The mathematics of CSP

There are three main ways of formalising what CSP means:

- The operational semantics discussed earlier.
- The algebraic properties of operators like $\Box$ and $\sqcap$.
- Identifying a process with the set of its behaviours.

While each of these can tell you something by itself, they give a more complete picture together and help to explain each other.

Algebra

Here are some laws we are all familiar with:

$x + y = y + x$	*commutative*, or *symmetry*
$x \times y = y \times x$	ditto
$x \cup y = y \cup x$	ditto
$(x + y) + z = x + (y + z)$	*associativity*
$(x + y) \times z = (x \times z) + (y \times z)$	(right) *distributive* law
$0 + x = x$	*unit* law
$\{\} \cap x = \{\}$	*zero* law
$x \cup x = x$	*idempotence*

What similar laws do CSP operators satisfy?

Laws of choice

The choice between P and P is no choice at all:

$$P \,\Box\, P \;=\; P \qquad \langle \Box\text{-idem}^* \rangle$$

$$P \sqcap P \;=\; P \qquad \langle \sqcap\text{-idem} \rangle$$

The choice between P and Q is the same as that between Q and P:

$$P \,\Box\, Q \;=\; Q \,\Box\, P \qquad \langle \Box\text{-sym} \rangle$$

$$P \sqcap Q \;=\; Q \sqcap P \qquad \langle \sqcap\text{-sym} \rangle$$

And the choice between P, Q and R is the same however bracketed:

$$P \,\Box\, (Q \,\Box\, R) \;=\; (P \,\Box\, Q) \,\Box\, R \qquad \langle \Box\text{-assoc} \rangle$$

$$P \sqcap (Q \sqcap R) \;=\; (P \sqcap Q) \sqcap R \qquad \langle \sqcap\text{-assoc} \rangle$$

Distributivity

If $F(\cdot)$ is a CSP construct, what is the difference between

$$F(P \sqcap Q) \quad \text{and} \quad F(P) \sqcap F(Q)?$$

If F doesn't run more than one copy of its argument, *none*.

There is then no way of telling whether the choice was made before or after applying F.

Just about all individual CSP operators satisfy this principle, and therefore have *distributive* laws over $\sqcap$ (and also $\bigsqcap$):

$$P \,\Box\, (Q \sqcap R) \;=\; (P \,\Box\, Q) \sqcap (P \,\Box\, R) \qquad \langle \Box\text{-dist} \rangle$$

$$P \,\Box \bigsqcap S \;=\; \bigsqcap \{P \,\Box\, Q \mid Q \in S\} \qquad \langle \Box\text{-Dist} \rangle$$

$$a \to (P \sqcap Q) \;=\; (a \to P) \sqcap (a \to Q) \qquad \langle \text{prefix-dist} \rangle$$

$$?x : A \to (P \sqcap Q) \;=\; (?x : A \to P) \sqcap (?x : A \to Q) \, \langle \text{input-dist} \rangle$$

While there are distributive laws over other operators, "distributivity" unqualified always means over $\sqcap$ and $\bigsqcap$.

When distributivity fails

Operators that run their arguments more than once are usually not distributive, for example recursion:

$$\mu\, p.((a \to p) \sqcap (b \to p)) \neq$$
$$(\mu\, p.a \to p) \sqcap (\mu\, p.b \to p)$$

This is because in $F(P \sqcap Q)$, the two copies may behave differently, while in $F(P) \sqcap F(Q)$ they must behave alike.

Distribution the other way

Note that in set theory $\cup$ and $\cap$ each distribute over the other.

In CSP there is a similar phenomenon with $\Box$ and $\sqcap$. For we have not only $\langle \Box\text{-dist} \rangle$, but also

$$P \sqcap (Q \Box R) = (P \sqcap Q) \Box (P \sqcap R) \qquad \langle \sqcap\text{-}\Box\text{-dist}^* \rangle$$

Both processes have the same options after the first visible action (obviously), and both can refuse exactly the same sets of actions on the first step.

