Modelling and model checking interactive programs

Here we are interested in programs that interact with the world around them and need not terminate to be useful. These programs run concurrently with the world around them. The formal modelling of such programs has been achieved in a wide variety of styles. Unfortunately each style of modelling take a significant effort to understand in detail and consequently most people become an *expert* in only one way of modelling. Each of the modelling styles is predicated on a set of assumptions that are commonly unvoiced and hence comparison between styles is far from easy.

To help formalise this we model programs that can themselves be concurrent. The class of such concurrent programs is very varied and many distinct styles of analysis appear in the literature. We find it helpful to decompose these concurrent programs into two broad, overlapping, classes *State based* and *Event based*.

Here we formally model interactive processes by first formally capturing their operational behaviour and then defining process equivalence on the operational semantics. Equality has the following properties:

1. it is a reflexive, symmetric and transitive relation,
2. equal processes can be substituted and
3. equal processes can not be distinguished.

We will use automata, Labeled Transition Systems LTS, to model the operational behaviour of processes and define operator on automata to compose processes and there by building larger automata. From our perspective the most important operator is parallel composition as we use this define how processes are "distinguished" and hence we use parallel composition to define process equality. To do this we, later, will formalise Testing in a way that can be applied in many situations.

By concurrent state based programs interact by sharing memory and concurrent event based programs interact by sharing events. Our event based programs are commonly referred to as *processes*. On the one hand the same real world object can be formalised in a state based or event based fashion and on the other hand different languages implement shared memory of shared events as primitive.

We believe that state based models can be implemented on event based primitives and event based models can be implemented on state based primitives. The usefulness of having different styles to formalise the same thing is that different objects are easier to formalise with each of the styles.

Although it is interesting to have a sound and complete inference system to reason about programs it should be borne in mind that some systems used practice are not complete. Both B and Event-B define refineement as forward simulation and do not use backward simulation hence are inherently incomplete with respect to refinement and yet have been shown to be be of great use in practice. What we believe to be of greater practical importance than completeness is that the inference system is

1. sound

2. easy to understand

3. push button feedback

4. extensible

1 A practical approach to formalisation

From our perspective we view the primary role of Formal Methods as the modelling of systems of interest. Having built the formal model we can then prove, or discover, properties the model has. Such properties are only of practical interest in as much as the model is a accurate reflection of the actual systems of interest. To give us confidence that the models we use are an accurate reflection of the behaviour of the systems we are interest in we construct testing semantics. Consequently you should have no confidence applying any results if you have no confidence in that the system of interest actually interacts with the world around it in the manor prescribed by the testing semantics.

Any model, formal or otherwise, is an abstraction of the real system. Ideally models will be small and simple while still faithfully capturing the aspects of the system we are interested in.

Naturally if you change what you are interested in you may have to change the models you are using. Here this can be achieved by changing the testing semantics. The moment you have two models of the same system the natural question is how are they related. The multiple models approach is quite standard in computer science. One, very abstract model, might be called a specification and a second, more concrete but still abstract, model an implementation. The relation between specifications and implementations is often formalised by refinement. Frequently both the specification and implementation use the same style of modelling, they differ only in their level of abstraction or the detail they contain. Here we are interested in both refinement with in a single formalism and refinement between different, but related, formalisms.

1.1 Initial event based modelling

Our style of modelling can be described as based on *hand shake events* and as is common we call such programs processes. We consider CSP, CCS and ACP as all using hand shake events and like here make the following assumptions:

1. we abstract away details about time
2. an observable event in one process can require the existence of another process to be ready to perform a *related event* before it is executed. This results in a branching structure and would require the use of branching temporal logic if one decided to use logic to specify the details of a processes behaviour.
3. an event is either *observable and blockable* or *unobservable and unblockable* (Note the DES community separate out these properties and hence have four types of event)
4. we abstract away details about one event *causing* a related event to occur. Consequently all events in a set of related events are treated equally (Note modelling causality between synchronising events has been shown to be useful when relating handshake events with broadcast events)
5. a process can be in a state where it is prepared to perform any event from some set of events. How this is achieved is not specified but is a programming language primitive in Ada rendezvous, Occam events and Go events.
6. we adopt the interleaving assumption - parallel processes can be thought equivalent to sequential process

Here we are less concerned with representing systems exclusively as terms and constructing a sound and complete set of axioms than in their operational semantics. Based on the operational semantics we define sound algorithms for the rewriting of the semantics. This can be seen as what in the literature is called *Visual verification*.

We wish to allow for the change of our models of a system by either changing the modelling language used or changing the operational semantics given to the terms. Not only this but we wish to make such changes formally. This we can do because we have an abstract definition of testing and testing refinement hence using this as a common language to define different styles of modelling we are able to define a formal relation between the styles of modelling.

Interpreting specifications and implementations as predicate we have the common notion of refinement as logical implication. From Logic we know that distinct logics can be related by Galois connections, commonly know as theory morphisms. Galois connections form a strong connection between theories that have well known and useful properties. Although Galois connections ensuring a close relation between two theories we have found them sufficiently flexible to formalise interesting refinement steps.

1.2 Secondary models

Our primary formalism is of *symmetric hand shake events* and adopts the *interleaving assumption*.

We will construct two models with out the interleaving assumption and show that they are equivalent. We do this by recording which sequential process owns and event and then building *automata of owned event* and *Petri Nets of owned events*.

We also construct *broadcast events*. In attempting to show the equivalence between hand shake processes i.e.processes with hand shake events and broadcast processes we were forced to extend what hand shake processes modelled. The hand shake processes were extended to distinguish *causality* between synchronising events.

First we extended hand shake event to being owned events and then extended them to being causal hand shake events. Quite naturally with could apply both extensions to have *causal owned events*. As all four models can be related by Galois connections we are able to change what appears in our model as we move from specification to implementation.

1.3 Other Event based Approaches

Process calculi CSP, CCS, ACP and Lotos are successful examples of event based formalisms. They have generated a wide range of denotational semantics and process equivalences. Each can be given an axiomatic semantics to enable reasoning about the processes. Although these approaches make central use of axioms some, notably CCS and ACP also construct an operational semantics on which both operations and equivalences can be defined.

The **B and Event-B approach** is to start with an abstract specification of the System as a whole. This can be stepwise refined until sufficient detail has been provided and then the detailed System can be decomposed into the *Process* of interest plus the *Context* it has been designed to work in.

Two practical aspects of this approach are:

1. the abstract and more detailed concrete models are built by hand, one is not constructed form the other (not correct by construction), then the relation between the two models is verified. Thus the method can be applied in both directions, analysis by refinement and by abstraction.

2. The models defined at each level can be deterministic. The retrieve relation between the two models introduces non determinism that is removed by verifying the relation is a refinement.

This approach could be described as supporting hierarchical specifications. That is each level in the hierarchy consists of one or more specification given at the same level of detail whereas between levels the specifications are of the same thing but given at differing levels of detail. Frequently an abstract specification can be constructed from a more detailed specification by the abstraction of some detail. In our simple models that contain only atomic events this means the abstraction of some of the events.

A **Control theory approach** used quite widely, is to model the context in which the process finds its self in addition to the process itself. In this approach we have separate models for the *Process* itself and the *Context* in which it resides. The combination of the two is called the *System*.

Frequently for processes we wish to build it is easier to decompose the specifications into two steps:

1. specify the behaviour of the context in which the program will run

2. specify the behaviour of the system as a whole, that is specify the behaviour of the program and the context running together

Because we have parallel composition as a first class citezen we can further refine the Process if we need to.

1.4 Our approach

We are interested in defining processes that have private state and only interact with other processes via **events**. Unlike the process calculi CSP, CCS, ACP and Lotos we do not make use of the axiomatic semantics to reason about the processes. Our non axiomatic approach that loosely follows the notion of *Visual Verification* from Valmaries .. [], later we will make use of the axioms check (*test*) the validity of our approach. Testing is not verification and in an ideal setting once a theory has been thoroughly tested it should be verified by, for example, encoding the theory a theorem prover.

Most industrial software, even most life critical software, is tested not verified the idea of testing a theory might prove acceptable, if not to theorists, at least to industrialists. Our approach of model checking a theory could be viewed as taking an experimental approach to theory development.

2 Defining Basic Processes

Details of, `Proc`, the language we use to specify will be introduced as needed. Philosophically we follow ACP and, where for a simple fragment of the language, we give the operations of out language meaning as automata to automata mappings.

```
simpleProc = Act->STOP | _[]_ | _=>_
```

Where `Act->STOP` is a set of event names and is interpreted each interpreted as a simple automata with two nodes, a start node and a distinct end node plus a single edge labelled with the event name. The operations `_[]_` represents choice between two process and `_=>_` the sequential composition between two processes. Many examples will follow but it will help t note that these operations are defined on the automata by *glueing* specific nodes together.

Event based formalisms define processes that interact with processes running concurrently with them. Only events can be seen, interacted with, by other processes as such these observable events can be thought of as *half events* that is they need the other half to be ready before they can be executed.

Our approach requires the distinction between what is in the interface of a process and what in private to the process. Subsequently we define how to remove (*abstract* away) what is private while preserving what can be seen of the process behaviour.

In the tool we introduce process names will start with an upper case letter whereas event names will start with a lower case letter.

Sequential 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.

An automata *is a tuple* $\mathsf{A} \triangleq (\mathsf{N_A}, \mathsf{S_A}, \mathsf{T_A}, \alpha_\mathsf{A})$ *where:*

$\mathsf{N_A}$ *is the set of automata nodes,*

$\mathsf{S_A}$ *is the set of start nodes* $\mathsf{S_A} \subseteq \mathsf{N_A}$

$\mathsf{T_A}$ *is the set of transitions, triples* (n, a, m) *where* $\mathsf{a} \in \alpha_\mathsf{A}$ *and* $\{\mathsf{n}, \mathsf{m}\} \subseteq \mathsf{N_A}$

α_A *is the alphabet of the process, the set of events in the processes* $\{a : n \xrightarrow{a}_A m\} \subseteq \alpha_A$

□

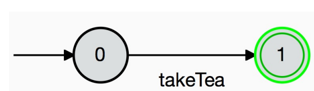
Processes definitions are enclosed in `processes { ... }` as shown in the examples below. We will frequently omit the `processes { ... }`. The processes that are to be displayed as automata appear as a list `automata A,Ap.`.

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.

2.1 Event prefixing

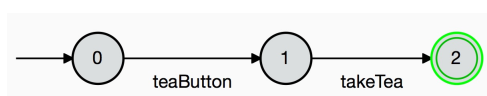
A simple process that performs a single event and stops can be built by prefixing an event `takeTea` to the `STOP` process using the `->` operator by the command:

```
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).
Two = (teaButton->Simple).
```



The two definitions of `Two` produce the same automata. If you only want to see `Two` then you can suppress the production of the initial automata `Simple` by add `*` as a suffix to the name:

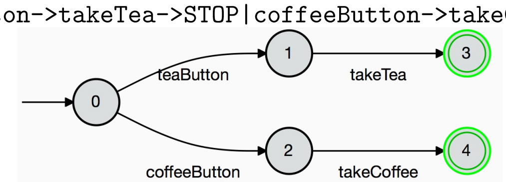
```
automata {
  Simple* = (takeTea->STOP).
  Two = (teaButton->Simple).
}
```

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.

