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. 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.

1 A practical approach to formalisation

Although it is interesting to have a sound and complete inference system to reason about programs it should be borne in mind that there is no one universally agreed upon notion of equality between interacting systems and hence putting lot of effort into constructing complete inference system may be of little practical use. Further both B and Event-B, by there use of forward simulation and not backward simulation demonstrate that incomplete inference systems can be of great practical use.

An approach, taken from control theory but 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

The approach taken in Event-B 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.

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

Private public To build understandable models they need to be small. We show how to *abstract* the private component while preserving the public behaviour.

Finite state approximations for pragmatic reasons we favour fast push button analysis over techniques that require a high degree of human time and expertise.

Visual verification Specifying properties of interacting systems is hard and in practice engineers are reluctant to spend a lot of time learning languages that may not be helpful.

Operational semantics provides a simple mapping between state based and event based systems.

Testing semantics provides confidence in the situations for which out definitions of equality and refinement are safe to use.

Robust semantics People frequently think in different ways about the same system depending upon the task at hand. Formalising our semantics in a variety of styles helps do this with some confidence.

Extensions via theory morphisms Each of our formalisms admit a logical interpretation where refinement is implication. The construction of Galois connections between these formalisms offers us a great deal of flexibility.

1. selecting different formalisms at different point in the development
2. mixing different formalisms
3. using the relational semantics we have silent out side of frame

.

Look to practice When things get tough to formalise look to your engineering intuitions. Refactor you theory rather than bolt on a patch.

2 State based interaction

Concurrent programs that interact by sharing variables have been formalised in many distinct ways. Here guarded actions similarly to both TLA+ and Event B. This has the benefit of making the connection event based interaction, as formalised by process algebras, easier to formalise.

Our state based programs will be called **modules** and contain a set of named, *guarded* **actions**. The module contains a set of variables that each action may reference.

3 Event based interaction

We model two styles of event based interaction, the first is based on the hand shake synchronisation of process algebras and the second is based on the broadcast synchronisation of IOA automata. The syntax used to index processes is taken from LTSA.

4 Defining Basic Processes

We are interested in defining processes that have private state and only interact with other processes via **events**. Process calculi CSP, CCS, ACP and Lotos are successful examples of such event based formalisms. These event based formalisms make use of a wide range of denotational semantics and hence a wide range of process equivalences. Each is given an axiomatic semantics to enable reasoning about the processes. We take a non axiomatic approach that loosely follows the notion of *Visual Verification* from Valmieies .. [].

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.

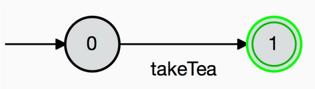
Sequential processes are defined using a simple process algebra $\Sigma = \{Act, - >, |, STOP\}$ the operational semantics of the defined process definition will be rendered as an automata by enclosing the definition in `automata { ... }` as shown in the examples below. We will frequently omit the `automata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

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

4.0.1 Event prefixing

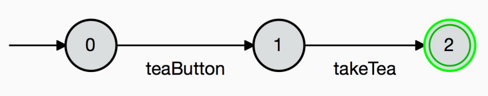
A simple process that performs a single event and stops can be built by prefixing an event `takeTea` to the `STOP` process using the `->` operator by the command:

```
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:

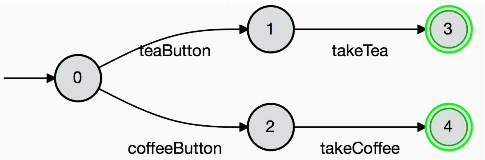
```
automata {
  Simple* = (takeTea->STOP).
  Two = (teaButton->Simple).
}
```

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.

4.0.2 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).
```

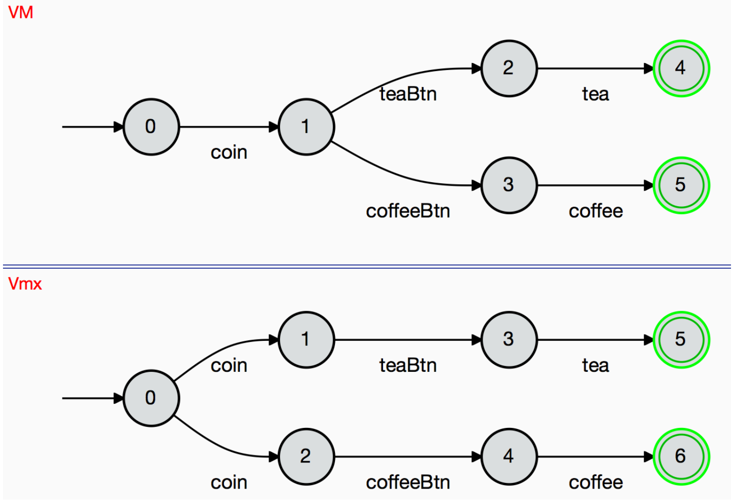


this automata *branches* at the initial node.

4.1 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.

4.2 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:

$$X|Y \sim Y|X$$

similarly we have:

$$X|(Y|Z) \sim (X|Y)|Z$$

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

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

$$X|X \sim X$$

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.

4.3 Non terminating processes

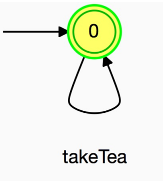
We call the set of know processes the *Process name space*. Initially the *Process name space* is `{STOP}`. Each process definition `P1 = ...` adds the the defined process `P1` to the name space.

