Set Theory

- $\{x : X | P(x)\}$ is the set which predicate P is true
- $S \subseteq T$ (S is a subset of T)
- $\mathbb{P}X$, also known as the power set of X, is the set of all subsets of X, including the empty set.
- Union, intersection and difference operators apply in Z as well.

Cardinality

If X is a finite set, then the cardinality of X is the number of elements in the set. Notation : $\#X$.
Example $\#\mathbb{P}A = 2^{\#A}$, the cardinality of the powerset of A.

Cartesian Product

The set of all **ordered pairs**. Notation: $A \times B$.

Types

Z is strongly typed; every expression given a type.

$$(x, y) : A \times B$$
$$x : A;\ y : B$$
$$x, y : A \quad \text{(when } B = A\text{)}$$

Type information is membership. $x : A$ means that x is in A and it has the type of A. $\bullet$ refers to such that

$$\forall S : \mathbb{P}\, A \bullet \ldots \quad \text{not} \quad \forall S \subseteq A \bullet \ldots$$

$$\forall S : \mathbb{P}\, A \bullet (\forall y : S \bullet \ldots) \quad \text{not} \quad \forall S : \mathbb{P}\, A;\ y : S \bullet \ldots$$

Don't use subsets when using $\bullet$, use type information (for the first point). For the second point, S was defined before the scope, so it can be used as a type. Must be defined within same scope to be used as type.

Relations

A relation R from A to B is denoted by $R : A \leftrightarrow B$ is a subset of $A \times B$. This means that R is a subset of $A \times B$.
The following are logically equivalent and helps to better understand relations:

$$(c, z) \in R \quad \text{and} \quad c \mapsto z \in R \quad \text{and} \quad c\, \underline{R}\, z$$

Domain of R is the set $\{a : A | \exists b : B \bullet a \underline{R} b\}$. This refers to the set of all first element of the relation $R : A \leftrightarrow B$. Range of R is the set $\{b : B | \exists a : A \bullet a \underline{R} b\}$. This refers to the set of all the second element of the relation R.

$$\begin{array}{|l} \_ \leqslant \_ : \mathbb{N} \leftrightarrow \mathbb{N} \\ \hline \forall x, y : \mathbb{N} \bullet \\ \quad x \leqslant y \ \Leftrightarrow\ \exists k : \mathbb{N} \bullet x + k = y \end{array}$$

i.e. the relation $\leqslant$ is the infinite subset

$$\{(0,0), (0,1), (1,1), (0,2), (1,2), (2,2), \ldots\}$$

The first row represents the set of interest. The second row represents the data that we want to filter from the set. $\Leftrightarrow$ refers to logical equivalence of both left and right.

Domain and Range Restriction

Consider the relation $R : A \leftrightarrow B$ and $S \subseteq A$ and $T \subseteq B$, then

$$\begin{array}{lll} S \lhd R & \text{is the set} & \{(a, b) : R \mid a \in S\} \\ R \rhd T & \text{is the set} & \{(a, b) : R \mid b \in T\} \end{array}$$

$S \lhd R$ means that S is the domain of R. And since $S \subseteq A$, the value of a in R must exist in S. Similarly for $R \rhd T$ means that T is the range of R.
Consider an example $has\_sibling : People \leftrightarrow People$. The relation $has\_sibling$ looks all pairs of people. So if $female \lhd has\_sibling$, this means that the first element of the pair is a female and is the sister of second element, forming is_sister_of. And if $has\_sibling \rhd female$, this means that the range of the mapping of people would be female and the first element would have a sister.

Domain and Range Subtraction

Similar to above except now we want elements that are not in the domain/range of interest.

$$\begin{array}{lll} S \lhd\!\!\!\!- R & \text{is the set} & \{(a, b) : R \mid a \notin S\} \\ R \rhd\!\!\!\!- T & \text{is the set} & \{(a, b) : R \mid b \notin T\} \end{array}$$

The following predicates are true

$$S \lhd\!\!\!\!- R = (A - S) \lhd R$$
$$R \rhd\!\!\!\!- T = R \rhd (B - T)$$
$$S \lhd\!\!\!\!- R \in A \leftrightarrow B$$
$$R \rhd\!\!\!\!- T \in A \leftrightarrow B$$

Relational Images

Suppose $R : A \leftrightarrow B$ and $S \subseteq A$

$$R(\!| S |\!) = \{b : B \mid \exists a : S \bullet a \underline{R} b\}$$

$$R(\!| S |\!) \subseteq B$$

If R is *divides*, then $S = \{7, 8\}$, $R(|S|)$ would contain the set of numbers that are divisible by 7 or 8.$R(|S|)$ contains elements in the range.