This is a theoretically important law, but not one that gets used much in practice!

A step law

Step laws allow us to calculate the first step actions of a process, and are therefore central to our understanding of CSP:

The step law of $\Box$ is

$$
\begin{aligned}
(?x : A \to P) \,\Box\, (?x : B \to Q) \;=\; & \\
?x : A \cup B \to ((P \sqcap Q) & \\
\lhd x \in A \cap B \rhd & \\
(P \lhd x \in A \rhd Q)) &
\end{aligned}
$$

$\langle \Box\text{-s}$

Note how this simply formalises what we said about $\Box$ before.

All operators other than prefix-choice, $\sqcap$ and recursion will have a step law.

More laws

The laws of $STOP$:

$$STOP = ?x : \{\} \to P \qquad \langle STOP\text{-step} \rangle$$

$$STOP \,\Box\, P = P \qquad \langle \Box\text{-unit} \rangle$$

The law of recursion:

$$\mu\, p.P = P[\mu\, p.P/p] \qquad \langle \mu\text{-unwind} \rangle$$

See the book for laws of $\lhd b \rhd$.

What are laws for?

- Provide intuition and understanding to us.
- Give sanity tests for any proposed mathematical theory/semantice.
- Can be used to prove processes equal to each other.
- With more effort, provide a complete *algebraic* semantics for the language (see Chapter 13 for details).

Traces

A *trace* is a sequence of visible communications that a process might communicate: a process's history.

In general it might be finite or infinite (the latter being the history over an infinite time) but we will consider only the finite case in this course.

In any case every *prefix* (initial subsequence) of an infinite trace is a finite one.

$traces(P)$ is the set of all P's (finite) traces, and is necessarily a *nonempty*, *prefix-closed* subset of Σ^* (the finite sequences formed from Σ). If P and Q are two processes such that $traces(P) = traces(Q)$, then we write $P =_T Q$ and say they are *trace equivalent*.

The set of all such subsets of Σ^* is called the *traces model* and written $\mathcal{T}$.

Trace notation

- $\langle a_1, a_2, \ldots, a_n \rangle$ is the sequence containing a_1, a_2 to a_n in that order. Note that $\langle a, a, b \rangle$, $\langle a, b \rangle$ and $\langle b, a \rangle$ are all different.
- $\langle\rangle$ is the empty sequence.
- $s\hat{}t$ is the *concatenation* of s and t: $\langle a, b \rangle \hat{} \langle b, a \rangle = \langle a, b, b, a \rangle$.
- If $t = s\hat{}w$, then s is a *prefix* of t, written $s \leq t$ (a partial order).

Working out $traces(P)$

There is a rule for each CSP operator that shows the effect it has on traces:

- $traces(STOP) = \{\langle\rangle\}$
- $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid s \in traces(P)\}$ – this process has either done nothing, or its first event was a followed by a trace of P.
- $traces(?x : A \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A \wedge s \in traces(P[a/x])\}$ – this is similar: $P[a/x]$ means the substitution of the value a for all free occurrences of the identifier x.

More clauses of $traces(P)$

- $traces(c?x : A \to P) = \{\langle\rangle\} \cup \{\langle c.a\rangle^\frown s \mid a \in A \wedge s \in traces(P[a/x])\}$ – the same except for the use of the channel name.
- $traces(P \,\Box\, Q) = traces(P) \cup traces(Q)$ – this process offers the traces of P and those of Q.
- $traces(P \sqcap Q) = traces(P) \cup traces(Q)$ – since this process can behave like either P or Q.
- $traces(\bigsqcap S) = \bigcup\{traces(P) \mid P \in S\}$
- $traces(P \mathrel{\triangleleft\!\!\triangleleft} b \mathrel{\triangleright\!\!\triangleright} Q) = traces(P)$ if b evaluates to $true$; and $traces(Q)$ if b evaluates to $false$.