2.2 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).
```



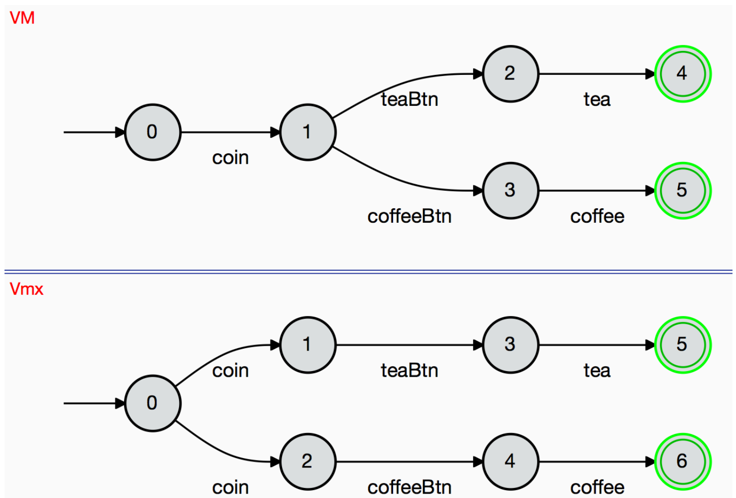
this automata *branches* at the initial node.

It will prove useful to define this operation on automata. As such the automata for the choice between two component processes can be built from the two component automata simply by glueing together the start nodes of each process. With out changing the behaviour of the process in any way we can also glue the two end nodes together.

2.2.1 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.

2.2.2 Properties of Choice

Choice is symmetric in that `X|Y` is equal to `Y|X`. In what follows we write process equality as $\sim$ and hence:

$$\mathsf{X|Y} \sim \mathsf{Y|X} \qquad \mathsf{X|(Y|Z)} \sim \mathsf{(X|Y)|Z}$$

details about, $\sim$, process equality follow much later.

The Choice between two identical processes is no real choice and hence:

$$\mathsf{X|X} \sim \mathsf{X} \qquad \mathsf{X|STOP} \sim \mathsf{X}$$

Meaning has been given to Processes in the form of an Axiomatic Algebraic semantics see [BW90]. Here we give meaning to processes by defining their operational behaviour as automata. From the details of what constitutes an automata we can see that the above equalities hold for our semantics. Looking at the same thing the other way in [BW90] they first define the algebraic properties of processes and then show the automata satisfy them.

2.3 Internal Choice +

Used to represent a process that internally, with no external involvement, can choose to behave like either process. Consequently such a process can not be guaranteed to behave like either of the processes.

The relation between the two definitions of choice is captured by the axioms:

```
(a->A+a->B) =_F (a->A|a->B) =_F (a->(A+B))
```

NOT bisim need to use testing or Failure equality!

Defined in CSP and ATP but not in ACP nor CCS. Whereas neither CSP nor ATP make central use of automata to visualise processes but both ACP and CCS do.

Internal choice can be modelled by allowing automata to have a set of start states. Thus `(a->STOP)+(b->STOP)` is a automata with two posible start states.

Using a set of start states facilitates the modelling of processes with either probabilistic choice or *active* actions.

2.3.1 Properties of Internal Choice

Choice is symmetric in that `X+Y` is equal to `Y|X`. In what follows we write process equality as $\sim$ and hence:

$$\mathsf{X} + \mathsf{Y} \sim \mathsf{Y} + \mathsf{X} \qquad \mathsf{X} + (\mathsf{Y} + \mathsf{Z}) \sim (\mathsf{X} + \mathsf{Y}) + \mathsf{Z}$$

2.4 Sequential composition

Two processes can be composed in sequence using `=>`. Just like with the choice operator we can define sequential composition by gluing nodes together. Consequently the automata `A=>B` can be built by gluing the end node of the automata of process `A` with the start node of automata `B`

Clearly action prefixing and sequential composition are closely related:

```
a->b->STOP = (a->STOP)=>(b->STOP) = (a->STOP)=>(b->STOP)=>STOP
```

in what follows we often make no distinction between such process specifications. Sequential composition should also satisfy the following axioms:

```
X=>(Y=>Z) ~ (X=>Y)=>Z
(X|Y)=>Z ~ (X=>Z)| (Y=>Z),  (X+Y)=>Z ~ (X=>Z)+ (Y=>Z)
W=>(X+Y) ~ (W=>X)+(W=>Y) ~ (W=>X)|(W=>Y)
```

2.5 Representation and mixing internal choice with sequential composition

The process `(s->t->STOP)=>(a->STOP)+(b->STOP)` is interesting in that the internal choice is made after the `t` event. Where as process in the `(s->t->a->STOP)+(s->t->b->STOP)` it is made before the `s` event. Despite this neither can be distinguished by any test. They are failure semantics equivalent. This is a direct consequence of the fact that you can not observe when a choice is made you can only *infer* a choice must have been made some time in the past. Because such process cannot be distinguished how you choose to represent such a specification is a matter of taste.

2.6 Non terminating processes

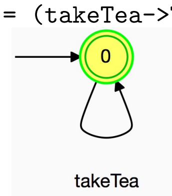
We call the set of know processes the *Process name space*. Initially the *Process name space* is `{STOP}`. Each process definition `P1 = ...` adds the the defined process `P1` to the name space.

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.

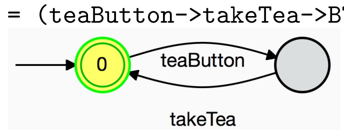
Clearly any state `S` in process `P1` could also be *conceptual identified* with the the process consisting of the same set of states and transitions but with initial state `S`. We use this idea to define non terminating processes. By allowing any valid process to be used where `{STOP}` has been used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

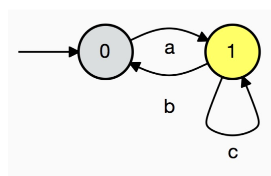



```
BT = (teaButton->takeTea->BT).
```



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).
```

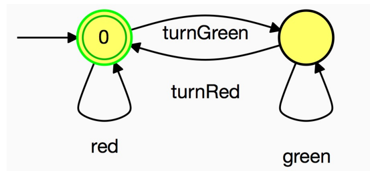


This allows complex processes to be defined without cluttering the *Process name space*.

2.7 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.

3 Semantics of processes with atomic events

Automata provide a formal definition to the operational (semantics) behaviour of our processes. Nonetheless distinct automata can be used to represent indistinguishable processes. We formalise this by defining semantic equivalences of automata. The question as to what automata should be equated and how do you justify you notion of equality we leave to later. Here we are going to introduce four different notions of process equality that are widely used and widely discussed in the literature.

3.1 Complete Trace equality

The trace semantics of a process are the set of executions the process can undertake.

Complete finite traces must end in state from which not event can occur:

$$Tr_{Fin}(\mathsf{P}) \triangleq \{tr : \exists n : S_{\mathsf{P}} \xrightarrow{tr} n \wedge \pi(n) = \emptyset\}$$

Infinite traces do not end:

$$Tr_{Inf}(\mathsf{P}) \triangleq \{tr : S_{\mathsf{P}} \xrightarrow{tr}\}$$

The complete traces of a process

$$Tr_c \triangleq Tr_{Fin} \cup Tr_{inf}$$

Complete trace equality:

$$\mathsf{P} =_{Tr_c} \mathsf{Q} \triangleq Tr_c(\mathsf{P}) = Tr_c(\mathsf{Q})$$

Trace equality dose not distinguish deterministic processes from non deterministic processes and hence the two processes in Section 2.2.1 are identified.

3.2 Bisimulation

A bisimulation $\sim$ is relation on the nodes of an automata that is symmetric $n \sim m \Rightarrow m \sim n$ and

$$n \sim m \wedge n \xrightarrow{a} n' \Rightarrow \exists m'.m \xrightarrow{a} m' \wedge n' \sim m'$$

Two processes P and Q are bisimular if and only if there is a bisimultaion relation that relate their start nodes, $\mathsf{P_S} \sim \mathsf{Q_S}$.

Bisimulation equivalence is much finer than complete trace equality and only equate processes that could not possibly be distinguished. Consequently the two processes in Section 2.2.1 are not bisimular (bisimulation equivalent).

To help us understand how bisimulation equivalence works we give a simple co-inductive algorithm to compute the maximal bisimulation relation using a node colouring where nodes with the same colour are related.

1. Initially colour all nodes with the same colour.

2. Repeatedly recolour the nodes using

$$Col_{i+1}(n) \triangleq \{(a, Col_i(m)).n \xrightarrow{a} m\}$$

3. stop when the recolouring changes nothing.

$$Col_{i+1}(n) = Col_{i+1}(m) \Leftrightarrow Col_i(n) = Col_i(m)$$
$$\text{Colour map } \mathsf{Col(n)} \mapsto \{(\mathsf{a}, \mathsf{Col(m)}).\mathsf{n} \xrightarrow{\mathsf{a}} \mathsf{m}\}$$

The maximal bisimulation relation that can be computed very quickly and bisimular nodes, nodes with the same colour, can be identified to produce a simpler automata. Thus bsimulation offers a computationally easy way to **simplify** processes.

This algorithm can be applied to many automata at the same time and can be used to compute an equivalence class on a set of automata. To cope with the multiple start states we first compute the bisimulation colouring and Colour Map. Then two processes, A and B are bisimular if and only if

$$\mathsf{A} \sim \mathsf{B} \triangleq \bigcup_{\mathsf{s} \in \mathsf{A_S}} \mathsf{Col(s)} = \bigcup_{\mathsf{s} \in \mathsf{B_S}} \mathsf{Col(s)}$$

Bisimulation has attractive mathematical properties, is easy to compute and has proven to be of practical use. Consequently bisimulation relations have been defined on many different structures. As we shall see the bisimulation colouring algorithm is often used as a component when computing other process equalities.

Both these equivalences are congruent with respect to our process operators (substitution of equivalent processes).

$$A \sim B \Rightarrow A||P \sim A||P$$

$$A =_{Tr_c} B \Rightarrow A||P =_{Tr_c} A||P$$

3.3 Failure Semantics

This denotational semantics is taken from [**?**] where it has been applied to processes with hand shake events. It has been shown to correspond to a natural notion of Testing semantics in [].

$$\mathsf{Fail}(\mathsf{A}) \triangleq \left\{(\mathsf{t},\mathsf{R}).\mathsf{s}\xrightarrow{\mathsf{t}}_{\mathsf{A}}\mathsf{n} \wedge \mathsf{R} \subseteq (\mathsf{Act} - \pi(\mathsf{n}))\right\}$$

$$\mathsf{A} =_{\mathsf{F}} \mathsf{B} \triangleq \mathsf{Fail}(\mathsf{A}) = \mathsf{Fail}(\mathsf{B})$$

3.4 Quiescent Trace semantics

This denotational semantics is taken from [**?**] where it has been applied to processes with broadcast events. It has been shown to correspond to a natural notion of Testing semantics in ?.

$$\mathsf{Qtr}(\mathsf{A}) \triangleq \left\{\mathsf{t}.\mathsf{s}\xrightarrow{\mathsf{t}}_{\mathsf{A}}\mathsf{n} \wedge \pi(\mathsf{n}) \cap \mathsf{Act}_{!} = \emptyset\right\}$$

$$\mathsf{A} =_{\mathsf{Qtr}} \mathsf{B} \triangleq \mathsf{Qtr}(\mathsf{A}) = \mathsf{Qtr}(\mathsf{B})$$

Note $\mathsf{A} =_{\mathsf{F}} \mathsf{B} \Rightarrow \mathsf{A} =_{\mathsf{Qtr}} \mathsf{B}$

4 Event hiding and process simplification

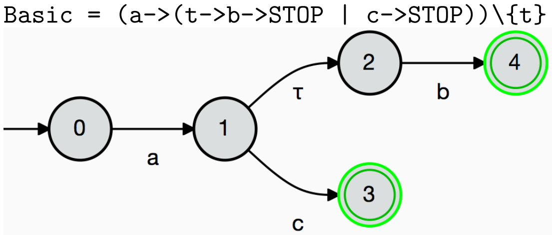
Our process language can be used both to specify *implementations* of processes and more abstract *specifications*. Our tool is designed to support the approach popularised by Event B. Both the abstract specification and the more concrete implementation are specified in the same language. The tool then checks that the implementation is a *refinement* of the specification.

In our event based approach events in the implementation but not the specification are hidden by abstraction and the resultant process is checked for equality with the specification. In this section we define event hiding and abstraction.

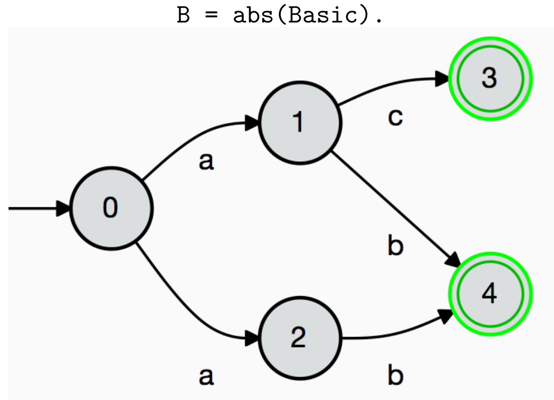
We can make events private by hiding them so they can not be seen. `_\{t}` operator renames the `t` event to the unobservable `tau` event.

See following example:

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



`B = abs(Basic).`



In the above example `abs` introduces two observable events:

1. $0 \xrightarrow{\mathsf{a}} 2$ in place of the event sequence $0 \xrightarrow{\mathsf{a}} 1$ and $1 \xrightarrow{\tau} 2$

2. $1 \xrightarrow{\mathsf{b}} 4$ in place of the event sequence $1 \xrightarrow{\tau} 2$ and $2 \xrightarrow{\mathsf{b}} 4$

The `abs(_)` operator abstract away the `tau` events. To cope with cases where many τ events can be executed one after the other we first define $x \overset{\tau}{\Longrightarrow} y$.

A sequence of zero or more τ events

$$x \overset{\tau}{\Longrightarrow} y \triangleq \exists i \geq 0 : \exists n_1, n_2, ..n_i : x \xrightarrow{\tau} n_1, n_1 \xrightarrow{\tau} n_2 \ldots n_i \xrightarrow{\tau} y$$

can be executed unseen and are represented as $x \overset{\tau}{\Longrightarrow} y$. When $i = 0$ we have $x \overset{\tau}{\Longrightarrow} x$ for any x.

Abstraction constructs, $x \overset{\mathsf{a}}{\Longrightarrow} y$ the observable semantics:

$$x \overset{\mathsf{a}}{\Longrightarrow} y \triangleq \exists u, v : x \overset{\tau}{\Longrightarrow} u \wedge u \xrightarrow{\mathsf{a}} v \wedge v \overset{\tau}{\Longrightarrow} y$$

4.1 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*.