Inverse

Reverse the ordering of the pairs and see whether it exists in the relation. Note that $A \times B \neq B \times A$.

Suppose $R : A \leftrightarrow B$

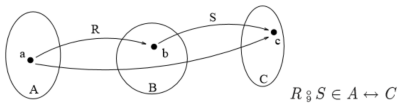
$$R^{-1} = \{(b, a) : B \times A \mid a \underline{R} b\}$$

$$R^{-1} \in B \leftrightarrow A$$

Relational Composition

Suppose $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$

$$R \fatsemi S = \{(a, c) : A \times C \mid \exists b : B \bullet a \underline{R} b \ \wedge\ b \underline{S} c\}$$



$R \fatsemi S \in A \leftrightarrow C$

Relational composition refers to the arrow pointing from A to C in the above diagram.

Partial Function

(a)

$$\begin{array}{|l} double : \mathbb{N} \to \mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet double(n) = 2n \end{array}$$

(b)

$$\begin{array}{|l} halve : \mathbb{N} \nrightarrow \mathbb{N} \\ \hline \mathrm{dom}\, halve = \{n : \mathbb{N} \mid \exists m : \mathbb{N} \bullet 2m = n\} \\ \forall n : \mathrm{dom}\, halve \bullet 2 * halve(n) = n \end{array}$$

Partial functions map 1-to-1. Everything in domain has to map to something in range.

Function Overriding

Function Overriding

Suppose $f, g : A \nrightarrow B$; then

$$f \oplus g \quad \text{is the function} \quad (\mathrm{dom}\, g \lhd\!\!\!\!- f) \cup g$$

i.e. the following predicates are true

$$\mathrm{dom}\, f \oplus g = \mathrm{dom}\, f \cup \mathrm{dom}\, g$$
$$\forall a : \mathrm{dom}\, g \bullet (f \oplus g)(a) = g(a)$$
$$\forall a : \mathrm{dom}\, f - \mathrm{dom}\, g \bullet (f \oplus g)(a) = f(a)$$
$$f \oplus g \in A \nrightarrow B$$

$$\{a \mapsto x, b \mapsto y, c \mapsto x\} \oplus \{a \mapsto y\} \ =\ \{a \mapsto y, b \mapsto y, c \mapsto x\}$$

The example captures everything that function overriding does.

Sequences

Non-empty sets that maps natural numbers (starting from 1) to the respective set elements. In other words, its a list that starts with index 1. 2 sequences are the same if the element at every index is the same. Set operations on sequence may not result in a sequence. The reason being that the index mappings no longer start at 1 and run continuously.

Concatenation

Not the same as union! Similar to python's list extend method; adds to the back of the first sequence.

Head and Tail

Head refers to the first element of the sequence. Tails refers to the entire sequence less the Head.

Message Buffer Case Study

Fixed buffer that has size n and operates in a FIFO manner. Initially, the buffer woud be empty. Then the *join* and *leave* operations can poccur whenever they are enabled, provided their pre-conditions are satisfied. Operations are assumed to be atomic. At all times, state schema for buffer is always satisfied. Something like loop invariant.

State Schema

$[MSG]$ (The exact nature of these messages is not important)

is the set of all possible messages that could ever be transmitted.

$max : \mathbb{N}$ (The actual value of max is not important)

is the constant maximum number of messages that can be held in the buffer at any one time.

$$\begin{array}{|l} Buffer \\ \hline items : \mathrm{seq}\, MSG \qquad \text{declaration} \\ \hline \#items \leqslant max \qquad \text{predicate} \end{array}$$

e.g. suppose $MSG = \{m_1, m_2, m_3\}$ and $max = 4$

Then $items = \langle m_1, m_2 \rangle$ is an instance, but $items = \langle m_3, m_1, m_1, m_2, m_2 \rangle$ is not

If the predicate fails, then it is no longer an instance. Predicate is some condition that the declared items must fulfill at every state.

Operation Schema

Specifies how the system can change. Convention: primed identifiers denotes after the change, while unprimed identifiers denotes before. k' refers to new value of k after the operation.