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.

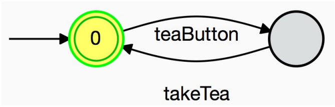
Clearly any state `S` in process `P1` could also be *conceptual identified* with the the process consisting of the same set of states and transitions but with initial state `S`. We use this idea to define non terminating processes. By allowing any valid process to be used where `{STOP}` has been used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

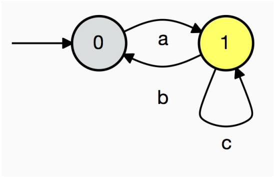


```
BT = (teaButton->takeTea->BT).
```



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*.

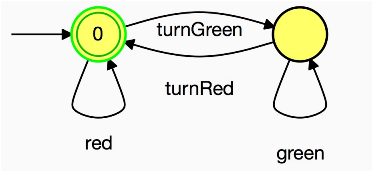
4.4 Translating any finite state automata into a process term

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.

5 Semantics of processes with atomic events

Distinct automata can be used to represent exactly the same process. We formalise this be defining a semantic equivalence 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 to very different notions of process equality that are widely used and widely discussed in the literature.

5.1 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:

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

Infinite traces do not end:

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

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 4.1 are identified.

5.2 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 4.1 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.

$$Col_{i+1}(n) = Col_{i+1}(m) \Leftrightarrow Col_i(n) = Col_i(m)$$

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. . Using this colouring algorithm it is easy to see the result of applying bisimulation simplification to simple automata.

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. 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.

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

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

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

6 Event hiding and process simplification

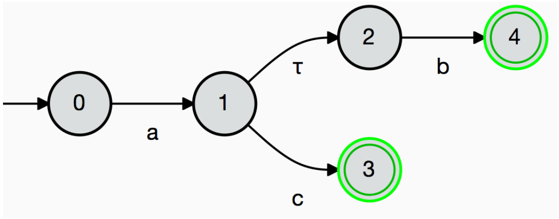
Our process language can be used both to specify *implementations* of processes and more abstract *specifications*. Our tool is designed to support the approach popularised by Event B. Both the abstract specification and the more concrete implementation are specified in the same language. The tool then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

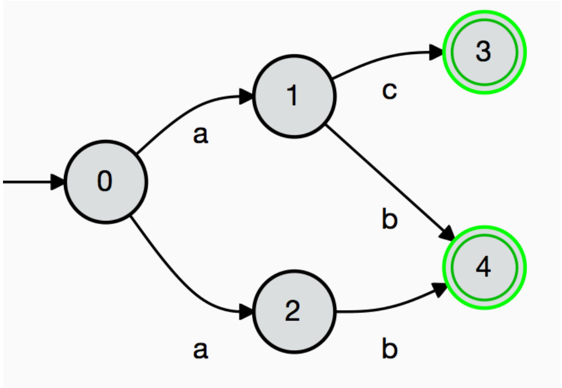
We can make events private by hiding them so they can not be seen. `_\{t}` operator renames the `t` event to the unobservable `tau` event.

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 $x\xRightarrow{\tau}y$.

A sequence of zero or more τ events

$$x\xRightarrow{\tau}y \triangleq \exists i \geq 0 \circ \exists n_1, n_2, ..n_i \circ x\xrightarrow{\tau}n_1, n_1\xrightarrow{\tau}n_2 \ldots n_i\xrightarrow{\tau}y$$

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

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

$$x\xRightarrow{\mathsf{a}}y \triangleq \exists u, v \circ x\xRightarrow{\tau}u \wedge u\xrightarrow{\mathsf{a}}v \wedge v\xRightarrow{\tau}y$$

6.1 Event hiding and non terminating processes bug to be fixed

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 $\mathsf{P} \parallel \mathsf{Q}$ 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*.

7 Concurrent Processes

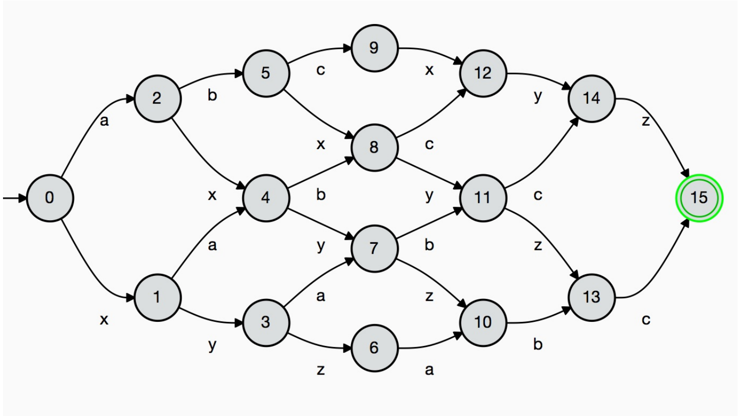
So far we have have defined sequential processes but now we wish to define how two sequential processes behave when they are both run together. This is modelled using the parallel composition operator _ || _. Two processes run in parallel can only interact via event synchronisation How this is defined 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.