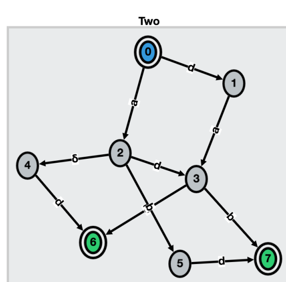
The command `abs(_)` is based on the fair assumption and `abs{unfair}(_)` is based on the unfair assumption.

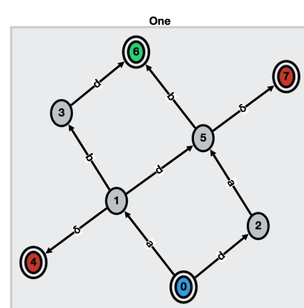
With the unfair assumption congruence with respect to interleaving parallel composition requires some care because the relation between events from parallel components is *fair*. That is to say the infinite execution of an event from on process can not prevent the parallel process executing an unrelated event.

In other words one process, P, diverging will not effect the events of a second process, Q, running in parallel whereas with the unfair assumption it can block other events on P. But with interleaving composition P $\|$ Q the events of P and Q can no longer be distinguished.

To accommodate this label nodes on a τ loop as divergent and require that parallel composition preserves divergence. This has the effect of allowing divergence of P to block the events of Q.

```
automata {
S* = a->X,
   X = (t->X|b->STOP).
Simple = S\{t}.
O* = Simple||(d->STOP).
One = abs{unfair}(O).
T* = abs{unfair}(Simple).
Two = T||(d->STOP).
}
```





Pragmatically divergence and deadlock are rarely wanted and their existence merely indicate that the definitions are erroneous. In such situations it is unimportant how we model divergence as it will be removed. In situations where we do want to model process with deadlock or divergent behaviour then we need to be more careful.

Frequently we are interested in how a system behaves when errors occur but when errors occur is rarely determined exactly. Hence modelling a systems correct and error behaviour will introduce some probabilistic or non deterministic behaviour. Hiding both errors and their handling may introduce divergence and this may be best interpreted *fairly*.

5 Concurrent Processes

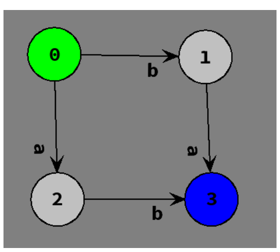
So far we have have defined sequential processes but now we wish to define how two sequential processes behave when they are both run together. This is modelled using the parallel composition operator $\_ \parallel \_$.

Two processes run in parallel can only interact via event synchronisation How this is defined depends upon the interpretation you wish to give your events. In what follows we consider two distinct styles of events, hand shake events as found in CSP and CCS as well as broadcast events as found in IOA.

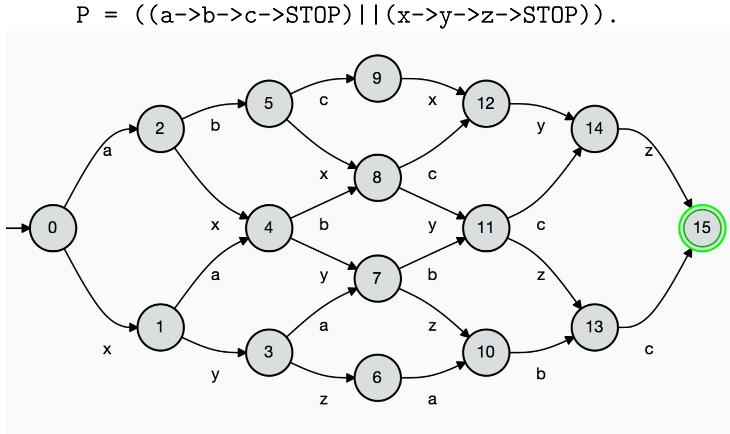
Two processes run in parallel can only interact via event synchronisation and events on only synchronise with other event having the same name.

Below left the parallel composition of two simple processes `(a->STOP||b->STOP)` is shown to be semantically equivalent to a single sequential process `(a->b->STOP|b->a->STOP)`. The ability to replace parallel processes with sequential process greatly simplifies their analysis.

`(a->STOP||b->STOP) ~`
`(a->b->STOP|b->a->STOP)`



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



Above right 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.

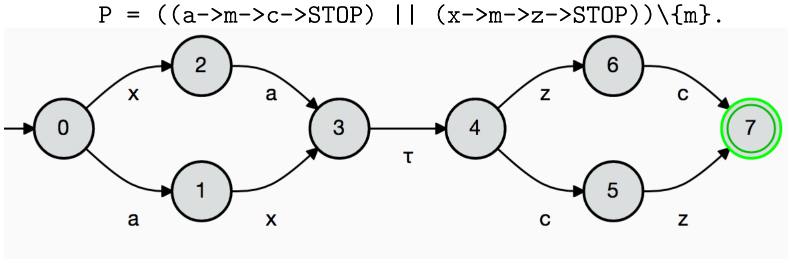
This definition of parallel composition equates parallel processes with a single sequential process. As such some aspects of real parallel processes are lost. Some times this equality is referred to as being based upon the *interleaving assumption*.

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.

5.1 Handshake synchronisation

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.

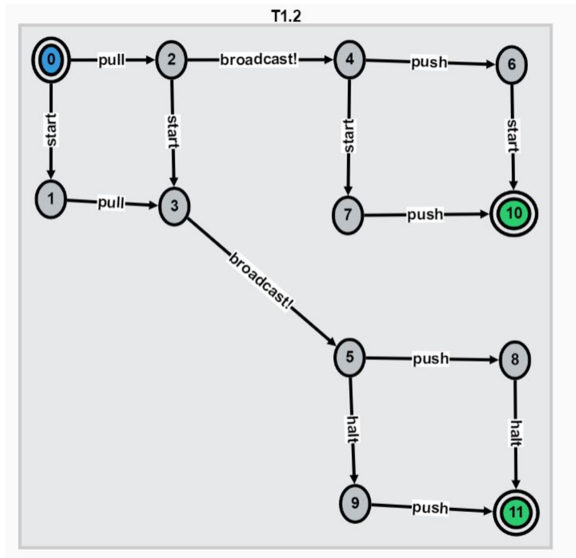
Hence the only way the control the order of two events from different concurrent processes is to introduce a synchronizing event. In above the `a` event and the `z` event are from different concurrent processes in the interleaving example either could occur first. Whereas in the synchronization of the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

5.2 Broadcast event synchronisation

In the previous section when events from two processes synchronised the the synchronising events from both processes were treated the same, both could block the other process. This style of synchronisation captures real events such as pushing a button I cannot push a button that is not there or is frozen nor can button on a vending machine be pushed if I am not prepared to push it. Both me and the vending machine must wait for the other to be ready before the buttonPush event can occur.

```
(a -> broadcast! -> c -> STOP)||(x -> broadcast? -> y -> STOP)
```



Other events are not like a broadcast! events cannot be blocked by a process not being ready for them. Real examples include: I can send an email even if you are not ready to read it. A traffic light is green even if no one is observing it. Such interactions by defined with **unblocking send** events and *receive* events.

send radio warning	warning!	green light is shining	green!
hear radio warning	warning?	I see the green light	green?

Non-blocking send events can be decomposed into:

point to point Emails are often messages from one person to one other unique person.

multicast A traffic light can be seen by many cars.

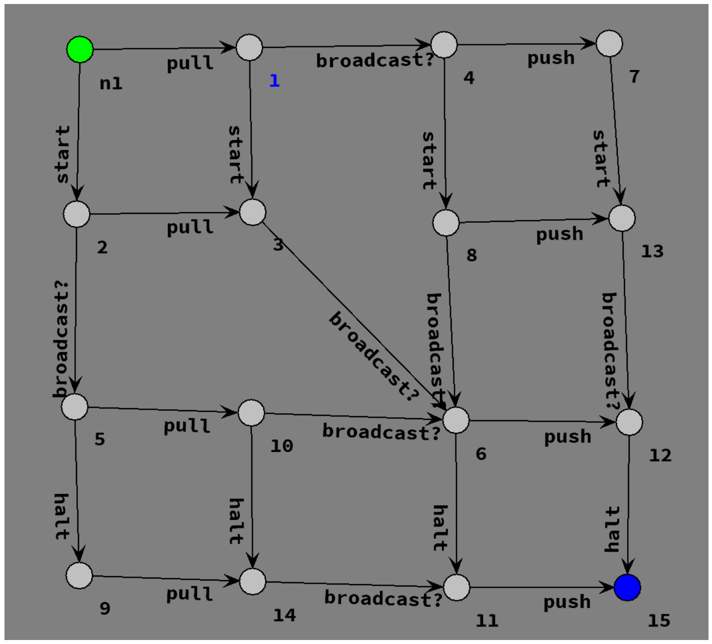
Note: if you are not listening you might miss the warning and if you are not looking you might not see the green light. Listening/looking is formalised as being in a state from which the receive event is enabled. We model processes so that if you are listening/looking then you will hear/see the broadcast event.

Considering systems containing separate processes for Cars and Traffic Lights we often need to consider the case when the Traffic Lights are unique but there are many Cars. The Cars do not interact but if two cars are both looking then they will both see the green light when it is on.

Consequently wo braodcast? events will not synchronise like hand shake events but behave as shown below:

```
(start -> broadcast? -> halt -> STOP)||(pull -> broadcast? -> push -> STOP)
```

The automata can be seen below:

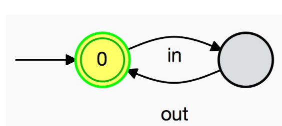


6 Renaming events and simplifying Processes

Frequently when we are thinking of one process it is natural to give an event a particular name. But, when considered from the perspective of another process that may interact with it this name might be confusing. Consequently we introduce ways to rename events. Finally we show how bisimulation colouring can be used to simplify processes. To aid understanding we will consider a simple buffer example in the following.

6.1 Labelling Processes

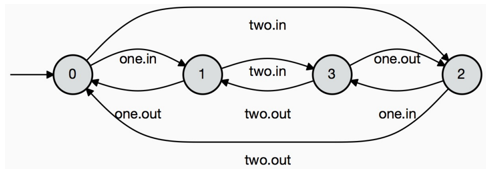
In the following example we make use of a one place buffer `Buf` is a process that when empty can receive some thing `in` and when full can return it `out`.



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.

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



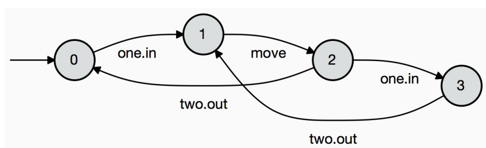
6.2 Event renaming

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}).
```

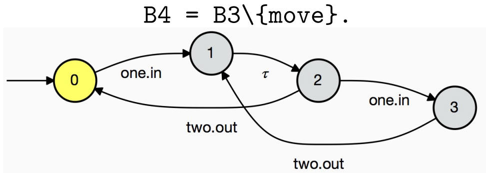


Note that the result is much simpler than the interleaving as the `move` event now can only occur when **both** buffers are able to perform it.

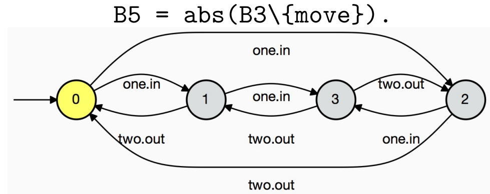
6.3 Process simplification

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}.
```

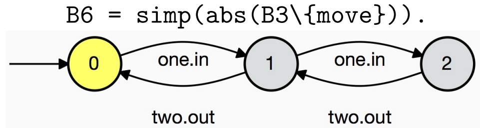



```
B5 = abs(B3\{move}).
```



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.

7 Testing and equality

In this section we are going to introduce a generic notion of equality, testing equality. This can be applied to to any set of things all they need is an operator to compose them and a definition of how to observer them, here the set of things is our set of processes, the operator to compose then is parallel composition and finally when we observer a process all we can see is the complete trace of events that are executed.

When a non-deterministic system is tested it is placed in some context and the combination of the system and text context is executed, but this test must be run a number of times and the set of observations (results) recorded.

Our processes, E, are taken from a set of processes $\mathbb{E}$. A context, $\_ \parallel \mathsf{X}$, consists of a process, $\mathsf{X} \in \mathbb{E}$, run in parallel with the process under test. Let $\Xi \triangleq \{\_ \parallel X | X \in \mathbb{E}\}$ be the set of all contexts. Placing E in context X can be written as $[\mathsf{E}]_{\mathsf{X}}$ or as $\mathsf{X} \parallel \mathsf{E}$. A single experiment consists of observing a single execution of $[\mathsf{E}]_{\mathsf{X}}$ and results in a single trace, taken from a set of possible observations Tr^c, being recorded. For non deterministic processes the experiment must be repeated and a set of observations Tr^c, being recorded.

A specification can be interpreted as a *contract* consisting of the *assumption* that the process will be placed only in one of the specified contexts Ξ and a *guarantee* that the observation of its behaviour will be one of the observations defined by the mapping $O : \mathbb{E} \to \Xi \to \wp Tr^c$. The mapping O defines what can be observed for all processes in any of the assumed contexts. Hence for any fixed Ξ we have a definition of the semantic equivalence of processes.