Join Operation

Operation when messages are added to the buffer.

$$\begin{array}{|l} \Delta Buffer \\ items, items' : \mathrm{seq}\, MSG \\ \hline \#items \leqslant max \\ \#items' \leqslant max \end{array}$$

and we can now write *Join* by including this schema:

$$\begin{array}{|l} Join \\ \Delta Buffer \\ msg? : MSG \\ \hline \#items < max \\ items' = items \frown \langle msg? \rangle \end{array}$$

? denotes an *input*. Additionally, there is an implicity $\wedge$ between each line of the predicate. Everything must equate to true all the time.

Schema Inclusion

Able to abstract out repeated declarations and predicates. Only can merge if types are compatible (same type).

$$\begin{array}{|l} \Delta Buffer \\ items, items' : \mathrm{seq}\, MSG \\ \hline \#items \leqslant max \\ \#items' \leqslant max \end{array}$$

and we can now write *Join* by including this schema:

$$\begin{array}{|l} Join \\ \Delta Buffer \\ msg? : MSG \\ \hline \#items < max \\ items' = items \frown \langle msg? \rangle \end{array}$$

Join

$items, items' : \text{seq } MSG$
$msg? : MSG$

$\#items \leqslant max$
$\#items' \leqslant max$
$\#items < max$
$items' = items \frown \langle msg? \rangle$

Leave Operation

Operation when messages leave the buffer.

Leave

$\Delta Buffer$
$msg! : MSG$

$items \neq \varnothing$
$items = \langle msg! \rangle \frown items'$

! denotes an *output*.

Initial State

States are used to show how the buffer changes from one instance to another.

$Buffer_{\text{INIT}}$

$items : \text{seq } MSG$

$\#items \leqslant max$
$items = \langle\, \rangle$

Auxiliary Variables

Doesn't alter functionality but aids in analysis (proving).

RecordedBuffer

$Buffer$
$inhist : \text{seq } MSG$
$outhist : \text{seq } MSG$

$RecordedBuffer_{\text{INIT}}$

$RecordedBuffer$
$Buffer_{\text{INIT}}$

$inhist = \langle\, \rangle$
$outhist = \langle\, \rangle$

RecordedJoin

$\Delta RecordedBuffer$
$Join$

$inhist' = inhist \frown \langle msg? \rangle$
$outhist' = outhist$

RecordedLeave

$\Delta RecordedBuffer$
$Leave$

$inhist' = inhist$
$outhist' = outhist \frown \langle msg! \rangle$

Can use auxiliary variables to help in proving that a system behaves a particular way using structural induction.

Theorem

$\forall RecordedBuffer \bullet inhist = outhist \frown items$

Proof:
Use structural induction.
Initially $inhist = outhist = items = \langle\, \rangle$,
so the predicate is true.

Suppose the predicate is true, and *RecordedJoin* occurs.
After the operation

$$inhist' = inhist \frown \langle msg? \rangle \wedge outhist' = outhist \wedge items' = items \frown \langle msg? \rangle$$

Hence: $inhist'$
$= inhist \frown \langle msg? \rangle = (outhist \frown items) \frown \langle msg? \rangle$
$= outhist \frown (items \frown \langle msg? \rangle) = outhist' \frown items'$

and the predicate remains true. A similar argument shows that the operation *RecordedLeave* also preserves the predicate. □

Conjunction

Merging of schemas.

$$SlowRecordedBuffer \mathrel{\widehat=} SlowBuffer \wedge RecordedBuffer$$

Declarations are union-ed and predicates are conjunct-ed.

Disjunction

Showing either or case, usually for error handling.

A

$x : T_1$
$y : T_2$

$P(x, y)$

B

$y : T_2$
$z : T_3$

$Q(y, z)$

conjunction

$A \wedge B$

$x : T_1;\ y : T_2;\ z : T_3$

$P(x, y) \wedge Q(y, z)$

disjunction

$A \vee B$

$x : T_1;\ y : T_2;\ z : T_3$

$P(x, y) \vee Q(y, z)$

Composition

A

$x : T_1$
$y : T_2$

$P(x, y)$

AOP_1

ΔA
$t_3? : T_3$
$t_4! : T_4$

$Q_1(x, x', y, y', t_3?, t_4!)$

AOP_2

ΔA
$t_5? : T_5$
$t_6! : T_6$

$Q_2(x, x', y, y', t_5?, t_6!)$

$AOP_1 \fatsemi AOP_2$

ΔA
$t_3? : T_3;\ t_4! : T_4;\ t_5? : T_5;\ t_6! : T_6$

$\exists x'' : T_1;\ y'' : T_2 \bullet$
$Q_1(x, x'', y, y'', t_3?, t_4!) \wedge Q_2(x'', x', y'', y', t_5?, t_6!)$

Basically the output of OP_1 must satisfy the precondition of OP_2. The output depends on which operation gives the output and where it is in the composition.