Note that the traces of $P \sqcap Q$ and $P \,\Box\, Q$ are the same: this strongly suggests that traces do not give a *complete* description of processes.

Recursion

The recursion $p = Q$ (or equivalently $\mu\, p.Q$) must (thanks to $\langle \mu$-unwind$\rangle$) satisfy

$$traces(\mu\, p.Q) = traces(\, Q[\mu\, p.Q/p])$$

In other words, $traces(\mu\, p.Q)$ is a value Y satisfying $Y = F(\, Y)$, where $F(X)$ is the traces of Q when a process with traces X is substituted for p in Q.

For example, if Q is $a \to p$,

$$F(X) = \{\langle\rangle\} \cup \{\langle a\rangle \hat{} s \mid s \in X\}$$

Of course, not all functions from $\mathcal{T}$ to $\mathcal{T}$ have a fixed point, just as $x \mapsto x + 1$ has none over the natural numbers, **BUT**

all CSP definable functions have a fixed point, and a *least* one (that is a subset of all others), and that is the correct value for recursions.

For example, the fixed point of $F(p) = a \to p$ is

$$\{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \langle a, a, a\rangle, \ldots\}$$

which is obviously the right answer.

Mutual recursion

The mathematics of mutual recursion is the same, except that instead of having functions from $\mathcal{T}$ to $\mathcal{T}$ we now have ones from $\mathcal{T}^\Lambda$ to itself where Λ is an *indexing* set with one member for each mutually defined process. $\mathcal{T}^\Lambda$ is the set of *vectors* of members of $\mathcal{T}$ indexed by Λ.

Λ may be finite: in

$$P = (a \to P) \,\Box\, (b \to Q)$$
$$Q = (c \to Q) \,\Box\, (b \to P)$$

Λ has size 2.

Or it may be infinite: in $COUNT$ we have $\Lambda = \mathbb{N}$ and the function operates on infinite vectors of trace-sets.

The value of a mutual recursion is the vector $\underline{X} \in \mathcal{T}^\Lambda$ which is the least fixed point of the function from $\mathcal{T}^\Lambda$ to itself the recursion generates. (The one with fewest traces in each component.)

Fixed point theories

Two different ways of looking at this: partial orders and metric spaces.

Partial orders give more generality (work for all recursions), while metric spaces only work for *guarded* ones, but the metric theory gives the most useful proof rules.

For the underlying mathematics see book and Appendix A of TPC.

Monotonicity and continuity

A function of $\mathcal{T}$ is *monotonic* if $P \subseteq Q$ implies $F(P) \subseteq F(Q)$, and *continuous* if $P_i \subseteq P_{i+1}$ for all i implies

$$F(\bigcup_{i=0}^{\infty} P_i) = \bigcup_{i=0}^{\infty} F(P_i)$$

Any distributive operator has both these properties. $P \subseteq Q$ implies $P \sqcap Q = Q$ (in traces) and so

$$F(Q) = F(P \sqcap Q) = F(P) \sqcap F(Q)$$

Hence monotone, and the definition of continuity is just restricted distributivity (identifying $\cup$ and $\sqcap$).

Thus all the individual CSP operators have these properties, and since it can be shown that compositions of monotone (continuous) operators are monotone (continuous), *all* CSP terms have these properties over $\mathcal{T}$.

Tarski's theorem

Many versions of this (see Appendix A of TPC), but the one we need is that if $(\mathcal{X}, \leq)$ is a partial order in which (a) there is a least element $\bot$ and (b) each chain $P_0 \leq P_1 \leq P_2 \leq \ldots$ has a least upper bound $\bigsqcup_{i=0}^{n} P_i$, then every continuous function has a least fixed point given by

$$\bigsqcup \{f^n(\bot) \mid n \in \mathbb{N}\}$$

Recursion over $\mathcal{T}$

Since $(\mathcal{T}, \subseteq)$ has both these properties (least element being $STOP$), it follows that every CSP definable single recursion $\mu\, p.F(p)$ has traces

$$\bigcup_{i=0}^{\infty} F^n(STOP)$$

and that the value of a mutual recursion is given by

$$\bigcup_{i=0}^{\infty} F^n(\underline{STOP})$$

where $\underline{STOP}$ is an appropriate vector of $STOP$'s and $\bigcup$ is componentwise union on the vectors.