Definition

$$[\![\mathsf{A}]\!]_{\Xi,O} \triangleq \{(x, o) | x \in \Xi \wedge o \in O([\mathsf{A}]_x)\}$$

and equality is

$$\mathsf{A} =_{\Xi,O} \mathsf{C} \triangleq [\![\mathsf{C}]\!]_{\Xi,O} = [\![\mathsf{A}]\!]_{\Xi,O}$$

Let $\mathbb{E}$ be the set of LTS and Tr^c be the complete traces of an automata then all we need to define to fix our definition of testing equality is the definition of $\parallel$ parallel composition. $\Box$

We have defined parallel composition between handshake events and another definition of parallel composition for broadcast events. Using parallel composition for handshake events the above definition of testing equality has been shown to be the same as the well known Failure equality from CSP. Whereas using parallel composition between broadcast events the above definition of testing equality corresponds to the well known quiescent trace equality.

There are times when specifying, for example, controllers that it is easier to specify more than the item in isolation. For example a traffic light controller should work so that no two law abiding cars would be on the cross road at the same time if they arrived on different roads. Given this situation we do not need to consider the specification as a component of a larger whole as the specification consists of all relevant features. Thus we only need to consider complete trace semantics no Failure semantics.

8 Indexed Process definitions

Basic process definitions you have seen so far a fixed finite set of states. This accurately reflects many situations very and allows easy and complete push button verification. Alternatively when what you are modelling has infinite state or an unknown state size you could use symbolic models but verification frequently requires input from a domain expert and is very time consuming.

The approach adopted here is to define both states and events using an index and limit the size of the index by a parameter. Prior to using a parameter it must be declared and given a fixed value. A finite state approximation of indexed process can be built and size of the approximation can be changed by changing the declared value of the parameter.

8.1 The small world assumption

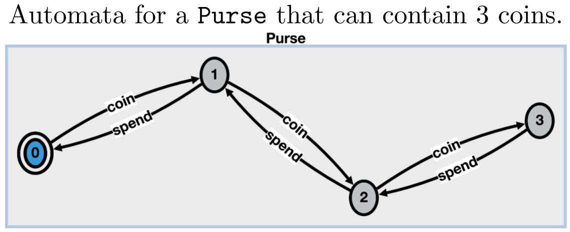
Most program bugs can be found while restricting variables to range over a small domain. Using this assumption we model processes with variables by indexing the processes and restricting the indexes to range over a small domain. Having done this the variables in the state can be removed by instantiating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

More than one parameter can be used to define a process. Once all the parameters are fixed you are back to a basic process with a finite set of states and events. 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.

8.2 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). We will consider a simple `Purse` that can contain a number of `coins`. We define the `Purse` based on a parameter `N` that depicts its size.

Automata for a `Purse` that can contain 3 coins.



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

```
const N = 3
```

Next we define the automata:

```
automata {
Purse = P[0],
 P[c:0..N] = (when c<N  coin -> P[c+1] |
             when c>0  spend -> P[c-1]).
}
```

The first line `Purse = P[0],` defines that the purse is initially empty then the definition `P[c:1..N] =` defines the `N` processes `P[1]`,`P[2]` and `P[3]`

The term `P[c:1..N]` on the left of the equality can be thought of as assigning a value to a variable `c`. The term `P[c+1]` on the right of the equality reads the value in the variable and then "returns" to the purse in a new state where "c:=c+1".

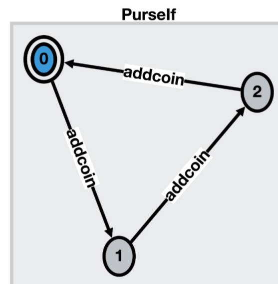
On the right hand side of the equality we define guarded events:

```
when(c<N) coin->C[c+1]
```

the `coin` event will only occur when the guard is true, `c<N` and the event ends at node `C[c+1]`. **Note a guard only applies to one event.** Each time you add a choice you need to add any required guard.

The command `when(c<N) P[0]...` uses process P[0] that is not event prefixed. Hence will this command not compile. Whereas `if...then...else P[0]` will compile and produces the automata displayed.

```
PurseIf = P[0],
  P[c:0..N] =
   (if (c < N) then  addcoin-> P[c+1]
    else    P[0] ).
```

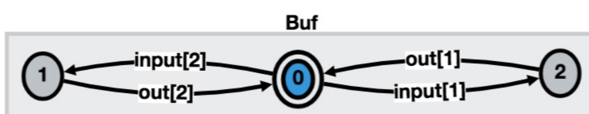


In Section 8.3 we will make use of this property of `if` to simplify the definition of process. An interesying exercise is to try to define the process with out using the `if` command.

8.3 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 is defined by:

```
Buf = input[v:1..N] -> out[v]->Buf.
```

The event `input[v:1..N]` **declares** a new variable `v` and when it executes it **inputs** a value that is assigned to the variable. The subsequent event `out[v]` refers to the previously declared variable `v` and when it is executed it **outputs** the value held in it. So information flows from the `input` event to the `out` event and hence the names.

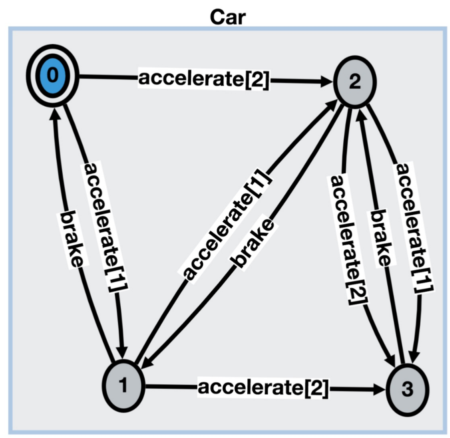
8.4 Cars Example

This example requires both state and event indexing.

> *A car can travel at different speeds and can accelerate at different rates. But no matter how hard it tries to accelerate it can never go beyond its maximum speed.*

In this example the car is indexed by its `speed` and the number on the nodes corresponds to the speed of he car. The rate of acceleration is indexed by `a` and this index is declared in the event `accelerate`. Hence the rate of acceleration can not appear as an index to the car as that is defined prior to the definition of the event.

```
const N = 3
automata }
Car = C[0],
 C[speed:0..N] =
   (when speed<N  accelerate[a:1..N-1]  ->
        ( if (speed+a<N) then C[speed+a]
            else C[N])
     | when speed>0 brake -> C[speed-1]).
}
```



In the example above the Car has a maximum speed of 3 and a maximum acceleration od 2. But when the Car has speed of 2 the effect of accelerating at 2 is only to change the speed by 1.

8.5 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

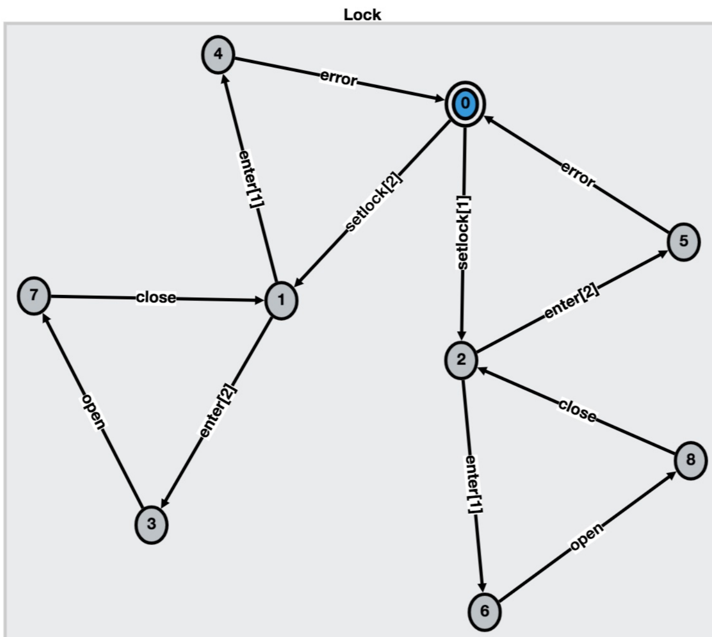
We will demonstrate this with a simple example of a lockable door.

> *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)).
```

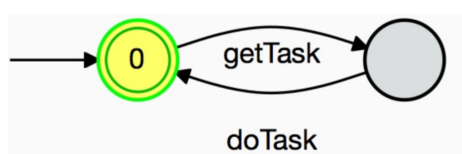
Step 5 Note the value input in the `setlock[i:1..N]` event is stored in the state of the process `L[i]` for subsequent comparison with the value input in the `enter[i:1..N]` event.

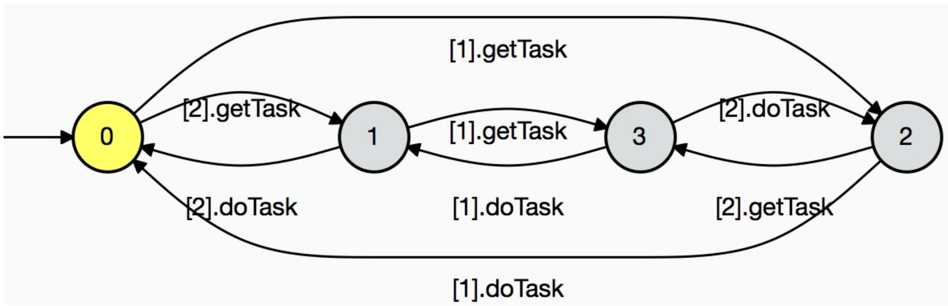
8.6 Indexing concurrent processes.

This means producing an indexed number of concurrent processes but has been implemented in a very restricted. As implemented the indexed process can only communicate with processes that are running in parallel with them. Alas they cannot communicate with each other.

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

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

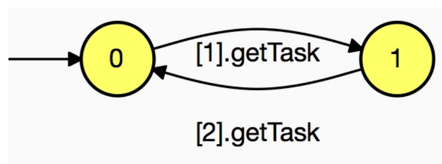




We can add a `Farmer` process to hand out the `Tasks` to the `Workers` in order. Then build a `Farm` composed of the `Farmer` and the `Workers`.

```
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.

9 Petri Nets

What we refer to as a Petri Net is actually an extension of Petri Nets with labels on the transition. This appear in the literature under the name of the *Box calculus*. Net semantics differ from automata significantly in that they offer a **non interleaving** interpretation of parallel processes. Such semantics are frequently referred to as a bf true concurrency semantics. We adopt a definition of parallel composition that originates in the Box calculus and is not a part of main stream Petri Nets.

A Petri Net is a tuple $\mathsf{A} \triangleq (\mathsf{P_A}, \mathsf{S_A}, \mathsf{T_A}, \mathsf{Arcs_A}, \alpha_\mathsf{A})$ where:

$\mathsf{P_A}$ is a set of places,

$\mathsf{S_A}$ is a set of initial Markings, where a Marking is a set of places $\mathsf{M_A} \subseteq \mathsf{P_A}$ and hence $S_A \subseteq 2^{P_A}$

$\mathsf{E_A}$ is a set of final Markings $E_A \subseteq 2^{P_A}$

$\mathsf{T_A}$ is a set of event labelled transitions

$\mathsf{Edges_A}$ Edges join places to transitions and transitions to places. There can be multiple places joined to a single transition and multiple transition joined to a single place.
$\mathsf{Edges_A} \subseteq \{(\mathsf{p},\mathsf{t}) : \mathsf{p} \in \mathsf{P_A}, \mathsf{t} \in \mathsf{T_A}\} \cup \{(\mathsf{t},\mathsf{p}) : \mathsf{p} \in \mathsf{P_A}, \mathsf{t} \in \mathsf{T_A}\}$

α_A is a set of event labels □

As defined Petri Nets offer a richer semantic models that Automata. Specifically Petri Nets do not make use of the interleaving interpretation of parallel composition and hence can distinguish `a->STOP||b->STOP` from a->b->STOP[]b->a->STOP which have identical interleaving semantics. Later we will show how to extend both semantics models so that they are equally expressive.

Let us note by introducing an internal choice operator we need to have a set of initial markings else we would need to alter the semantics of τ event from that used in CCS. (Although CSP dose exactly this we have decided not to.) The introduction of sequential composition as well as action prefixing is facilitated by the explicit definition of successful termination. For convenience we define $Name_A$ a transition naming function $\mathsf{Name_A} : \mathsf{T_A} \to \alpha_\mathsf{A}$

9.1 Appearance

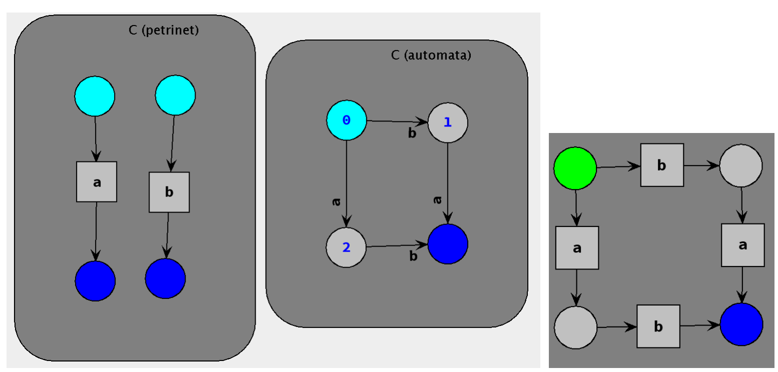
Nets have Places, large circles, not nodes and the initial state of a Petri Net is a set of places each *marked* with a token, a small black circle. The transitions are represented by Boxes and event names. Arcs are added from *pre place*, •t to the Box of transition t and from transition t Box to *post place* t•.