Piping

Pass the outputs of OP_1 to OP_2. The output would always be a result of OP_2. $OP_1 \gg OP_2$

Class Construct

Buffer

$(max, \text{INIT}, Join, Leave)$ [visibility list]

$max : \mathbb{N}$ [constant]
$items : \text{seq } MSG$ [state schema]

$\#items \leqslant max$

INIT

$items = \langle\, \rangle$ [initial state]

Join

$\Delta(items)$
$msg? : MSG$

$\#items < max$
$items' = items \frown \langle msg? \rangle$

Leave

$\Delta(items)$
$msg! : MSG$ [operation]

$items \neq \langle\, \rangle$
$items = \langle msg! \rangle \frown items'$

Objects have integrity - change state via class operations only. Objects have persistence - exist from creation to deallocation. Communicate by message passing. $a.op$, op performed on object a. $a.op||b.op$. Parallel execution of operations. $\downarrow A$ schema is subclass of A. $\restriction (A, B, C)$ refers to things that are publicly accessible by others.

Events

Engaged by a process. Each event is atomic.

Alphabet

Set of events that a process can possibly engage in. $\{coin, choc\}$

Traces

A finite sequence of events. A deterministic process is specifed by the set of traces denoting its possible behavior. Enclosed in ⟨ and ⟩. Any execution of the process will be one of these sequences. Read left to right in order.

Basic Process Notation

- lower case denote events - x, y, z
- uppercase denote processes and sets of events- X, Y
- alphabet is denoted by add α before the process - αP
- a set of traces is denoted by wrapping the process in a function - $traces(P)$

Trace Notation

- *seq* A denotes the set of all finite sequences of events from A
- possible to concantenate sequences together.

(1) $STOP_A$ is the process with alphabet A that can do nothing.
$traces(STOP_A) = \{\langle\, \rangle\}$

(2) $CLOCK$ is the process with $\alpha CLOCK = \{tick\}$ which can 'tick' at any time.
$traces(CLOCK) = tick^*$

(3) VM is the process with $\alpha VM = \{coin, choc\}$ which repeatedly supplies a chocolate after a coin is inserted.
$traces(VM) = \{s : \text{seq}\{coin, choc\} \mid \exists n : \mathbb{N} \bullet s \leqslant \langle coin, choc \rangle^n\}$

(4) $WALK$ is a one-dimensional random walk process with $\alpha WALK = \{left, right\}$.
$traces(WALK) = (left \cup right)^*$

(5) $LIFE$ is the process with $\alpha LIFE = \{beat\}$ which can stop (die) at any time.
$traces(LIFE) = beat^*$

Asterisk means that it will occur infintely. While n means that the traces are concatenated with itself n times.

Prefix

A process which may participate in event a then act according to process description P is written $a \to P$. a must be started first before P can start.

$$SHORTLIFE = (beat \to (beat \to STOP)) = beat \to beat \to STOP$$

Sequential Composition

A distinguished event is used to represent and detect process termination. $P;\ Q$ acts as P until P terminates by communicating a distinguished event and then proceeds to act as Q. Termination signal is hidden from process environment and occurs automatically when signalled. The process which may only terminate is written $SKIP$

Parallel Composition

$P|[X]|Q$ - 2 processes P, Q synchronised on a set of event X. No event in X may occur unless enabled **jointly by both** P and Q. When events in X occurs, they occur **in both** P, Q simultaneously. Events not in X may occur in P or Q separately but **not jointly**.

$$(a \to P) \,[|\, a \,|]\, (c \to a \to Q)$$

all a events must be synchronisations between the two processes.

In the above example, c can start first but the LHS cannot start since they are synchronised on a. Once RHS reaches a, then both can begin their execution. For $P|||Q$, both components can execute concurrently without any synchronisation.