Intuition

Since $\mu\, p.F(P) = F^n(\mu\, p.F(p))$, every trace of $F^n(STOP)$ is one of $\mu\, p.F(p)$.

And since every finite trace s of $\mu\, p.F(p)$ takes a finite time to observe, the recursion can only have been unwound some finite number n times in this period. Necessarily s belongs to $F^n(STOP)$.

This justifies

$$\bigcup_{i=0}^{\infty} F^n(STOP)$$

Guarded recursions

A recursion is *guarded* if every recursive call is preceded by a communication, either directly

$$P = (a \to P) \,\Box\, (b \to P)$$

or indirectly

$$Q = a \to (Q \,\Box\, b \to Q)$$

(We will have to refine this definition a bit when more operators are introduced later.)

In practice, very nearly all sensible recursions are guarded.

Intuition behind the metric fixed point

Intuitively, if we want to know the first n steps of the behaviour of a guarded recursion $\mu\, p.F(p)$, all we have to do is unwind the recursion n times:

$$F^n(\mu\, p.F(p))$$

This is what lies behind the *metric* theory of recursion, and the principle of *Unique Fixed Points* (UFP):

If $P = F(P)$ is a guarded recursion (perhaps mutual), and Q is a process (or vector) such that $Q =_T F(Q)$ (in traces) then $P =_T Q$.

A simple metric

If $P \in \mathcal{T}$, its n-place restriction $P \downarrow n$ is just $\{s \in P \mid \#s \leq n\}$: the traces of length n or less.

We can define a metric (rather odd compared to the ones you may have seen in mathematics courses) over $\mathcal{T}$ by

$$d(P, P) = 0, \quad \text{and otherwise}$$

$$d(P, Q) = 2^{-n}, \quad n \text{ maximal such that } P \downarrow n = Q \downarrow n$$

Thus the longer it takes to tell P and Q apart, the closer they are.

Metric space theory

This makes $\mathcal{T}$ into a *complete metric space* (i.e., if P_i is a sequence such that for any $\epsilon > 0$ we can find m with all P_i, $i > m$ within ϵ of each other, then P_i converges to a limit P').

Any guarded recursion corresponds to a *contraction mapping* over $\mathcal{T}$: $d(F(P), F(Q)) \leq d(P, Q)/2$, and so has a unique fixed point by the Banach contraction mapping theorem (see Appendix A of TPC).

All the above is easily modified to take account of mutual recursions.

Using UFP

Here are some trivial examples of the unique fixed point principle. Recall the recursions:

$$P_1 = up \rightarrow down \rightarrow P_1$$

$$P_2 = up \rightarrow down \rightarrow up \rightarrow down \rightarrow P_2$$

$$P_u = up \rightarrow P_d$$

$$P_d = down \rightarrow P_u$$

Using $\langle \mu\text{-unwind} \rangle$ twice, it is easy to see that P_1 satisfies the definition of P_2. UFP (applied to the guarded P_2 recursion) then proves them equivalent.

Unwinding P_u twice shows that it satisfies the definition of P_1, proving them equivalent. So P_1, P_2 and P_u are all equivalent.

Mutual UFP

Most interesting uses of UFP seem to be on mutual recursions (usually one-step tail recursions where we are defining one process for each state a system can get into). The following process is an integer counter process

$$ZCOUNT_n = up \rightarrow ZCOUNT_{n+1} \\ \Box\ down \rightarrow ZCOUNT_{n-1}$$

A bit of thought tells you that the index is actually irrelevant here: we might suspect that all $ZCOUNT_n$ behave like

$$AROUND = up \rightarrow AROUND \ \Box\ down \rightarrow AROUND$$

This can be *proved* by UFP considering the vector of processes $\underline{A}$ (one component for each integer) with each component $AROUND$.