7.1 Handshake synchronisation

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

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

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

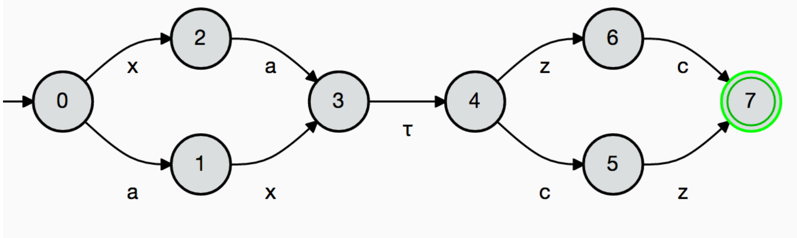


This definition of parallel composition equates parallel processes with a single sequential process. As such some aspects of real parallel processes are lost. Some times this equality is referred to as being based upon the *interleaving assumption*.

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.

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.

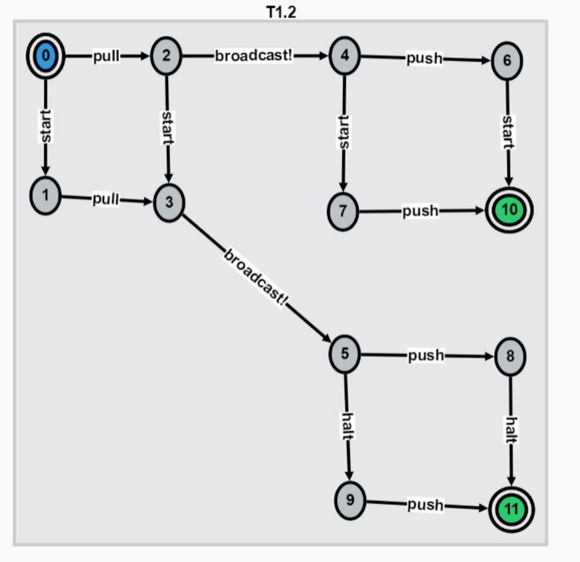
Hence the only way the control the order of two events from different concurrent processes is to introduce a synchronizing event. In above the `a` event and the `z` event are from different concurrent processes in the interleaving example either could occur first. Whereas in the synchronization of the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

7.2 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 this see broadcast! above these 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. We model such events by defining event pairs that model the unblocking *send* events and separately *receive* events.

Non-blocking send events can further 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.

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

Note: if you are not listening you might miss the warning and if you are not looking you might not see the green light.

Two broadcast! events or two braodcast? events will synchronise just like normal hand shake events.

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. If we could introduce fake Traffic Lights then it would be hard to guarantee any properties of the system.

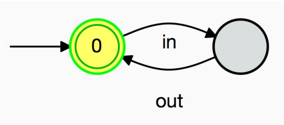
We need to understand how events interact in our formal model so that we can accurately capture the world w are modelling.

8 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.

8.1 Labelling Processes

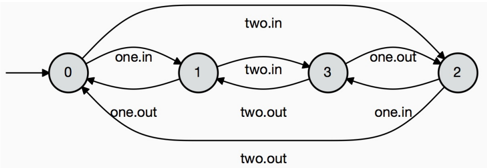
In the following example we make use of a one place buffer `Buf` is a process that when empty can receive some thing `in` and when full can return it `out`.



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).
```



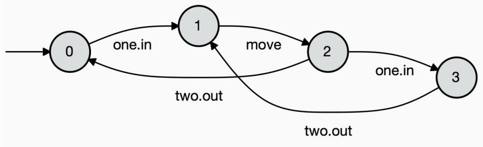
8.2 Event renaming

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}).
```

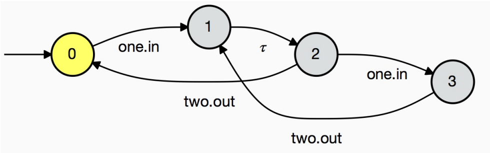


Note that the result is much simpler than the interleaving as the `move` event now can only occur when **both** buffers are able to perform it.

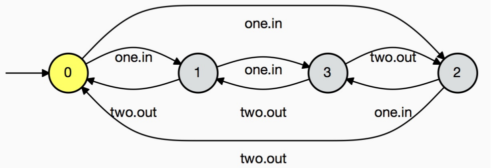
8.3 Process simplification

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}.
```

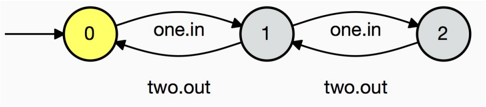


```
B5 = abs(B3\{move}).
```



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.

9 Testing and equality

In this section we are going to introduce a very 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. □

In what follows we will define 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.

10 Indexed Process definitions

Basic process definitions you have seen so far a fixed finite set of states. This accurately reflects many situations very well and allows easy and complete push button verification. Alternatively when what you are modelling has infinite state 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.

10.1 The small world assumption

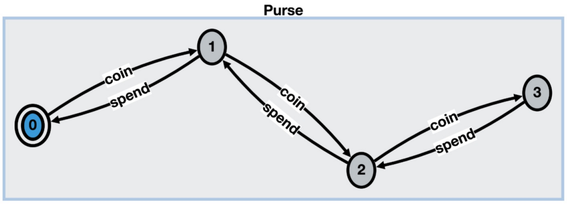
Most program bugs can be found while restricting variables to range over a small domain. Using this assumption we model processes with variables by indexing the processes and restricting the indexes to range over a small domain. Having done this the variables in the state can be removed by instantiating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

10.2 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”.

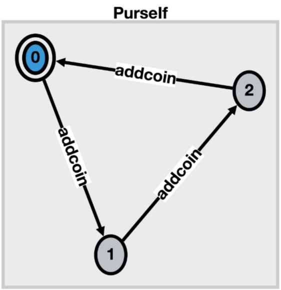
On the right hand side of the equality we define guarded events:

```
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 `if...then...else` command is very like the choice between two `wait` commands with an exception. Notice how the else part of the if command is not prefixed with an event and it parses and produces the automata displayed.