Finite state Petri Nets like finite state automata can be approximation of potential infinite sate processes. The location of the Petri net transitions is the same as that for Automata.

There is a simple relation between finite state sequential automata and finite state Petri Nets. In addition the definition of event hiding and event renaming on Petri Nets and its relation to event hiding and event renaming on automata is quite obvious.

Our tool takes process specifications $\mathcal{P}$ and builds finite state automata P. But in addition it can build Petri Nets from specification Petri($\mathcal{P}$). PertiNets provide a richer process model than automata in that automata.

```
A = a->STOP.
B = b ->STOP.
C = A||B.
```



Two simple processes run in parallel are represented by a single process composed of the interleaving of the events. Whereas a Petri Net can be used to capture the concurrency with a net that has two sf tokens or to model the interleaving with a Net having a single token.

The operations defined on atomic processes can be lifted to operations on Petri Nets. Parallel merge of two processes, parallel composition with no synchronisation is just the union of the component nets. Hence each automata node is corresponds to a Petri Net marking. For the process considered a marking is a pair of places, one element of the pair taken from each component sequential process. With the language considered so far all Petri Nets constructed will be one-safe. That is no more than one token at a time will be places on any of the Petri Net places. Thus each place can be thought to represent a predicate defining the state of one of the component processes, the marking of a place is the assertion that the predicate is currently true. Each sequential process can be can be though to *own* a set of places and transitions. This ownership information can be computed directly from the process term and hence could be added as an annotation to either the Petri Net or to the Automata.

In the diagram above the start node of the automata is coloured light blue and corresponds to the the pair of initially marked places that are also coloured light blue.

9.2 The operational semantics of Petri Nets

We give an operational meaning to Petri Nets by defining their behaviour in terms of how tokens can be moved around a Petri Nets. This is achieved by defining the Token Rule that is a mapping from Petri Nets to automata.

The `TokenRule` maps Petri Nets to automata and the automata can be seen as the operational semantics of the Petri Net. A transition can only be executed when all its pre places are marked and the result of firing a transition is to move the tokens from the pre places to the post places.

This can easily be extended to allow a set of transitions to be executed at the same time when the multi-set union of the places are marked. Thus Petri Nets offer a true concurrent semantics of processes.

If we use the Token rule but ignore the true concurrency $\mathsf{TokenRule}_{\mathsf{It}}$ then the automata it builds are the interleaving semantics that would have been constructed had we built the automata directly from a process specification $\mathcal{P}$, hence we have:

$$\mathsf{TokenRule}_{\mathsf{It}}(\mathsf{Petri}(\mathcal{P})) \sim \mathsf{Automata}(\mathcal{P})$$

Let us write A and B for Petri Nets. We can lift the in-fixed operations Op_a on automata to operations Op_n on Petri Nets so that they obey the following algebraic rules:

$$\mathsf{TokenRule}_{\mathsf{It}}(\mathsf{A}\,Op_n\,\mathsf{B}) \sim (\mathsf{TokenRule}_{\mathsf{It}}(\mathsf{A})\,Op_a\,\mathsf{TokenRule}_{\mathsf{It}}(\mathsf{B}))$$

9.3 From Process specifications to Petri Nets

We use `Proc`, a simple process language to specify the processes we are interested in. The Automata semantics of the operations of the language can be given by, or at least characterised by, gluing pairs of automata nodes together. A single automata node is represented by a marking of a Net. Consequently by defining how to *glue together two markings* we have a definition of the basic process operations on Nets.

```
A = (a->STOP||b->STOP) => (s->STOP||t->STOP).
B = (a->STOP||b->STOP) [] (s->STOP||t->STOP).
```

The sequential composition of two processes, when viewed as an automata can be built by gluing together the end node of the automata of the first process with the start node of the automata of the second automata, see automata `A` below.

SeqNets.png

The gluing together of the end marking of the first Petri Net with the start marking of the second Petri Nets is achieved by the building the product of the two sets of places see Petri Net `A` above.

Choice of two Petri nets can be built by applying the same gluing function but in this case to the start markings of both operands.

ChoiceNets.png

Both for technical ease and to build more compact representations we have chosen to glue the end marking together. This way all automata/ Petri Nets have a single end state/marking.

Our Petri Net semantics distinguishes concurrent processes from their interleaving and thus captures more information implicit in the process terms. It is easy to model event synchronisation in Petri Nets

Event synchronisation of t_1, $\bullet t_1 \xrightarrow{a} t_1 \bullet$ and t_2, $\bullet t_2 \xrightarrow{a} t_2 \bullet$ is a new transition t_3 with the same name and with pre places the union of the component transitions pre places. The post places are constructed similarly.

$$t_3 \triangleq (\bullet t_1 \cup \bullet t_2) \xrightarrow{a} (t_1 \bullet \cup t_2 \bullet)$$

This can easily be seen in the Petri Net `P` shown below.

```
P = (a->x->b->STOP||
      r->x->t->STOP).
```

P.png

9.4 Owned events

Although Petri Nets capture more information than automata they do not capture everything from our process language as can be seen by considering both the Petri Net `P` displayed above and the Petri Net `Q` displayed below. The processes consist of two sequential components and differ only in that events `b` and `t` appear in different sequential components in each process.

```
Q = (a->x->t->STOP||
      r->x->b->STOP).
```

Q.png

As currently defined the Petri Net semantics of both processes are identical yet processes `P` and `Q` are not the same. We describe the difference as being the events `b` and `t` are *owned* by different sequential processes and this information is lost in the Petri Net semantics as defined.

Clearly we can extend both automata and Petri Net semantics to record the ownership of events. This way we have *owned automata* and *owned Petri Nets*. The two very different representations are now semantically equivalent. We show this my constructing mappings from *owned Petri Nets* to *owned automata* and back again. The mapping from *owned Petri Nets* to *owned automata* can be given simply by extending the Token Rule to preserve the ownership of events whereas the mapping in the other direction is given next and is new.

9.5 Owners Rule - from Automata to Petri Nets

Building Petri Nets from automata with located transitions Owners Rule works as follows:

Automata alone do not preserve concurrency information. By recording which process **owns** any event the **owned automata** captures all the information needed to build a Petri Net that displays the concurrence information.

From the process term define a set of primitive locations or owners, one for each sequential process. Build a Place for each primitive location and by projection of the automata onto the location build a sequential Petri Net. In essence events at a location containing own are preserved all other events are hidden. Finally the set of projected Nets can be composed in parallel synchronising events that have been projected onto more than one location.

This approach works but we need to restrict event synchronisation to be between transitions that originate from the same automata event and not just having the same label. This cn easily be achieved by first "tagging" the edge labels of the automata to ensure that all events have distinct labels. Then after the Petri Net has been built removing the tags from the transition labels.

One further issue is that the algorithm works only for processes with a single start node, i.e. non initial non-determinism. The algorithm can be extended to cope with initial non-determinism in two steps:

1. prefixing the automata with a single event, **"star"**
2. remove the **"star"** transition from the Petri Net thus revealing the multiple root markings.

by

The algorithm we use is:

```
for each primative location O
push rootnode onto stack
 while stack not empty
   pop node from stack
   if not processed
     collect set S of all nodes connected (undirected) by edges not at O
     build Place  and mapping from all nodes in S to Place
     for all nodes connected by edges at O push toNode onto Stack
push rootnode onto stack
 while stack not empty
   pop node from stack
    if not processed
     for all nodes connected by edges at O build transition between mapped Places
```

The round trip from an automata to a Petri Net (Owner Rule) and back (Token Rule) produces a bisimilarly equivalent automata. This has been tested for thousands of automata.

9.6 Two semantically equivalent views of the same process specification

From this point we will assume that our semantics models are annotated by *ownership* and that the Token Rule and the Ownership Rule both preserve the ownership information. Mathematically speaking they are isomorphic. The advantage of being able to model the same process in two quite distinct semantics allows us to define operations on which ever semantics is most natural and then to project the operation to the other.

Process equivalences, bisimulation, failures equality and complete trace equality can all be defined on automata and easily extended to our located automata by the simple trick of viewing the location s a part of the event name.

Whereas event refinement can be defined on Petri Nets. Because of the addition of location the two models are equivalent but nonetheless trying to define event refinement on located automata or testing equivalence on Petri Nets is far from easy. Several definitions of process equality have been defined on Petri Net semantics but they are difficult to understand and very hard to compare in detail.

9.7 Event Refinement

We can define how to refine a single event in a process (PetriNet) into a whole process (Petri Net). Refinement is constructed by applying the *glueing* function used in the definitions of sequential composition and external choice. In essence all that is needed is to glue the pre places of an event to the root of the process that replaces it and then the post places of the event to the end marking of the process.

```
A = a->b->c->STOP.
R = r->STOP||s->STOP.
N = A/{R/b}.
```

Ref$imp.png

It is important to realise that event refinement is not defined as syntactic substitution. To do so would require the syntax to be modified and even then it would either not be able to apply it to processes built via abstraction and simplification or would have to re-compute the process from new syntax. Although we want the same result as if it were constructed via syntactic substitution, this is clearly the case with the example above where the net `N` is clearly what we would expect from `(a->STOP)=>(r->STOP||s->STOP)=>(c->STOP)`.

A more complex and interesting example is given below. It shows that event refinement can construct nets with markings that are not sets but are finite multi sets. Such nets are referred to as n-safe nets. Up to now all nets have been one-safe nets sometimes referred to as predicate nets.

```
B=b->B.
E=((s->STOP) || B).
D = c->STOP||c1->STOP.
X = E/{D/b}.
```

Xwings.png

To check your intuitions about Petri Nets ask your self it the two *"wings"* places which are always marked are just an artefact of our algorithm and are semantically redundant or not?

The execution of either `c` or `c1`, in Petri Net `X` above will result in two tokens on the same place. That is `X` is not one-safe. Having built the Petri Net by refining a loop event we can apply the token rule to build the automata (annotated with event owners) and then apply the OwnersRule to rebuild a Petri Nets.

```
B=b->B.
E=((s->STOP) || B).
C=c->STOP.
C1=c1->STOP.
D=(C||C1).
X=E/{D/b}.
Y=ownersRule(X).
```

Ywings.png

Interestingly the Owners Rule, as defined, dose not rebuild the n-safe net we started from. It builds a semantically equivalent net that is 1-safe, see above. It would be interesting to consider if the algorithm could be amended to build n-safe nets.

The same process can be represented by the Petri Nets `X` and `Y` given above. Which representation is *better* is a matter of personal taste. The Petri Net `X` is smaller which is good but on the other hand not being one-safe prevents us from interpreting places as predicates and the marking of the place as asserting the truth of the predicate it represents.

That the Owners Rule converts n-safe nets into "equivalent" one-safe nets came as a surprise and hence prompts the questions: is the rule correct? and what do we mean by correct?

There are a vast number of interleaving process equivalences in the literature and as non-interleaving semantics are much richer even more non-interleaving semantics can be defined. Here we justify our semantic equivalence by the construction of a plausible, in our opinion, testing semantics. We wish to enrich our interleaving semantics to capture the non-interleaving behaviour while not loosing the testing semantics defined on automata. To achieve this we extend the event semantics to model the ownership (or location) of the events. By this we can model processes both as automata and as Petri Nets and we have the Token Rule to build the automata from the Petri Net and the Owners Rule to build the Petri Net from the automata.

Much work has been undertaken to define a non-interleaving semantics on Petri Nets. It can be seen form the literature that there are many ways to construct semantics equivalences based on the shape of Petri nets. But we have neither found, nor can see how to construct such a definition our self, that would equate Petri Nets such as X and Y given above.

9.8 Advantages of the mixed semantics of owned events

The semantics of processes, where events are annotated with ownership, can be represented both as automata and as Petri Nets. Properties can be defined on which ever semantics is most natural and then *lifted* to the other semantics.

Petri Nets can be very much smaller than automata and make concurrency easily seen. Petri Nets also support event refinement whereas it can not be defined on the unowned interleaving semantics of automata.

Automata support a rich set of equivalences that can be justified by plausible testing semantics and existing definitions of event hiding and simplification can readily be applied to the automata annotated with ownership. Further reachability is largely obscured by Petri Nets and easy to see in automata.

ReachNets.png

In the Petri Net S above the events y and c are unreachable. This is immediately apparent when converted to an automata S. Subsequently applying the OwnersRule to the automata builds the Petri Net R in which the unreachable events do not appear.

Because we can freely map between Petri Nets and Automata that are annotated with Ownership we can construct definitions, and implementations on which ever is easier and then map the result to other. For example Use the **Token Rule** to map a Petri Net to an Automata then simplify the automata and use Owners Rule to map the result back to Petri Nets.

