the process that endlessly performs `a` events. We allow local process or states to be defined within the definition of a process by separating definitions with a comma.

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

Translating any FSA into an LTS term

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

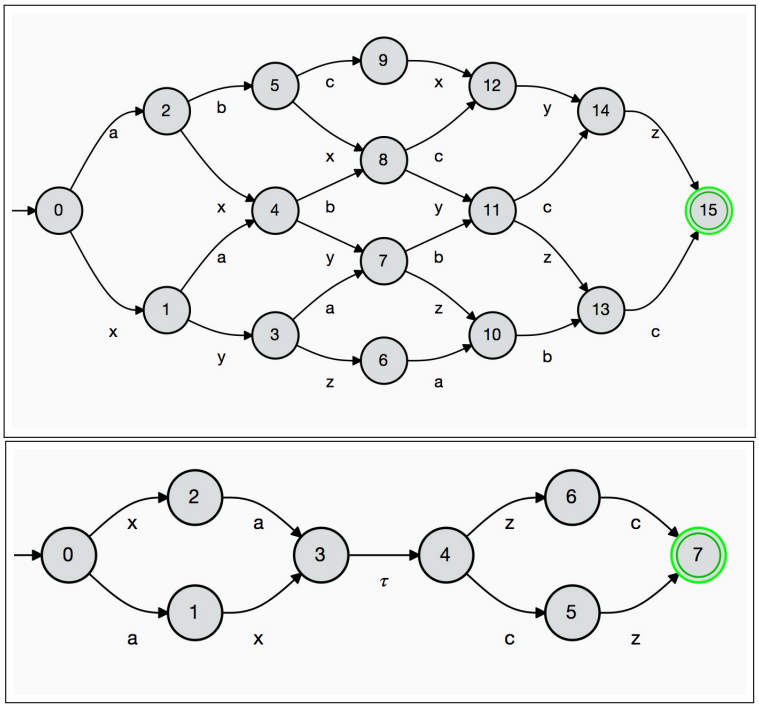
```
TrRed = (red->TrRed | turnGreen ->TrGreen),
  TrGreen = (green->TrGreen|turnRed->TrRed).
```

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

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 process interact and because of event synchronisation we have:

If you can see and 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 the 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** dose the **b** event actually get performed.

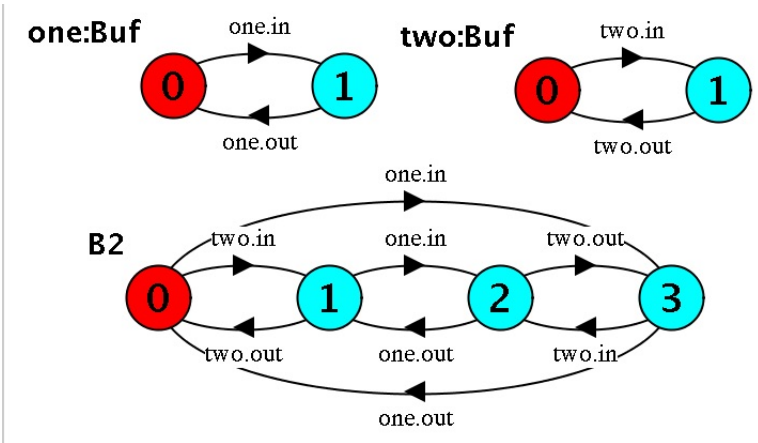
Labeling Processes

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 labeling processes `one:Buf` the tool labels all events in the process `one.in` and `one.out`.