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



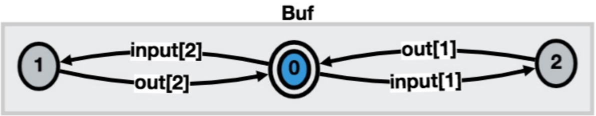
Where as if we use a process that is not event prefixed in a `wait` command, see example below, then it fails to parse.

```
PurseW = P[0],
  P[c:0..N] =
   (when (c < N)   addcoin-> P[c+1] |
   when (c >= N)   P[0] ).
```

10.3 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.

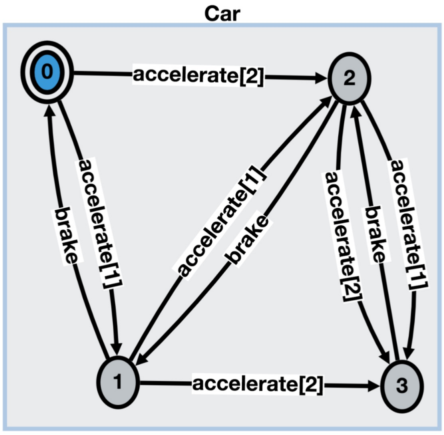
10.4 Cars Example

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.

```
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]).
```



10.5 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.

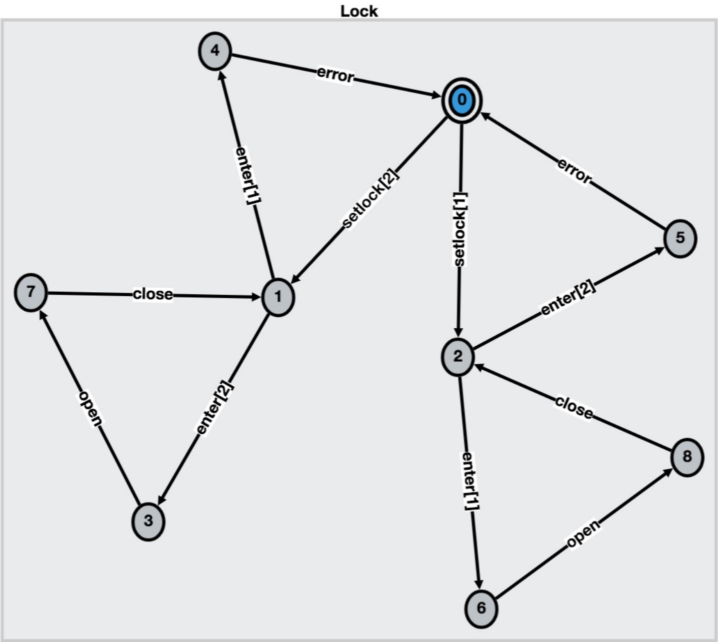
> *Closed doors are always locked. The door starts closed. The lock can hold any of a number of codes. To open you need to input the correct code and after opening the door can only close. Inputting the wrong code is an error and the door returns its start state. Before using the door the code must be 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)).
```

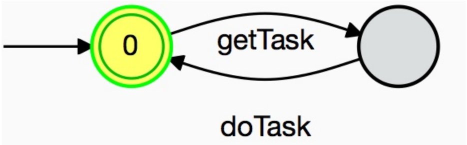
Step 5 Note the value input in the `setlock[i:1..N]` event is stored in the state of the process `L[i]` for subsequent comparison with the value input in the `enter[i:1..N]` event.

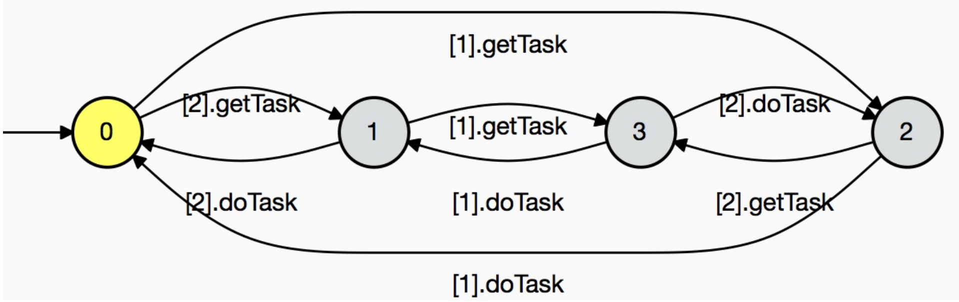
10.6 Indexing concurrent processes.

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)).
```