If F_{ZC} is the function of the $ZCOUNT$ recursion, $F_{ZC}(\underline{A}) = \underline{A}$:

$$
\begin{aligned}
(F_{ZC}(\underline{A}))_n &= up \rightarrow A_{n+1} \,\Box\, down \rightarrow A_{n-1} \\
&= up \rightarrow AROUND \,\Box\, down \rightarrow AROUND \\
&= AROUND \;=\; A_n
\end{aligned}
$$

Traces and laws

Note that every law we state implies that the trace sets of the two sides are always equal. For example, thanks to ⟨prefix-dist⟩, we need

$$a \to (P \sqcap Q) =_T (a \to P) \sqcap (a \to Q)$$

Since we can work out both sides in terms of $traces(P)$ and $traces(Q)$ using the trace semantics, this provides a test of our semantics that could, in principle, go wrong.

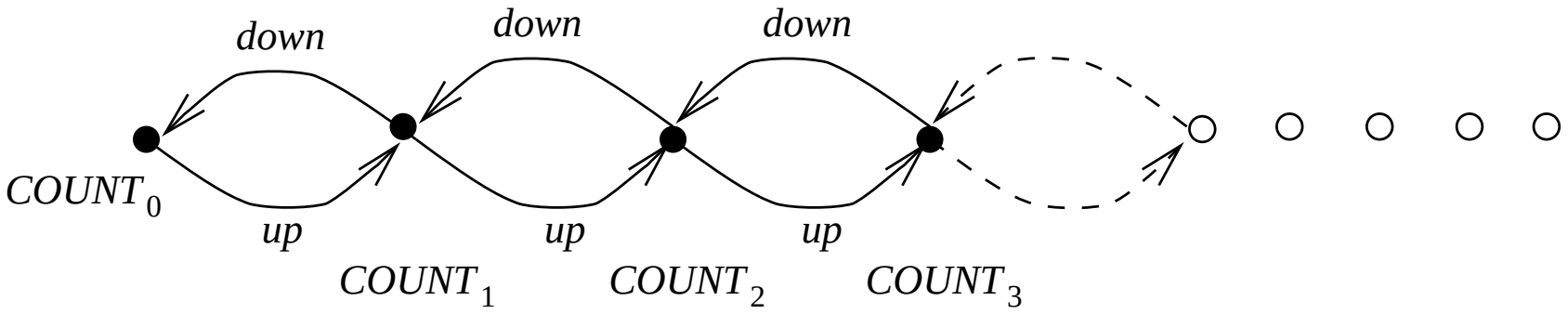
However in each case it is fairly easy to show that the implied result really is true, for example both sides above reduce to

$$\{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid s \in traces(P) \cup traces(Q)\}$$

Such results are boring to prove, but worth the effort if you can avoid stating a false law.

Traces and pictures

Evidently $traces(P)$ can also be computed from the LTS (transition graph) of P: possible sequences of actions, ignoring τ's.



This creates both an opportunity:

perhaps an easier way to work out traces (it is how tools tend to do it);

and another obligation:

to show that the traces got this way are the same as those calculated the other way (not a problem we address in this course).

Trace specifications

Traces allow us to formulate many useful specifications of processes.

A *behavioural trace specification* asserts some property of each trace of the process P: if R is a condition on traces,

$$P \,\mathbf{sat}\, R(tr) \quad \text{means} \quad \forall\, tr \in traces(P).R(tr)$$

(tr is the identifier conventionally used to represent an arbitrary trace.)

$R(tr)$ is usually expressed in some combination of predicate logic and trace notation.

In practice, just about all useful specifications on $traces(P)$ are lifted from individual trace specifications in this way.

Trace specifications are very good at limiting behaviour, but do nothing to force it: note that P and $P \sqcap STOP$ have exactly the same traces.