Choice

$(a \to P) \Box (b \to Q)$ begins with a, b enabled. The environment chooses which one of these does it want to execute. Alternatively, $(a \to P) \sqcap (b \to Q)$ means that the internal state can randomly choose which process(es) to enable. The environment cannot affect internal choices. Nondeterminism can be introduced into a nominally deterministic choice if the initial events of boths sides of the choice are identical.

$$(a \to a \to \text{STOP}) \Box (a \to b \to \text{STOP})$$

is equivalent to

$$a \to \big((a \to \text{STOP}) \sqcap (b \to \text{STOP})\big)$$

Channel

Collection of events of the form $c.n$, where c is called the channel name and the collection of suffixes is called the values of the channel. When $c.n$ occurs, the value n is communicated on channel c. When value of communication is determined by channel (external choice) its called **input**. And when determined by internal state of process (internal choice) its called **output**.
$c?n : N \to P(n)$ describes behavior over a range of allowed inputs. $c!n : N \to P(n)$ represents a range of outputs.

Interrupt

$P_1 \triangledown e \to P_2$, P_1 executes until the first occurance of e occurs, then control passed to P_2.

Recursion

$\mu P \bullet a?n : \mathbb{N} \to b!f(n) \to P$. Repeatedly input a on channel a, calculates some function f of the input and then output the result on channel b.

State Parameters

Finite representation of an infinite family of process descriptions : $P_{n:N} \triangleq Q(n)$.

Sample Traces

$$traces(a \rightarrow P) = \{t : \text{seq } A \mid t = \langle\rangle \lor \text{head } t = a \land \text{tail } t \in traces(P)\}$$

$$traces(a \rightarrow P \mid b \rightarrow Q) = traces(a \rightarrow P) \cup traces(b \rightarrow Q)$$

$$traces(x : B \rightarrow P(x)) = \{t : \text{seq } A \mid t = \langle\rangle \lor \text{head } t \in B \land \text{tail } t \in traces(P(\text{head } t))\} = \cup\{x : B \bullet traces(x \rightarrow P(x))\}$$

$$traces(\mu X : A \bullet F(X)) = \cup n : \mathbb{N} \bullet traces(F^n(STOP_A))$$

$$VM = \mu X : \{coin, choc\} \bullet (coin \rightarrow choc \rightarrow X)$$

so in this case

$$F(X) = coin \rightarrow choc \rightarrow X$$

and

$$F^0(STOP_A) = STOP_A$$
$$F(STOP_A) = coin \rightarrow choc \rightarrow STOP_A$$
$$F^2(STOP_A) = F(F(STOP_A)) = coin \rightarrow choc \rightarrow (coin \rightarrow choc \rightarrow STOP_A) = coin \rightarrow choc \rightarrow coin \rightarrow choc \rightarrow STOP_A$$
$$F^3(STOP_A) = \cdots$$

In order of traces that appear in the images:

1. the trace is either empty (when the environment doesnt allow a to happen) or the head of the trace would be a followed by some trace in P
2. trace of choices gives a union of traces.
3. first event can be anything from set B. Either the trace is empty, or the head of the trace must exist in B and the tail of the traces of P where the parameter is head of t.

Concurrency

Since $P||Q$ must co-operate on al common actions; in general alphabets of both are the same. To determine deadlock - calculate common events and see if they wait on the common alphabets somehow. If alphabets are declared, must be synchronised on every event in the alphabet. If system has multiple processes, they must be synchronised on the alphabets in order.

Laws of Concurrency

- $P||Q = Q||P$
- $P||(Q||R) = (P||Q)||R$
- $P||STOP_{\alpha P} = STOP_{\alpha P}$

$$a \in (\alpha P - \alpha Q)$$
$$b \in (\alpha Q - \alpha P)$$
$$\{c, d\} \subseteq (\alpha P \cap \alpha Q)$$

L4A $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

L4B $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP \quad \text{if } c \neq d$

L5A $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))$

L5B $(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)$

L6 $(a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q)) \mid b \rightarrow ((a \rightarrow P) \parallel Q))$

Traces for Concurrency

If $t \in \text{seq } A$ then

$$t \upharpoonright B$$

denotes the trace of t restricted to events in B. e.g.

$$\langle c, a, d, b, a, c\rangle \upharpoonright \{b, c\} = \langle c, b, c\rangle$$

If

$$A = \alpha P \cup \alpha Q$$

then

$$traces(P \parallel Q) = \{t : \text{seq } A \mid (t \upharpoonright \alpha P) \in traces(P) \land (t \upharpoonright \alpha Q) \in traces(Q)\}$$

Interleavng

Similar to concurrency but they must be synchronised on termination. All events must terminate together.