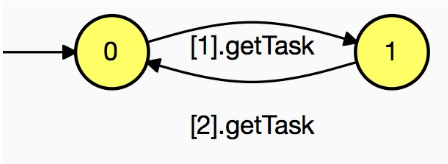




We can add a `Farmer` process to hand out the `Tasks` to the `Workers` in order. Then build a `Farm` composed of the `Farmer` and the `Workers`.

```
  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.

11 Petri Nets 2b added 2017

Up until now we have not distinguished between a process specification $\mathcal{A}$ written in our language and its semantics displayed as an automata A. Now that we define a Petri Net semantics for the same language will define a function Petri that constructs a Petri Net with event labelled transitions from a process specification $\mathcal{P}$. We will writte $\mathsf{Petri}(\mathcal{P})$.

11.1 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*, $\bullet$`t` to the Box of transition `t` and from transition `t` Box to *post place* `t`$\bullet$.

Finite state Petri Nets like finite state automata can be approximation of potential infinite sate processes.

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 now we want to build Petri Nets from specification $\mathsf{Petri}(\mathcal{P})$. 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 automata node is represented by a marking. That is a pair of places, one element of the pair taken from each component process. Thus each place represents the state of one of the component processes. Each process can be give a unique location and then each node annotated with the location of its process. The start node corresponds to the the pair of initially marked places.

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.

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

11.2 Token Rule

We define `TokenRule` that maps Petri Nets to atomic processes. These are the same automata that would have been constructed had we built the automata directly from our process language, hence we have:

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

Let us write A and B for Petri Nets. We have lifted operations Op_a on automata to operations Op_n on Petri Nets so that they obey the following algebraic rules:

$$\mathsf{TokenRule}(\mathsf{A} \parallel_{\mathsf{n}} \mathsf{B}) \sim (\mathsf{TokenRule}(\mathsf{A}) \parallel \mathsf{TokenRule}(\mathsf{B}))$$

$$\mathsf{TokenRule}(\mathsf{A}\$\{\mathsf{x}\}) \sim (\mathsf{TokenRule}(\mathsf{A}))\$\{\mathsf{x}\} \qquad \mathsf{TokenRule}(\mathsf{abs}(\mathsf{A})) \sim \mathsf{abs}_{\mathsf{n}}(\mathsf{TokenRule}(\mathsf{A}))$$

The `TokenRule` builds an automata from a Petri Net by executing transitions of the Petri Net. A Petri Net transition can only be executed when all its pre places are marked. When the transition is executed the tokens on its pre places are removed and tokens are placed on the transitions post places.

Above explains how an atomic specification can be used to build a Finite State Petri Net, FSPN. The construction of FSPN from indexed specifications proceeds by constructing the FSPN for each sequential process analogously to the construction of finite state automata.

We can compute, PTokenRule, a partial Token Rule that takes a set of process names that must be converted into an automaton prior to applying $\_ \parallel_n \_$ to build a Petri Net from the newly built sequential Net and the remaining 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 and be applied to PTokenRule the partial expansion of a net into an automata. All options should result in interleaving equivalent Petri Nets.

Can we define Petri Net bisimulation so that:

$$\mathsf{Petri}(\mathcal{A} \parallel \mathcal{B}) \sim_{\mathsf{n}} (\mathsf{Petri}(\mathcal{A})) \parallel_{\mathsf{n}} \mathsf{Petri}(\mathcal{B}))$$

$$\mathsf{Petri}(\mathcal{A}\$\{\mathsf{x}\}) \sim_{\mathsf{n}} (\mathsf{Petri}(\mathcal{A}))\$\{\mathsf{x}\} \qquad \mathsf{Petri}(\mathsf{abs}(\mathcal{A})) \sim_{\mathsf{n}} \mathsf{abs}_{\mathsf{n}}(\mathsf{Petri}(\mathcal{A}))$$

12 Symbolic Processes mostly 2b added 2017

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.

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

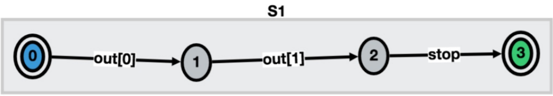
In our notation $\{x := F(x, y)\}@\{y := G(x, y)\} \triangleq \{x := F(x, y), y := G(F(x, y), y)\}$

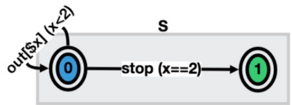
13 Index freezing (z3 - headless Isabelle)

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}.
```

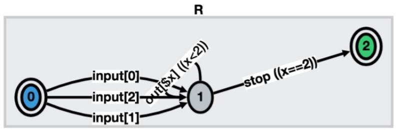


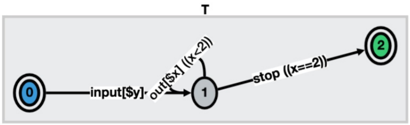


1. assignment `x:=x+1` is missing
2. presentation order *guard name assignment*
3. 2 should be `N` and `$x` should be `x`

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}.
```

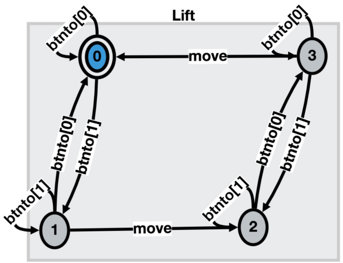




1. `S = R1${y}` currently fails to build.

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}).
```

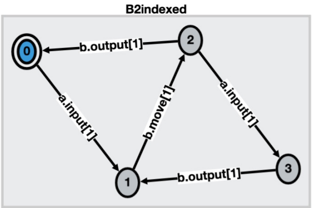
The result of expanding all variables should be the same as building the atomic automata from scratch, hence R∼T in above. (Note building from the automata/Petri Net allows interacting with the diagram)