Using owned event semantics we can *lift* failure semantics nd hence Failure equality from automata to Petri Nets. Given two Petri Nets P and Q we define them to be Failure equivalent precisely when their automata semantics are Failure equivalent.

$$\mathsf{P} =_{\mathsf{F}} \mathsf{Q} \triangleq \mathsf{Token}(\mathsf{P}) =_{\mathsf{F}} \mathsf{Token}(\mathsf{Q})$$

This lifting can naturally can be applied to any process equivalence.

10 Semantic equivalence of processes

The three process equivalences bisimulation, $\sim$, failure equality $=_F$ and complete trace equality $=_{CTr}$ satisfy the following laws:

$$\mathsf{A} \sim \mathsf{B} \implies \mathsf{A} =_{\mathsf{F}} \mathsf{B} \implies \mathsf{A} =_{\mathsf{CTr}} \mathsf{B}$$

An equality is a relation between items from some domain, here we are interested in process equality so the domain is the set of all processes. Equality has the following properties:

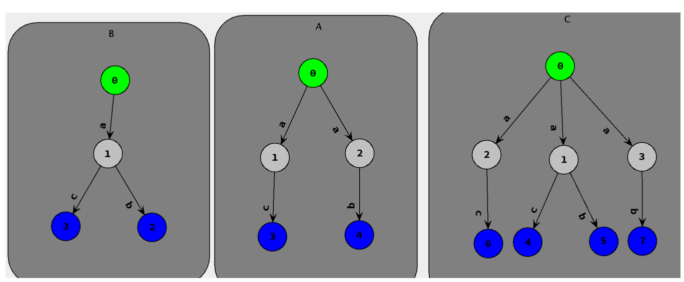
1. an equivalence relation, a reflexive, symmetric and transitive relation
2. substitutive and
3. where equivalent objects are indistinguishable.

Indistinguishable processes can be formalised as *testing equivalent*. Substitutive can be formalised are congruent w.r.t. process operators, from a practical perspective the most important being parallel composition.

10.1 Example processes

Process equalities can be checked by the tool by defining operations after the processes have been defined and the automata built. Not equivalence is defined by prefixing one of the equality operators with a !.

```
processes {
 A=(a->b->STOP|a->c->STOP).
 B=a->(b->STOP|c->STOP).
 C=(a->b->STOP|a->c->STOP|
       a->(b->STOP|c->STOP)).
 }
 automata A,B.
 operation {
  A # B. A !* B.
  A * C. A !~ c.
 }
```



The operations state that A and B are trace equivalent but not failure equivalent. Whereas A and C are Failure equivalent but not bisimular equivalent.

A might perform an a event and then fail, when offered to synchronise with a b event. In contrast whenever B performs an a event it will never refuse to perform an b event. Consequently the two processes are distinguishable by testing.

Although A and C are not bisimular equivalent they can not be distinguished by any test.

Some relations between equalities can be used to understand certain aspects of the equivalences. Bisimulation is strictly stronger that Failure equality which in turn is strictly stronger than Complete trace equality.

$$\mathsf{P} \sim \mathsf{Q} \Rightarrow \mathsf{P} =_F \mathsf{Q} \Rightarrow \mathsf{P} =_{Tr^c} \mathsf{Q}$$

A deterministic process P is failure equivalent to another process only when it is bisimular to it.

$$det(\mathsf{P}) \Rightarrow (\mathsf{P} \sim \mathsf{Q} \Leftrightarrow \mathsf{P} =_F \mathsf{Q})$$

Consequently if one process is deterministic then failure equality, refinement can be computed easily be computing bisimulation.

Weak equalities, refinements can be computed by first computing the weak semantics and then, computing the strong equality or refinement on the weak semantics we have the weak semantics on the original automata. A useful analytic technique is to start with a detailed deterministic process, hide some events and simplify the process using bisimulation. This builds a more abstract representation that may well be deterministic, if it is not deterministic this tells you some thing about your detailed process.

11 Model checking our theory

Although we reason about processes by rewriting the operational semantics of the processes and not by rewriting the equational axioms we can make use of the equational axioms to test the theory implicit in the tool.

There are two sections we can use, operations and equations. The operations section consists of a set of terms: `A~(B=>(a->STOP)).` consisting of an in-fixed equality, bisimulation, failure or testing and two operands that are terms containing previously defined processes, (closed terms). All the ground equations in the operations section are evaluated and any errors reported

The equations section is similar to the operations but now the operands can contain process variables, that is open terms : `X~(Y=>(a->STOP)).` To evaluate one of these equations the variables need to be replaced by actual processes. These processes are taken from the set of processes defined in the processes section.

Evaluating the given equation when the processes section contains 20 processes will result in 400 (20^2) ground equations being tested. By this technique it is feasible generate and evaluate thousands of tests. This test generation is far from perfect and results in multiple instances of the same test being generated. Ways to improve this are known but have yet to be implemented.

Adding implication to the language used in operations and equations will allow congruence relations to be tested an further by adding failure refinement as well as failure equality will allow the Galois Connections mentioned to be tested.

12 Syntax

Processes are first defined then any you wish to view as an automata appear in the list see below:

```
const  Max = 2
processes  {
A = a ->STOP.
B = b -> STOP.
C = A || B }
automata B,C.
```

Event type	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!`

In above three processes are defined but only two displayed. The constant Max is bound to the value 2.

<table>
<tr><td></td><td>atomic</td><td>indexed</td></tr>
<tr><td>Prefixing</td><td><code>A = act->P</code></td><td><code>if (i<N) then (act[i]->P[i+1]) else P[0]</code></td></tr>
<tr><td></td><td></td><td><code>Money = C[1],</code>
<code>C[i:1..N] = (when(i<N) coin->C[i+1]</code>
<code> |when(i==N) coin->C[1]).</code></td></tr>
<tr><td>Choice</td><td><code>A = a->P|b->Q</code></td><td><code>Farmer = ([i:0..N].task ->W[i]),</code>
<code>   W[i:0..N] = ([i].end->Farmer).</code></td></tr>
<tr><td>Labeling</td><td><code>lab:P</code></td><td>see below</td></tr>
<tr><td>Parallel</td><td><code>A = (P||Q)</code></td><td><code>Workers = (forall [i:0..N] ([i]:Worker)).</code></td></tr>
<tr><td>Relabeling</td><td><code>P/{new/old}</code></td><td><code>P/{new[i:0..N]/old[i]}</code></td></tr>
<tr><td>Hiding</td><td><code>P\{act}</code></td><td><code>P\{act[i:0..N]}</code></td></tr>
<tr><td>Event Refinement</td><td><code>P/{Q/e}</code></td><td>Replaces event <code>e</code> by process <code>Q</code></td></tr>
<tr><td>Sequential Composition</td><td>A => B</td><td>Behave like A and when A terminates behave like B</td></tr>
<tr><td>Internal Choice</td><td>A+B</td><td>Can behave like A or like B but which is chosen internally.
Not chosen externally by event synchronisation.</td></tr>
</table>

For processing automata:

abstraction	`abs(P)`	fair removal of τ events
abstraction	`abs{unfair}(P)`	unfair removal of τ events
hiding	`hide{S}(P)`	$\triangleq$ `abs(P\{S})`
simplification	`simp(P)`	for the simplification of automata
hiding index `x`	`R = R1${x}`	builds symbolic automata

Bisimular	$A \sim B$	A and B are bisimular
Complete Trace	A#B	A and B are complete trace equivalent
Failure	$A * B$	A and B are failure equivalent
Negation prefix	!_	$A! * B$, A!#B, $A! \sim B$

For evaluating equations use the following:

```
operations  {
(A||B)=>C ~ (A=>C)||(B=>C)
 }
```

When there are P distinct processes defined then equation

```
equations {
X=>(Y[]Z) # (X=>Y)[](X=>Z)
}
```

with its 3 variables will generate P^3 equations.

References

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, 1990.

[BW98] Ralph-Johan J. Back and J. Von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[dNH84] R. de Nicola and M Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34, 84.

[Hen88] M Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

[ReS04a] S. Reeves and D. Streader. Unifying state and process determinism. Technical report, University of Waikato, http://hdl.handle.net/10289/1001, 2004.

[ReS04b] Steve Reeves and David Streader. Atomic Components. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004: First International Colloquium*, volume 3407 of *Lecture Notes in Computer Science*, pages 128–139. Springer-Verlag, September 2004.

[ReS09] Steve Reeves and David Streader. Guarded operations, refinement and simulation. In *Proc Fourteenth BAC-FACS Refinement Workshop (REFINE 2009), doi:10.1016/j.entcs.2009.12.024*, volume 259 of *Electronic Notes in Theoretical Computer Science*, pages 177–191, Eindhoven, The Netherlands, 2009. Elsevier.

[ReS11] Steve Reeves and David Streader. Contexts, refinement and determinism. *Science of Computer Programming, DOI: 10.1016/j.scico.2010.11.011*, 2010.

[Tay99] P. Taylor. *Practical Foundations of Mathematics.* Cambridge University Press, 1999. Cambridge studies in advanced mathematics 59.

[vG90] R. J. van Glabbeek. Linear Time-Branching Time Spectrum I. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, LNCS 458, pages 278–297. Springer-Verlag, 1990.

[vG93] Rob J. van Glabbeek. The Linear Time - Branching Time Spectrum II. In *International Conference on Concurrency Theory*, pages 66–81, 1993.

13 TLA+ Overview

In what follows we will refer to a formal step that moves from an abstract specification to more concrete specification as a refinement step and the reverse step as an abstraction step.

TLA+ is mainly used to model shared memory concurrency but not exclusively. A central aspect of TLA+ specifications is use of logic to specify assignment and behaviour.

The basic building blocks of TLA+ are Modules. Modules contain two basic components variables and actions. The state of a module is an evaluation, a variable value mapping $State : Var \rightarrow Val$. That is the state is a context in which expressions can be evaluated. Commonly State is modelled as a record where the names of the fields in the record are the names of the variables. The state of a module can be changed by executing one of its named actions.

TLA+ has no parallel composition operator Modules are commonly defined to model both the process under construction and the world around it. In Process algebra an observable event needs to synchronise with its counterpart in another process, as such the two component events can be thought of as *half events*. This is not true for the actions of TLA+.

Assignment is defined as a predicate by introducing primed representation of pre state. This way `x:=x+1` becomes `x'= x+1`. Converting as much as you can into predicates makes the use of a theorem prover more effective.

Actions consist of a guard and a set of assignments. Consequently the semantics of an action can be given as a named relation over the lifted state, $State^{\perp}$. To facilitate reasoning actions are defined as a conjunction of predicates. Hence an action is a named predicate.

To define a TLA+ action that sends data we write `Send(d)` and

$$\mathsf{Next} \triangleq \big(\exists \mathsf{d} \in \mathsf{Data} : \mathsf{Send}(\mathsf{d})\big) \wedge \mathsf{Rcv}$$

Although this action resides solely on the sending process it may be in a Module that specifies both the sending and the receiving processes. The receiving action makes no explicit reference to the value `d` because TLA+ uses shared memory all that need happen is that

$$\mathsf{Send}(\mathsf{d}) \triangleq \mathsf{var}' = \mathsf{d} \wedge \ldots$$

the sending event stores the data sent in a variable and then the receiving event is free to read it.

Modules declare constants, variables and inner Modules The variables are *public* in that they are visible to an outer Module. We can define the behaviour of a Module using a Next state predicate and an Init predicate

$$\mathsf{Spec} \triangleq \mathsf{Init} \wedge \mathsf{Next}$$

Note the Next predicate is the critical component that defines what next step the Module can take given its current state.

The variables in a Module can be hidden, made *private* as we will see shortly.

A module `N` can declare a predefined module `Chan`

$$\mathtt{InChan} \triangleq \mathtt{INSTANCE\ Chan} \ldots$$

An action `get` in `M` can include one of the actions in `InChan` simply by adding a conjunction the action `input` from the module `InChan` by:

$$\mathtt{get} \triangleq \mathtt{InChan!input}(\ldots) \wedge \ldots$$

13.1 Partial specifications

To create partial specifications that is to leave some thing unspecified define what is to be unspecified as a parameter. This is the same as indexed processes.

Let us assume that Module `Chan` contains a variable `q`. Now we can declare a parametrised Module `Chan(q)` where the parameter will instantiate `Chan`s variable `q`

$$\texttt{Inner(q)} \triangleq \texttt{INSTANCE Chan} \ldots$$

13.2 Variable hiding

Another use of parametrised Modules is when to hide a variable or put another way make a variable *private*. For example if we want the variable `q` of the inner module `Chan` to be *private* then we also declare

$$\mathsf{Spec} \triangleq \exists \mathsf{q} : \mathsf{Chan!(q)!Spec}$$

Note this kind of variable hiding is not a mainstream idiom of TLA+ but is essential to our visual verification:

> *In fact, for most applications, there's no need to hide variables in the specication.* [**?**, page 41]

If the declared modules state is treated as private, not accessed directly then the action `get` is a superposition refinement of the action `input`. Hence with some restrictions `INSTANCE` can be used to define superposition refinement between modules (Event-B).

Modules with private state are like processes, actions are like process events and a modules *Init* action can take the place of a processes start state.