Timed Extension of CSP

$P = *CSP\#\ Process\ Constructs*$

- $Wait[d]$ – *delay*
- $P\ timeout[d]\ Q$ – *timeout*
- $P\ interrupt[d]\ Q$ – *timed interrupt*
- $P\ deadline[d]$ – *deadline*

$$\frac{t \leq d}{(V, Wait[d]) \xrightarrow{t} (V, Wait[d-t])}\ [\ de1\ ] \qquad \frac{}{(V, Wait[0]) \xrightarrow{\tau} (V, Skip)}\ [\ de2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V, P\ timeout[d]\ Q) \xrightarrow{x} (V', P')}\ [\ to1\ ]$$

$$\frac{(V,P) \xrightarrow{\tau} (V',P')}{(V, P\ timeout[d]\ Q) \xrightarrow{\tau} (V', P'\ timeout[d]\ Q)}\ [\ to2\ ]$$

$$\frac{(V,P) \xrightarrow{t} (V,P'), t \leq d}{(V, P\ timeout[d]\ Q) \xrightarrow{t} (V, P'\ timeout[d-t]\ Q)}\ [\ to3\ ]$$

$$\frac{}{(V, P\ timeout[0]\ Q) \xrightarrow{\tau} (V, Q)}\ [\ to4\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V, P\ interrupt[d]\ Q) \xrightarrow{x} (V', P'\ interrupt[d]\ Q)}\ [\ it1\ ]$$

$$\frac{(V,P) \xrightarrow{t} (V,P'), t \leq d}{(V, P\ interrupt[d]\ Q) \xrightarrow{t} (V, P'\ interrupt[d-t]\ Q)}\ [\ it2\ ]$$

$$\frac{}{(V, P\ interrupt[0]\ Q) \xrightarrow{\tau} (V', Q)}\ [\ it3\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V, P\ deadline[d]) \xrightarrow{x} (V', P'\ deadline[d])}\ [\ dl1\ ]$$

$$\frac{(V,P) \xrightarrow{t} (V,P'), t \leq d}{(V, P\ deadline[d]) \xrightarrow{t} (V, P'\ deadline[d-t])}\ [\ dl2\ ]$$

For wait, the premise states that the total wait time t cannot exceed the total time d. When a wait occurs, the resulting time left is $d - t$. In the event that a wait is done when the process has no more time, it is skipped instead. For timeout and interrupts, such events occurs only after $d = 0$, otherwise, its just similar to wait.

Linear Temporal Logic

Linear Temporal Logic (LTL)

In PAT, we support the full set of LTL syntax. Given a process P(), the following assertion asks whether P() satisfies the LTL formula.

#assert P() |= F;

where F is an LTL formula whose syntax is defined as the following rules,

F = e | prop | [] F | <> F | X F | F1 U F2 | F1 R F2

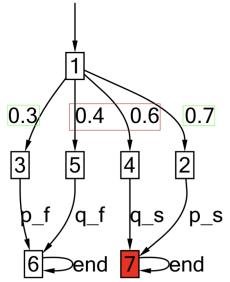
where e is an event, prop is a pre-defined proposition, [] reads as "always" (also can be written as 'G' in PAT), <> reads as "eventually" (also can be written as 'F' in PAT), X reads as "next", U reads as "until" and R reads as "Release" (also can be written as 'V' in PAT).

An LTL formula can be evaluated over an infinite sequence of truth evaluations and a position on that path. The semantics for the modal operators is given as follows.

Common Questions

```
var success = false;

P = pcase{
  [0.3]: p_f -> R
  [0.7]: p_s {success=true;} -> R
};

Q = pcase{
  [0.6]: q_s {success=true;} -> R
  [0.4]: q_f -> R
};

R = end -> R;

A = P [] Q;
```

P takes from PA



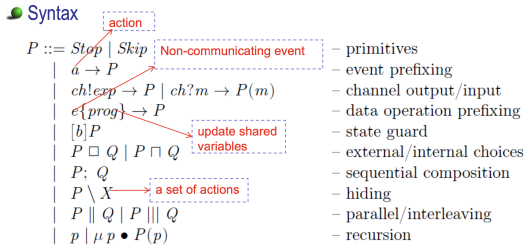
1. The active process in state 2 is $p\_s\{success = true\} - > R$
2. If P performs an event first, the invisible event from compositional pcase to a distribution of branches. Basically just to the parent node of the pcases first.
3. *default* takes whatever probability is left.