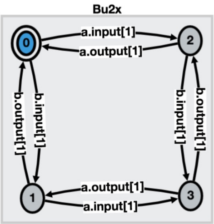
13.1 Need for index freezing

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}.
}
```





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

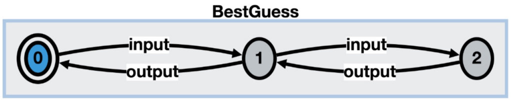
What I do not understand is how `B2indexed` dose not have event `a.output[1]` but `Bu2x = B2indexed${x}.` dose?

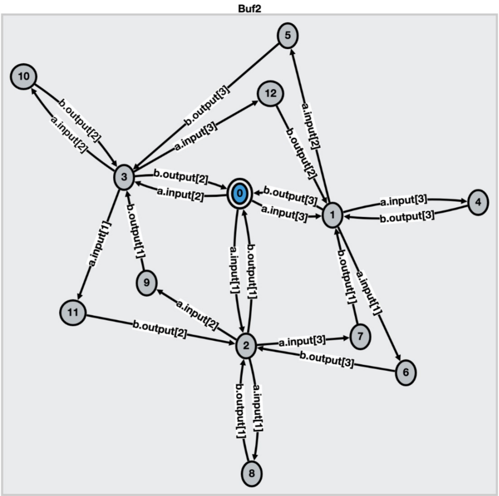
A more complex example is:

```
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}`

13.2 Symbolic 2 Atomic mapping S2A

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

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

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.

13.3 Symbolic parallel composition $\_ \|_s \_$

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:

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

Symbolic parallel composition $\_ \|_s \_$ is an extension of atomic parallel composition $\_ \| \_$ 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.

13.4 Symbolic abstraction

Let a transition $\mathsf{t1} \triangleq (\mathsf{n1}, \mathsf{g1}, \mathsf{ev1}, \mathsf{a1}, \mathsf{m1})$ be represented as $\mathsf{n1} \xrightarrow{\mathsf{g1,ev1,a1}} \mathsf{m1}$. We may refer to $\mathsf{n1}$ as $\mathsf{t1_{pre}}$, to $\mathsf{m1}$ as $\mathsf{t1_{post}}$ and refer to $\mathsf{e1}$ as $\mathsf{t1_{en}}$.

We need to compute $\mathsf{t1}\circ\mathsf{t2}$ the transition representing the execution of $\mathsf{t1}$ followed by $\mathsf{t2}$. 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.

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

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 $\mathsf{t1}$. Thus if $\mathsf{t1_{pre}} = \mathsf{t2_{post}}$ we can add the transition :

$$t1 \circ t2 = (\mathsf{t1_{pre}}, \mathsf{t1_g} \wedge \mathsf{t2_g@t1_a}, \mathsf{t1_{en}} : \mathsf{t2_{en}}, \mathsf{t2_a@t1_a}, \mathsf{t2_{post}})$$

but if $\mathsf{t1_g} \wedge \mathsf{t2_g}@\mathsf{t1_a} \neq \mathsf{False}$ 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:

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

13.5 Symbolic simplification

We certainly want property:

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

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.

14 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.

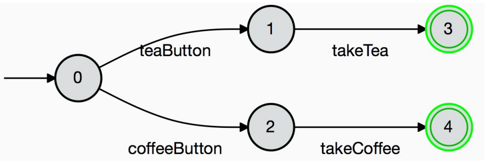
We have seen the partial expansion of both Petri Nets and of symbolic automata an clearly this can be generalised to a partial expansion of Symbolic Petri Nets.

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 Semantic equivalence of processes

One of the key questions is: what automata should be equated and how do you justify you notion of equality? Before we try to answer this we need to remember what equality means.

16.1 Classic notions of equality

A 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.

16.2 Complete Trace equality

See Section 5.1 for details.

This can be computed by the well documented *Subset Construction* that computes a Deterministic automata from a non deterministic automata. and then applying the bisimulation algorithm discussed in Section 5.2

16.3 Failure equality

The Failure semantics of a process consists of a set of Failures. A Failure is a pair consisting of a trace and a Refusal set (tr, R).

$$Fail(\mathsf{P}) = \{(\rho, R) \circ S_{\mathsf{P}} \xrightarrow{\rho} n \wedge R \subseteq \overline{\pi(n)}\}$$

If you execute a process in parallel with a context then the context can:

1. *see* the trace of execution and
2. when the execution stops it can infer that the process refuses to perform any of the events that the context is offering.

Processes are *Failure* equivalent if the have the same set of Failures.

$$\mathsf{P} =_F \mathsf{Q} = Fail(\mathsf{P}) = Fail(\mathsf{Q})$$

Failure refinement, $\sqsubseteq_F$

$$\mathsf{P} \sqsubseteq_F \mathsf{Q} = Fail(\mathsf{P}) \supseteq Fail(\mathsf{Q})$$

One of the most important aspects of Failure semantics is that failure refinement is the same as testing refinement using parallel composition for handshake events, $\sqsubseteq_{Ths}$.

$$\mathsf{P} \sqsubseteq_F \mathsf{Q} = \mathsf{P} \sqsubseteq_{Ths} \mathsf{Q}$$

16.4 Bisimulation

See Section 5.2 for details.

16.5 Some practical considerations

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.

17 TLA+ Overview

TLA+ is used to model shared memory concurrency. 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 $State : Var \to Val$. That is the state is a context in which expressions can be evaluated. 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.

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.

We can define the behaviour of the Module using a Next state predicate and an Init predicate

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

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

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

The receiving action makes no explicit reference to the value `d` because TLA+ uses shared memory all that need happen is that

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

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 Modules

Assume module `N` declares a predefined module

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

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:

$$\texttt{get} \triangleq \texttt{InChan!input}(\ldots) \land \ldots$$

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.