13.3 Composition and Decomposition of specifications

Let a system be composed of two components, two processes or Modules. We might, quite reasonable want to reason in various related styles. We might start with an abstract system specification and want to decompose it into two separate parts alternatively we might go in the reverse direction and start with the specification of two components and want to compose them to form a system specification. Event B provides a way to verify refinement steps having previously defied both the abstract system specification and the more concrete composition of two components. This technique is agnostic as the direction you are taking.

TLA+ formalises the behaviour of composition to two Modules as the disjunction of the behaviour of the component Modules. Slight variants are able to capture an interleaving or true concurrent behaviour.

13.4 Further Comparison with processes

Process algebras are built from of a set of events and process operators.

A binary process operator is a bit like a parent module declaring as an instance two child modules. Such a declaration must give the parent access to the state of the child but the child modules will be private to the parent. In particular both the state and actions of the children can not be seen outside of the parent.

Process algebraic parallel composition is a binary process operator. Event synchronisation can be formalised by introducing an action that contains the conjunction of the two synchronised actions. Non synchronised child events are either lifted to the parent by introducing a parent action that simply includes the child action or can be blocked by not lifting them.

In TLA+ there is no counterpart to event hiding or testing equivalence. TLA+ makes use of temporal logic to specify the behaviour of a Module there is no ability to hide parts of an implementation or detailed specification to build a more abstract specification. Consequently if we were to add shared state to the processes our tool uses then the style of analysis would eb quite different to that in TLA+.

14 Tool Development

Currently toll builds finite state automata and Petri Nets. We hope to add symbolic semantics by incorporating the theorem prover Isabel.

14.1 Technical overview

Ongoing review and discussion!

1. Compile each process separately and only recompile if that part of the document has changed.
2. Isabelle is now designed for *proof as a service* with Isabelle running in docker containers in the background. Alas this appears to be work in progress and we might have to stick to Z3 at the moment.

14.2 TODO

1. Add
   (a) co-events
   (b) probabilistic choice (may well postpone till later)
2. **Symbolic processes closely related to Petri Nets shown above :**
   (a) Define $\_ \parallel \_$ on symbolic processes. Integrated with Petri Nets. To apply index hiding after abstraction need to track indexes
   (b) Define $\mathsf{S2A}$ a function that maps symbolic processes to atomic processes.
   (c) **Add Process invariants** for Z3
   (d) Use Isabelle in place of Z3 **but only if headless Isabelle ready**

   Test like Petri Nets using equalities: $\forall \mathsf{P} : \mathsf{P} \sim \mathsf{S2A}(\mathsf{P\$\{x\}})$

   $\forall \mathsf{P}, \mathsf{Q} : \mathsf{P} \parallel \mathsf{Q} \sim \mathsf{S2A}(\mathsf{P\$\{x\}} \parallel \mathsf{Q\$\{y\}})$
3. **Add probabilities**
   (a) probabilistic choice - reactive
   (b) abstraction will produce generative probability
   (c) could build lifting into abstraction - start set of nodes not just node?

4. **Enhanced debugging:**
   (a) Show shortest trace to expose inequality
   (b) Visualise bisimulation state equality between different automata or Petri Nets
   (c) keep appending to console output and add button to clear.
   (d) debugging tool tips when you hover over:
      i. events ⇒ evaluation of variables in scope
      ii. nodes ⇒ evaluation of variables in scope
5. Isabelle extension overview:
   (a) Isar text managed by Jedit (JEdit is a plugin extensible editor designed for the construction of IDEs). Isabelle/Scala is used to manage to the JEdit
   (b) Could define functions, bisimulation, abstraction, automata building in Isabelle
   (c) Could keep the algorithms in Java and just use Isabelle to do symbolic to atomic conversion.
6. **Gui to define event renaming and synchronisation - needs Petri Nets**
7. **Syntax alternatives?** Variables are currently named and can be read but writing to variable is done by writing to the index at the same location at which the variable is declared.

   Alternatively declare variables in a process and:
   (a) s and use a common variable assignment syntax `x := x+1` or
   (b) use Z specification style primed variables for post state `x' = x+1`
8. **Add Process invariants**, need to look at TLA+ syntax / parsing + import libraries
9. Support for SDN may be TLA+

14.3 Link with TLA+

TLA+ has

1. Theories + data types
2. syntactic substitution
3. term simplification
4. expansion to finite state model.

Process Tool has

1. visualisation
2. parallel composition + synchronisation
3. abstraction
4. simplification bisim + failure + trace

Stepped development of a Bridge between TLA+ and PA.

Step Zero sequential processes with atomic events to TLA and back. Not sure if this should be to TLA text or parse tree!

1. `P2M`: $Process \rightarrow Module$ Process mapped to Module with additional State variable.
   This will add the ability to check process satisfies temporal logic specification

2. `M2P`: $Module \rightarrow Process$ Module mapped to atomic automata using built in expansion

Add tests to test directory for any atomic processes P:

1. `M2P`(`P2M`(**P**)) $\sim$ **P**

Tests added to repository can be refactored by adding new versions of P. (Might be worth defining algebra over process variables and set of processes to instantiate process variables)

Step One Define $\|_M$, parallel composition of TLA+ modules:

1. Component modules as INSTANCE declaration.
2. Event synchronisation defined as conjunction of component actions
3. default lifting of non synchronising actions.

Add tests for any atomic processes P and Q:

1. `M2P` (`P2M`(**P**) $\|_M$ `P2M`(**Q**)) $\sim$ **P**$\|$**Q**

Second step Change PA with indexed state and events to make use of TLA+ numbers

1. add import statement to PA
2. change PA to use TLA code for parsing and simplification of numbers:

Add tests using indexed processes

Third (2a) Code `M2sP` based on `M2P` but to return a symbolic numeric process:

1. `M2sP`: $Module \rightarrow Process$ Module mapped to symbolic automata.

For any symbolic processes P and Q use TESTS:

1. `M2sP`(`P2M`(P)) $\sim$ P
2. `M2sP` (`P2M`(P) $\|$ `P2M`(Q)) $\sim$ P$\|$Q

Forth step use existing TLA code to add data theories to PA

1. extend PA to parse theories and display
2. extend `M2sP` and `P2M`

Fith (3a) step add symbolic abstraction to PA

1. **implement** $sabs(\_)$ **symbolic abstraction using TLA+ code:**
2. add tests
   $S2A\{x\}(sabs(P\$\{x\})) \sim abs(P)$

Sixth (3b) step add symbolic bisimulation PA

1. **implement** $ssimp(\_)$ **symbolic bisimulation using TLA+ code:**

2. add tests
   $S2A\{x\}(ssimp(P\$\{x\})) \sim simp(P)$

Bits and pieces.

1. Find TLA+ parse tree for module, TLA_{PT}
2. build a TLA_{PT} for a Petri Net
3. Find TLA+ term evaluation (tree for an action)
4. computation options:

   (a) implement symbolic abstraction
   (b) implement symbolic equality
   (c) expand

   (a) expand
   (b) atomic abstraction
   (c) atomic equality

14.4 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.

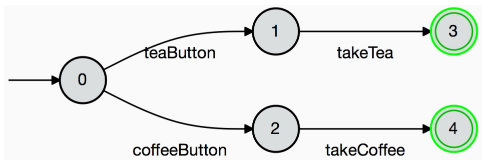
1. **Add support for Event B style reasoning.**
2. **Add probabilistic choice.**
3. **Add interrupts. see after "\end{documnet}"**
4. **Add hierarchical processes (where the state of one process becomes a whole process) the result will be include adding signals that will better model interrupts.**
5. **Add δ events - unobservable and blocked + model known**
6. **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**

15 Statefull syntax

1. Declare constants in each process
2. Declare variables in each process
3. Process = process name, indexed process or **assignment statements**
4. **guard event** $->$ **process** OR **event** = **guard process**

15.1 Translation:

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



Introduce a variable `St` to distinguish different processes (nodes).

```
CM = const N;
    var St:0..N;
    init = St:=0;
    events =
      when(St==0) teaButton->St:= 1
     |when(St==0) coffeeButton->St:= 2
     |when(St==1) takeTea->St:=3
     |when(St==2) takeCoffee->St:= 4.
```

```
CM = const N;
   var cnt:0..N;
   init = cnt:=1;
    teaButton = when(St==0) St:=1
    coffeeButton = when(St == 0)  St:=2
    takeTea = when(St == 1)  St:=3
    takeCoffee =when(St == 2)  St:=4.
```

1. `C[1]` into initialise first index to `1`
2. `C[cnt:1..N] = ...` into declaration of `cnt` at first index and rhs as process definition
3. `C[cnt+1]` into process after `cnt:=cnt+1`.

```
Money = const N;
    var cnt:0..N;
    init = cnt:=1;
    events =
      when(cnt<N) coin -> cnt:=cnt+1
     |when(cnt==N) coin -> cnt:=1.
```

```
Money = const N;
   var cnt:0..N;
   init = cnt:=1;
   coin = when(cnt<N) cnt:=cnt+1
         |when(cnt==N) cnt:=1.
```

16 Symbolic Processes mostly 2b added

Simple process specifications, those with out indexes, are "atomic" specifications that produce atomic automata. An atomic automata has atomic transitions with a single atomic name and a finite set of atomic nodes each representing one state.

Indexed process specifications represent processes with variables, the indexes, and may be infinite state. The indexed specifications are expanded to a finite state approximation of the underlying and potentially infinite state automata. They do this using defined bounds such as `const N = 4`.

Alternatively we could define symbolic automata, automata with variables. The state of the process, represented by a symbolic automata, is the pair (n, μ) where μ is an evaluation i.e. a mapping from variables to values and n is a node of the automata.

$$\mu : Var \to Val$$

Hence the node of a symbolic transition only tells you part of the state of the process. When a event of a symbolic process is actually executed the state of the process both moves from the pre-node of the transition to the post-node of the transition and the evaluation of the variable changes as defined by the assignment. Symbolic execution represents a whole set of actual executions.

The symbolic transitions need to be annotated with its name plus a boolean guard and an assignment. Let a transition $\mathsf{t1} \triangleq (\mathsf{n1}, \mathsf{g1}, \mathsf{ev1}, \mathsf{a1}, \mathsf{m1})$ be represented as

$$\mathsf{n1} \xrightarrow{\mathsf{g1,ev1,a1}} \mathsf{m1}.$$

Both the guards $\mathsf{g1}$ and assignments $\mathsf{a1}$ may contain the process variables. Note symbolic Petri Nets can be built by adding the variables from a process to the places that represent the state of that process.

Our tool takes process specifications $\mathcal{P}$ and, by default, builds finite state automata P. But now we want to prevent the expansion of indexes and build symbolic automata from the specification $\mathcal{P}\$\{\mathsf{x}\}$.

The execution of a guarded event can only occur when the guard is true and then the assignments of the transition are applied hence building another evaluation. Two essential functions needed in the definition of the application of assignments a, writen $\_@\mathsf{a}$ are:

1. syntactic substitution and
2. simplification.

Simplification certainly needs to be out soured to proof services as many decades of work has gone into developing such algorithms. And doing this allows for the development of theories of specific data types that include both their definition and the proof of rules of inference used in simplification. Essential this provides an extensible proof engine.

Syntactic substitution look easy but either you need to; hard bake in the language the data types used or out source both parsing and substitution. *Initially hard bake in integers, lists and sets?*

Reasoning about symbolic processes requires that we can concatenate sequential transitions. This can be achieved using two basic standard techniques, *symbolic execution* and backward reasoning via *Hoare Logic*. Hoare Logic tells us how to compute the weakest precondition prior to an assignment ass of a post condition bg and is written $\mathsf{bg}@\mathsf{ass}$. Whereas symbolic execution tell us how to combine the sequential execution of two assignments $\mathsf{a1}$ and then $\mathsf{a2}$ into a single assignment that is written $\mathsf{a1}@\mathsf{a}_2$

Let assignment ass be a set of assignments all applied in parallel $\{x := E, y := F\}$ and let semicolon be used to compose assignments sequentially.

Compute the weakest precondition of an assignment for a known post condition:

$$\{P[E/x]\}x := E\{P\}$$

In our notation the precondition $P@\{x := E\} \triangleq P[E/x]$

Use symbolic execution to remove the sequencing of assignments.

$$x := F(x, y); y := G(x, y) = \{x := F(x, y), y := G(F(x, y), y)\}$$

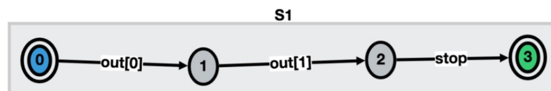
In our notation $\{x := F(x, y)\}@\{y := G(x, y)\} \triangleq \{x := F(x, y), y := G(F(x, y), y)\}$

17 Index freezing (z3 - headless Isabelle)

Closely related to Petri net Construction and Token Rule.