Reachability

```
var success = false;
P3 = pcase{
  [0.6]: p_s {success=true;} -> R
  [0.4]: p_f -> P4    };

P4 = pcase{
  1: end -> R
  1: back {success=false} -> P3  };

R = end -> R;
#define goal success==true;
#assert P3 reaches goal with prob;

// What are the [min, max] here?
```

Notice that numbers in [] must add up to 1 since this is the direct probability. But for numbers not in square brackets, then it would be just a weightage.

Background



Operational Semantics

$$\frac{Premises}{Conclusion}$$

- e.g.,

$$\frac{P \xrightarrow{a} P'}{P \,\square\, Q \xrightarrow{a} P'}$$

The premises on top contributes to the conclusion some how. In the above example, when the process P

is executed, it will becomes P', hence its reflected in the conclusion as well. Integrate CSP modelling operators with sequential programs. Some other semantics are shown below:

- $STOP$,
- $SKIP$,

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}\ [\ skip\ ]$$

- Prefixing,

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}\ [\ prefixing\ ]$$

- External choice[a],

$$\frac{P \xrightarrow{a} P'}{(P \,\square\, Q) \xrightarrow{a} P'}\ [\ extchoice1\ ] \qquad \frac{Q \xrightarrow{a} Q'}{(P \,\square\, Q) \xrightarrow{a} Q'}\ [\ extchoice2\ ]$$

- Internal choice, let τ be the silent invisible event,

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}\ [\ intchoice1\ ] \qquad \frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}\ [\ intchoice2\ ]$$

Operational Semantics: Sequential Composition

In process $P;\ Q$, P takes control first and Q starts only when P has finished. Let $\checkmark$ be a distinguished event denoting termination.

$$\frac{P \xrightarrow{a} P'}{(P;\ Q) \xrightarrow{a} (P';\ Q)}\ [\ seq1\ ] \qquad \frac{P \xrightarrow{\checkmark} P'}{(P;\ Q) \xrightarrow{\tau} Q}\ [\ seq2\ ]$$

Operational Semantics: Interrupt

In process $P \triangledown Q$, whenever an event is engaged by Q, P is interrupted and the control transfer to Q.

$$\frac{P \xrightarrow{a} P'}{(P \triangledown Q) \xrightarrow{a} (P' \triangledown Q)}\ [\ interrupt1\ ] \qquad \frac{Q \xrightarrow{a} Q'}{(P \triangledown Q) \xrightarrow{a} Q'}\ [\ interrupt\ 2]$$

In process $P \,|||\, Q$, P and Q behaves independently[a].

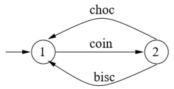
$$\frac{P \xrightarrow{a} P'}{(P \,|||\, Q) \xrightarrow{a} (P' \,|||\, Q)}\ [\ interleave1\ ]$$

$$\frac{Q \xrightarrow{a} Q'}{(P \,|||\, Q) \xrightarrow{a} (P \,|||\, Q')}\ [\ interleave2\ ]$$

[a]except termination. Assume that a is not $\checkmark$.

Labeled Transition System

Contains a set of states, an initial state (where system starts from) and a labeled transition relation.



where $VMS = coin \rightarrow (choc \rightarrow VMS \,\square\, bisc \rightarrow VMS)$, state 1 represents the process VMS and state 2 represents the process $(choc \rightarrow VMS \,\square\, bisc \rightarrow VMS)$.

Methodology

1. specify the problem as PAT
2. get the alphabets of each process
3. apply the operational semantics rules (one at a time) to build the labeled transition system.

CSP Trace, Failure and Divergence Semantics

Failure is a pair (s, X), consisting of a trace s, and a refusal set X which identifies the events that a process may refuse once it has executed the trace s. Basically once the trace is executed, it would refuse to execute events in X.
Process is divergence if it performs an endless series of hidden actions.

$Clock = tick \rightarrow Clock$

The traces of this process are defined as:

$\text{traces}(Clock) = \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \cdots\} = \{tick\}^*$

Now, consider the following process, which conceals the *tick* event of the *Clock* process:

$P = Clock \backslash tick$

By definition, *P* is called a divergent process.