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+.

18 Syntax

Summary of event syntax:

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

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

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

```
automata  \{   ...   \}
```

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

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

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

Choice	`A = a->P|b->Q`	`Farmer = ([i:0..N].task ->W[i]),`
		`    W[i:0..N] = ([i].end->Farmer).`

Labeling	`lab:P`	see below
Parallel	`A = (P||Q)`	`Workers = (forall [i:0..N] ([i]:Worker)).`
Relabeling	`P/{new/old}`	`P/{new[i:0..N]/old[i]}`
Hiding	`P\{act}`	`P\{act[i:0..N]}`

For processing automata:

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

For processing automata use the following operations within:

```
operation  \{  ...  \}
```

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

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.

19 Tool Development

Currently toll builds finite state automata. Needs options to build symbolic automata, finite state Petri Nets and symbolic Petri Nets.

19.1 Quality Code

The event based tool has been used for teaching but greater support in terms of functionality and debugging needed. Support for improved code quality and ease of extension needed.

1. *Algorithm design + documentation + Review*
2. *Code review prior to committing*
3. *Public Interfaces Java Docs + documentation*

A process specification $\mathcal{P}$ is input as text and automata P is built. The specification contains various operations $\_ \parallel \_$, $\mathsf{abs}(\_)$, $\mathsf{simp}(\_)$, $\_ \backslash \{\mathsf{evt}\}$, $\_/\{\mathsf{new/old}\}$ ... Are these defined as automata to automata mappings or as specification to automata mappings?

19.2 Technical overview

Ongoing review and discussion!

1. Process variables will help speed up test construction.
2. Asynchronous architecture with processing in a docker containers
3. Build hierarchy of processes where the leaves are defined without reference to other processes. and where child processes are directly used in the construction of the parent. Needed for flexible construction - of symbolic automata and Net options
4. Compile each process separately and only recompile if that part of the document has changed. **Could use JEdit in place of web?**
5. 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.

TODO

1. ***** Design Considerations *****
   (a) Interface for Automata made public and documented.
   (b) Support for plugin style development.
   (c) efficient immutable state so that processing can be performed asynchronously!
   (d) docker containers!
2. ****start** Bug fixes:**
   (a) abstraction
   (b) broadcast

(c) Add process location and local divergent state. Then use in unfair abstraction.

(d) change the `wait` command to be more like the `if...then` command see previously

(e) Index hiding $P\$\{x\}$ fix bugs for *leaf* processes see above

3. ****start**** Build a Process Hierarchy (probabily from the abstract syntax tree) where the leaves are defined without reference to other processes. and where child processes are directly used in the construction of the parent. Branch at parallel composition hiding and renaming. Other operators may be added later!

4. ****start** Process variables `X$`** There are many algebraic equalities between processes, such as `X$+Y$ ~ Y$+X$` that are true for all processes. Hence define a set of `n` processes in a directory and use these to substitute the variables `X$` and `Y$`, thus generating n^2 tests.

   Just writing X equalities we will have Xn^2 tests and as many of our extensions have many algebraic properties the saving in time setting up tests could be very considerable.

5. ****start** Add Petri Nets**,

   (a) compile to Petri Net Petri (and location) The tool takes as input process specifications and by default builds finite state automata as approximations. The function Petri can be applied to a process specification where it must build Nets for all processes below it in the Process Hierarchy. Define $\_ \parallel_n \_$ on Petri Nets initially only for hand shake synchronisation.

   (b) Implement Token Rule to build automata TokenRule

   Test using equalities: $\forall \mathsf{P} \circ \mathsf{P} \sim \mathsf{TokenRule}(\mathsf{Petri}(\mathsf{P}))$

   $\forall \mathsf{P}, \mathsf{Q} \circ \mathsf{P} \parallel \mathsf{Q} \sim \mathsf{TokenRule}(\mathsf{Petri}(\mathsf{P}) \parallel_{\mathsf{n}} \mathsf{Petri}(\mathsf{Q}))$

   As a side effect this will speed up parallel composition.

6. **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 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: $\forall \mathsf{P} \circ \mathsf{P} \sim \mathsf{S2A}(\mathsf{P}\$\{\mathsf{x}\})$

   $\forall \mathsf{P}, \mathsf{Q} \circ \mathsf{P} \parallel \mathsf{Q} \sim \mathsf{S2A}(\mathsf{P}\$\{\mathsf{x}\} \parallel \mathsf{Q}\$\{\mathsf{y}\})$

7. **Fast abstraction option** for failure semantics

   (a) On automata

      i. for n so that $(\forall t \circ t \in pre(n) \to name(t) = \tau) \land (\forall r \bullet r \in post(n) \to name(r) = \tau)$

      ii. Replace $n, pre(n), post(n)$ with $\{x \xrightarrow{\tau} y \bullet x \in pre(pre(n)) \land y \in post(post(n))\}$

      iii. better done in Labelled Petri Nets

   (b) On Petri Net: For transition t let $post(t) = t\bullet$ and $pre(t) = \bullet t$.
   For set of places R let $pre(R) = \{t \circ t \bullet \cap R \neq \emptyset\}$ and let $post(R) = \{t \circ \bullet t \cap R \neq \emptyset\}$

      i. for transition r such that $name(r) = \tau$

      ii. if $(\forall t \bullet t \in pre(r\bullet) \to name(t) = \tau) \land (\forall s \bullet s \in post(\bullet t) \to name(s) = \tau)$

iii. Replace $r\bullet, pre(r\bullet), post(r\bullet)$ with $\{\bullet x \xrightarrow{\tau} y\bullet \circ x \in pre(r\bullet), \wedge y \in post(r\bullet)\}$

Tested using $abs(P) =_F fabs(P)$

8. **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?