Using process labeling 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 `||`

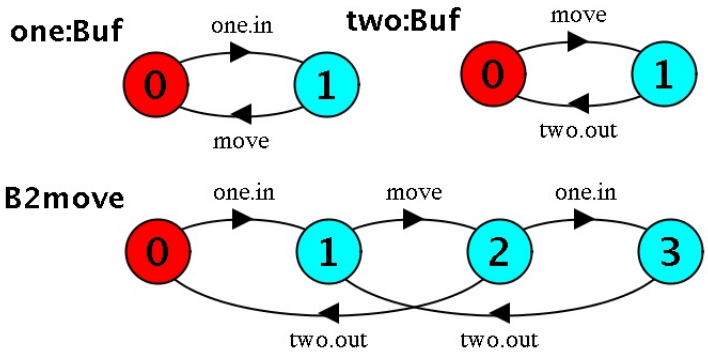
```
||B2=(one:Buf||two:Buf).
```



Event renaming

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

```
||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.

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

Event hiding

We can go further and hide the `move` event

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

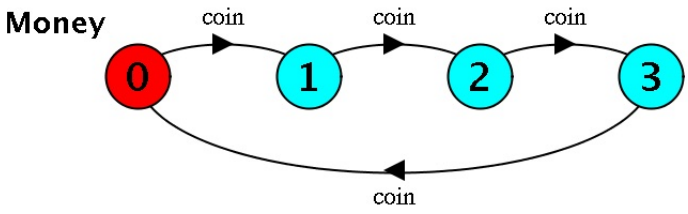
The `move` event becomes a `tau` event that can not by synchronized with nor can be blocked. The `tau` events can be removed by **minimizing**. Hiding is commonly used to make communication private.

Indexed Process definitions

Basic process definitions have a fixed bounded set of states. 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]`

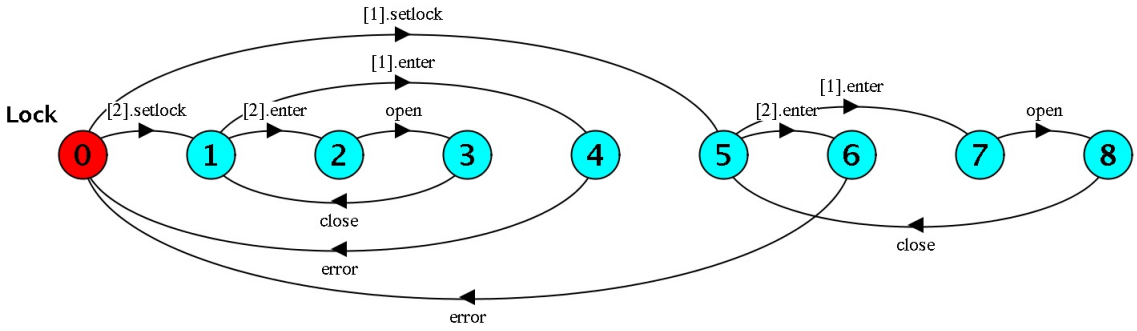
```
Money = C[1],
C[i:1..N] = (when(i<N) coin->C[i+1]
            |when(i==N) coin->C[1]).
```

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]`

```
||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`)

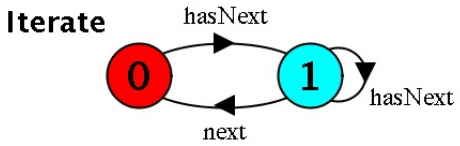
The LTSA tool can check both types of requirements and will return a trace to the state in which the error occurs. Doing nothing satisfies any safety requirements but fails liveness requirements. So if you are not sure if a requirement is a safety or a liveness requirement:

ask your self 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.

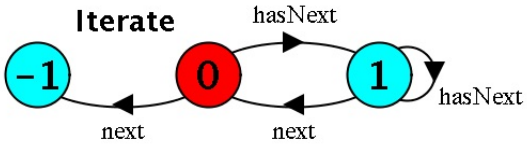
Safety is a process, over a restricted alphabet, that defines the desired safe behavior. Any other behavior, over the restricted alphabet, will lead to an ERROR.

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




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

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 -> Iterate).
```

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 act like an observer. That is they do not block any event of the process under test until it has violated the safety property and then it reports the error.

Progress command defies a set of events, one of which must always be reachable, no mater 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).`
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]}`

abstraction	`abs(P)`
simplification	`simp(P)`

For requirements checking we have:

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