The default interpretation, semantics, of an indexed process definition is its finite state expansion using the declared values of the parameters. An alternative interpretation is to not expand some indexes but to leave them as unknown and define symbolic events.

```
S1 = X[0],
  X[x:0..N]=(when (x<N) out[x]->X[x+1]|
             when (x==N) stop->STOP).
S = S1${x}.
```

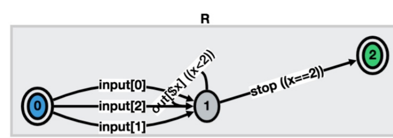


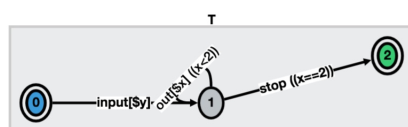
1. assignment `x:=x+1` is missing
2. presentation order *guard name assignment*
3. 2 should be `N` and `$x` should be `x`

For the parallel composition of indexed processes we will need to α convert index names to prevent name clashes. This can be achieved by profixing an index name with its process name, hence `x` above would become `S1.x`

```
R1 = input[y:0..N] -> X[y],
  X[x:0..N] = (when (x<N) out[x] -> X[x+1]|
               when (x==N) stop ->STOP).
R = R1${x}.
T = R1${x,y}.
S = R1${y}.
```

1. `S = R1${y}` currently fails to build.

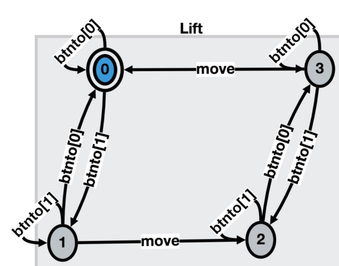




In the above example both indexes `x` and `y` are needed.

```
Lift = L[0][0],
  L[x:0..N][to:0..N] = btnto[i:0..N] -> L[x][i]   |
        when (x!=to) move ->L[to][to].
```

All three Indexes `x`, `to` and `i` are needed.



Need a symbolic expansion mapping, `S2A` that takes as input a symbolic automata and returns an atomic automata.

```
R = R1${x}.
T = S2A(R{x}).
```

The result of expanding all variables should be the same as building the atomic automata from scratch, hence `R`∼`T` in above. (Note building from the automata/Petri Net allows interacting with the diagram)

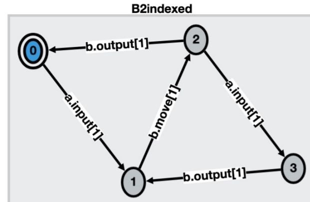
17.1 Need for index freezing

Take a look at the examples below.

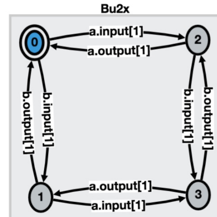
```
const N = 1
automata {
Buf1data = input[i:1..N] -> F[i],
   F[d:1..N] = output[d] -> Buf1data.
Buf2a = (a:Buf1data)/{b.move[x:1..N]/a.output[x]}.
Buf2b = (b:Buf1data)/{b.move[x:1..N]/b.input[x]}.

B2indexed = Buf2a||Buf2b.
Bu2x = B2indexed${x}.
}
```

B2indexed



Bu2x



The two place buffer that, in each place can only hold the number 1.

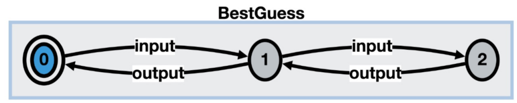
What I do not understand is how `B2indexed` dose not have event `a.output[1]` but `Bu2x = B2indexed${x}.` dose?

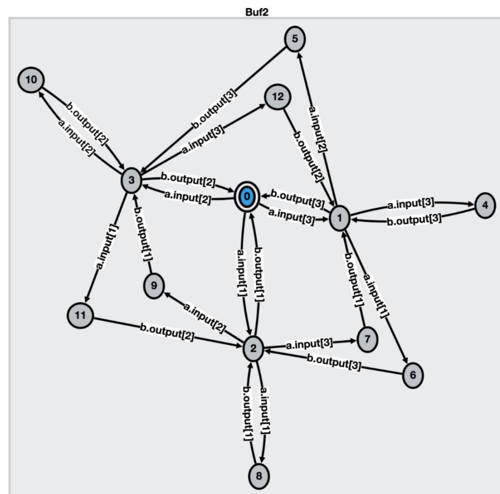
A more complex example is:

```
const N = 3
automata {
Buf1data = input[i:1..N] -> F[i],
   F[d:1..N] = output[d] -> Buf1data.

Buf2a = (a:Buf1data)/{b.move[x:1..N]/a.output[x]
Buf2b = (b:Buf1data)/{b.move[x:1..N]/b.input[x]}

Buf2data* = Buf2a||Buf2b.
Buf2 = simp(abs(Buf2data\{b.move[x:1..N]})).
//B2 = Buf2${x}.
B2indexed = (Buf2a||Buf2b)\{b.move[q:1..N]}.
Temp = B2indexed /{input/a.input[x:1..N],
                   output/b.output[x:1..N]}.
BestGuess = simp(abs(Temp)).
//Bu2x = B2indexed${x}.
}
operation {  Bu2x ~ BestGuess.}
```





The two place buffer that, in each place can hold a number from $\{1, 2, 3\}$ is defined and displayed. It is not as easy to see that it is a buffer as I wild like and if we could *freeze* the data in each place we might have simpler visual representation.

This will require some work to achieve. One tracking indexes so that we know what indexes exist in a process. For example what indexes are in `Buf2` defined above? Hence what should we write

`B2 = Buf2${x}`

17.2 Symbolic 2 Atomic mapping S2A

Symbolic execution is the obvious way to map `S2A` Symbolic Processes to Atomic Processes.

$$\mathsf{S2A}(\mathsf{A}\$\{\mathsf{x}\}) \sim \mathsf{A}$$

State contains an evaluation of the variables.
All you need to do is build a new node for each distinct evaluation reached.

> Add the set of initial evaluations to a "to do list" and repeatedly:
>
> *Remove the top node from the to do list and evaluate the boolean guard of each symbolic event and those that evaluate to true you apply the assignment to compute the new reachable state. If the new state already exists then add the transition ending at the corresponding node else add the state. When all events of the selected node have been processed either select the next node on the to do list else if the list is empty terminate.*

Reasoning about symbolic systems is problematic as they are infinite state and consequently frequently require a degree of theorem proving. A vast amount of work has gone into both push button theorem proving and interactive theorem proving over the recent years. Yet push button theorem needs to be a lot stronger and interactive theorem proving a lot easier.

To code with data structures, such as lists and records, needs headless Isabelle. Initially we will only have Z3 and are restricted to integers. For data structures we need rules for rewriting, simplifying, the data structures. These rules constitute a *Theory* of the data structure and as they are heavily reused are defined and stored on file. Thus a procedure should import them not define them.

17.3 Symbolic parallel composition $\_ \parallel_s \_$

The operations defined on atomic processes can be lifted to operations on symbolic processes. We first define `S2A` that maps symbolic process to atomic processes and then lift the atomic operation $\mathsf{Op_a}$ to the symbolic operation $\mathsf{Op_s}$ so that:

$$\mathsf{S2A}(\mathsf{Op_s}(\mathsf{A},\mathsf{B})) \sim \mathsf{Op_a}(\mathsf{S2A}(\mathsf{A}),\mathsf{S2A}(\mathsf{B}))$$

Symbolic parallel composition $\_ \parallel_s \_$ is an extension of atomic parallel composition $\_ \parallel \_$ where:

1. the nodes have a union of the indexes, suitably renamed to prevent name clashing
2. synchronising transitions have guards the conjunction of the component guards and assignments the union of the component assignments.

17.4 Symbolic abstraction

Let a transition $\mathtt{t1} \triangleq (\mathtt{n1},\mathtt{g1},\mathtt{ev1},\mathtt{a1},\mathtt{m1})$ be represented as $\mathtt{n1} \xrightarrow{\mathsf{g1,ev1,a1}} \mathtt{m1}$. We may refer to $\mathtt{n1}$ as $\mathtt{t1}_{\mathsf{pre}}$, to $\mathtt{m1}$ as $\mathtt{t1}_{\mathsf{post}}$ and refer to $\mathtt{e1}$ as $\mathtt{t1}_{\mathsf{en}}$.

We need to compute $\mathtt{t1} : \mathtt{t2}$ the transition representing the execution of $\mathtt{t1}$ followed by $\mathtt{t2}$. The execution of two transitions one after the other only occurs if the port node of the first transition is the pre-node of the second transition.

$$t_1 : \tau : \frac{\mathsf{n} \xrightarrow{\mathsf{g1,ev1,a1}} \mathsf{m} \quad \mathsf{m} \xrightarrow{\mathsf{g2},\tau\ \mathsf{,a2}} \mathsf{p}}{\mathsf{n} \xRightarrow{\mathsf{g1} \wedge \mathsf{g_2@a_1},\ \mathsf{ev1},\ \mathsf{a1@a_2}} \mathsf{p}} \qquad \tau : t_2 : \frac{\mathsf{n} \xrightarrow{\mathsf{g1},\tau\mathsf{,a1}} \mathsf{m} \quad \mathsf{m} \xrightarrow{\mathsf{g2,ev2\ ,a2}} \mathsf{p}}{\mathsf{n} \xRightarrow{\mathsf{g1} \wedge \mathsf{g_2@a_1},\ \mathsf{ev2},\ \mathsf{a1@a_2}} \mathsf{p}}$$

The guard $\mathsf{g2}$ is the post condition to transition $\mathsf{t_1}$ hence to compute the weakest precondition we need to apply Hoare Logic thus compute $\mathsf{g_2@a_1}$ which we add as a conjunction to the guard of $\mathtt{t1}$. Thus if $\mathtt{t1}_{\mathsf{pre}} = \mathtt{t2}_{\mathsf{post}}$ we can add the transition :

$$t1 : t2 = (\mathtt{t1}_{\mathsf{pre}}, \mathtt{t1}_{\mathsf{g}} \wedge \mathtt{t2}_{\mathsf{g}}@\mathtt{t1}_{\mathsf{a}}, \mathtt{t1}_{\mathsf{en}} : \mathtt{t2}_{\mathsf{en}}, \mathtt{t2}_{\mathsf{a}}@\mathtt{t1}_{\mathsf{a}}, \mathtt{t2}_{\mathsf{post}})$$

but if $\mathtt{t1}_{\mathsf{g}} \wedge \mathtt{t2}_{\mathsf{g}}@\mathtt{t1}_{\mathsf{a}} \neq \mathsf{False}$ then this transition can never be executed and hence can be removed with out effecting the behaviour of the process.

We can construct tests from:

$$\mathsf{S2A}(\mathsf{abs}(\mathsf{P})) \sim \mathsf{abs}(\mathsf{S2A}(\mathsf{P}))$$

17.5 Symbolic simplification

We certainly want property:

$$\mathsf{S2A}(\mathsf{simp}(\mathsf{P})) \sim \mathsf{simp}(\mathsf{S2A}(\mathsf{P}))$$

this could certainly be achieved if symbolic bisimulation simply matched transitions $t1$ and $t2$ by

$$t1_b \Leftrightarrow t2_b \wedge t1_{en} = t2_{en} \wedge t1_a \Leftrightarrow t2_a.$$

It might be possible to further simplify the symbolic automata while preserving the stated property.

18 Symbolic Petri Nets 2b added 2017

The symbolic Petri Nets are an *orthogonal* combination of the symbolic extension to automata and the construction of Petri Nets rather than automata. Thus we have a square with finite automata, finite Petri Nets, symbolic automata and symbolic Petri Nets on the corners. Mappings between adjacent edges are functions that should preserve the semantics of the processes, be monotonic with respect to refinement and congruent with respect to the operators.

$$\begin{tikzcd}
\mathsf{Petri}(\mathsf{Symb}(\mathcal{A})) \arrow[r, "\mathsf{TokenRule}"] \arrow[d, "\mathsf{S2A}"] & \mathsf{Symb}(\mathcal{A}) \arrow[d, "\mathsf{S2A}"] \\
\mathsf{Petri}(\mathcal{A}) \arrow[r, "\mathsf{TokenRule}"] & \mathsf{A}
\end{tikzcd}$$

$$\mathsf{S2A}(\mathsf{Petri}(\mathcal{A}\$\{\mathsf{x}\})) \sim \mathsf{Petri}(\mathcal{A})$$

$$\mathsf{TokenRule}(\mathsf{Petri}(\mathcal{A}\$\{\mathsf{x}\})) \sim \mathcal{A}\$\{\mathsf{x}\}$$

Index freezing prevents the building of finite state approximations of the index and instead builds a symbolic process. The construction $\mathsf{Symb}(\_)$ freezes all indexes but in the above square could be replaces by the more general index freezing $\_\$\{\mathsf{x}, ..\}$ of any set of indexes.

We have seen the partial expansion of both Petri Nets and of symbolic automata an clearly this can be generalised to a partial expansion of Symbolic Petri Nets.

18.1 Indexed Petri Nets

Indexed, specifications can be turned into FSPN in two quite separate ways. For a sequential indexed process $\mathcal{P}_{\mathcal{I}}$:

1. expand the index $\mathcal{I}$ converting $\mathcal{P}_{\mathcal{I}}$ into a sequential FSPN
2. turn the index $\mathcal{I}$ into a FSPN and the sequential process $\mathcal{P}$ into another then build, $\mathcal{P} \parallel_n \mathcal{I}$, net parallel composition of the process and the index.

Both options can be applied to $\mathsf{PTokenRule}$ the partial expansion of a net into an automata. All options should result in interleaving equivalent Petri Nets.