9. **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

10. **Compute Failure semantics** needed for example below.

    (a) Apply subset construction to P to build a DFA, $nfa2dfa(\mathsf{P}) = \mathsf{D}$

    $$\text{as } \forall n \in N_{\mathsf{D}} \circ \exists! S_{\mathsf{D}} \xrightarrow{tr} n \text{ hence } \exists \mathsf{n2tr_D}(\_) \circ S_{\mathsf{D}} \xrightarrow{\mathsf{n2tr_D}(n)} n$$

    (b) Annotate each DFA node $n \in N_{\mathsf{D}}$ with Ready sets taken from nodes in P.

    $$\{\pi(v) \circ v \in N_{\mathsf{P}} \wedge S_{\mathsf{P}} \xrightarrow{\mathsf{n2tr_D}(n)} v\}$$

    i. DFAR normal form add Ready set but keep A and drop B if $A \subsetneq B$

    (c) Compute DFAR equality

11. 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.

12. **Gui to define event renaming and synchronisation - needs Petri Nets**

13. **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`

14. **Add Process invariants**, need to look at TLA+ syntax / parsing + import libraries

15. Support for SDN may be TLA+

19.3 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`: $Process \to Module$ Process mapped to Module with additional State variable. This will add the ability to check process satisfies temporal logic specification
2. `M2P`: $Module \to Process$ 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`: $Module \to Process$ 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

19.4 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 the ability to model check algebraic rules by generating ”all” process models of a fixed size and verifying the rules against these processes.**
4. **Add interrupts. see after ”\end{documnet}”**
5. **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.**
6. **Add event refinement**
7. **Add δ events - unobservable and blocked + model known**
8. **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

20 Bugs and examples

20.1 Bug 1 Abstraction

```
             /* Bug1  in abstraction */
automata {
Z = (a->l->Z|b->(r->x->STOP|y->STOP)).
Yl = abs(Z\{l}).
Ylr = abs(Yl\{r}).
Yr = abs(Z\{r}).
Yrl = abs(Yl\{r}).
Y = abs(Z\{r,l}).
}
operation {
Y~Yrl.
Y~Ylr.
Ylr~Yrl.
}
```

20.2 Bug 2 Broadcast communication

```
automata {
             /* Bug2  in broadcast comunication*/
S = x->S|a->STOP.
R = x->STOP.
Good = S||R.
SR = (a->STOP|x->a->STOP).

Sb = x!->Sb|a->STOP.
Rb = x?->STOP.
Bug = Sb||Rb.
SbRb = (a->STOP|x!->X),
     X = (x!->X|a->STOP).

Sbb = x?->Sbb|a->STOP.
Rbb = x!->STOP.
Bugb = Sbb||Rbb.
SbbRbb = (a->x!->STOP|x!->a->STOP).
}
operation {
Bug ~ SbRb.
Bugb ~ SbbRbb.
Good ~ SR.
}
```

20.3 Bug 3 restricted rendering

```
const N = 3
automata {
             /*Bug in not rendering automata */
Sendb* = input-> sout! -> Sendb.
Receiveb* = sout? -> output->Receiveb.
Go = Sendb||Receiveb.

Buf1data = input[i:1..N] -> F[i],
   F[d:1..N] = output[d] -> Buf1data.

           /* not rendering Buf2a makes the compile fail
 Performing layout for Buf2, remaining: 1
An exception was thrown that was not related to your script.
java.util.ConcurrentModificationException
at java.util.LinkedHashMap$LinkedHashIterator.nextNode(LinkedHashMap.java:711)
at java.util.LinkedHashMap$LinkedValueIterator.next(LinkedHashMap.java:739)
 */
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]})).

}
}
```

20.4 Need for failure semantics

```
automata {
/* Question is 2 place buffer with: two faulty  one place bufferes
   = 2 place buffer with: one faulty  and one good one place buffers?
*/
FA = (zero.in ->(zero.out->FA|drop->FA) | one.in ->(one.out->FA|drop->FA )).
FAs = simp(abs(FA\{drop})).
FB = (zero.in.[d:0..N] ->(zero.out.[d]->FB|drop->FB) |
      one.in.[e:0..N] ->(one.out.[e]->FB|drop->FB ))\{drop}.
FBs = simp(abs(FB)).

Next = b:FAs/{a.zero.out/b.zero.in,a.one.out/b.one.in}.
First = a:FAs.
Two = (First|| Next)\{a.zero.out,a.one.out}.
TwoS = simp(abs(Two)).

Buff = a.one.in ->a.one.out->Buff | a.zero.in->a.zero.out->Buff.
One =  (Buff|| Next)\{a.zero.out,a.one.out}.
OneS =  simp(abs(One)).
}
operation {
One !~Two.
One # Two.
/* these are not bisimular  but are trace equivalent
Let One->a.one.in -> X1 and  Two->a.one.in -> X2 where X1!=One and X2!=Two
```

```
   Now X1 can not forget the input "one" where as X2 can.
   But are they failure equivalent?
*/
}
```