More trace notation

- If s is a finite sequence, $\#s$ denotes the *length* of s.
- If $s \in \Sigma^*$ and $A \subseteq \Sigma$ then $s \upharpoonright A$ means the sequence s *restricted* to A: the sequence whose members are those of s which are in A. $\langle\rangle \upharpoonright A = \langle\rangle$ and $(s\hat{}\langle a\rangle) \upharpoonright A = (s \upharpoonright A)\hat{}\langle a\rangle$ if $a \in A$, $s \upharpoonright A$ otherwise.
- If $s \in \Sigma^*$ then $s \downarrow c$ can mean two things depending on what c is.
  - If c is an *event* in Σ then it means the number of times c appears in s (i.e., $\#(s \upharpoonright \{c\})$).
  - If c is a *channel name* (associated with a non-trivial data type) it means the sequence of values (without the label c) that have been communicated along c in s. For example

$$\langle c.1, d.1, c.2, c.3, e.4\rangle \downarrow c = \langle 1, 2, 3\rangle$$

Example trace specifications

- P_1, P_2 and P_u all satisfy

$$tr \downarrow down \leq tr \downarrow up \leq tr \downarrow down + 1$$

- The specification of $COUNT_n$ is similar but less restrictive:

$$tr \downarrow down \leq tr \downarrow up + n$$

- $B^{\infty}_{\langle\rangle}$ and $COPY$ both satisfy the basic *buffer* specification:

$$tr \downarrow right \leq tr \downarrow left$$

Proof rules for sat

In Hoare's book the main way of handling specifications like these is through a series of proof rules such as:

$$STOP\ \mathbf{sat}\ (tr = \langle\rangle)$$

$$\frac{\forall\, a \in A.P(a)\ \mathbf{sat}\ R_a(tr)}{?a : A \to P\ \mathbf{sat}\ (tr = \langle\rangle \vee \exists\, a \in A.\, \exists\, tr'.\, tr = \langle a\rangle\hat{}tr' \wedge R_a(tr'))}$$

$$\frac{P\ \mathbf{sat}\ R(tr) \wedge Q\ \mathbf{sat}\ R(tr)}{P \,\Box\, Q\ \mathbf{sat}\ R(tr)}$$

Proof Rules

$$\frac{P \,\mathbf{sat}\, R(tr) \wedge Q \,\mathbf{sat}\, R(tr)}{P \sqcap Q \,\mathbf{sat}\, R(tr)}$$

which essentially translate the trace semantic rules into logic, and general inferences such as:

$$\frac{P \,\mathbf{sat}\, R(tr) \wedge \forall\, tr.R(tr) \Rightarrow R'(tr)}{P \,\mathbf{sat}\, R'(tr)}$$

$$\frac{P \,\mathbf{sat}\, R(tr) \wedge P \,\mathbf{sat}\, R'(tr)}{P \,\mathbf{sat}\, R(tr) \wedge R'(tr)}$$

Proof rule for recursion

Just as a recursive program calls upon itself, the proof rule has a "circular" feel to it. It is a form of *fixed point induction*. The following is the single recursion form:

Suppose $\mu\, P.F(P)$ *is a recursive definition, and that* X *is the (least) fixed point which it defines in* $\mathcal{T}$*. Suppose* $R(tr)$ *is a trace specification such that:*

- $STOP\ \mathbf{sat}\ R(tr)$*, and*
- $Y\ \mathbf{sat}\ R(tr) \Rightarrow F(\,Y\,)\ \mathbf{sat}\ R(tr)$

then $X\ \mathbf{sat}\ R(tr)$*.*

$STOP$ and trace specifications

The requirement that $STOP\,\mathbf{sat}\,R(tr)$ seems uncomfortable, since $STOP$ is a pretty useless process, and so you would not expect it to satisfy many sensible specifications.

But $traces(STOP) \subseteq traces(P)$ for *all* P, so

$$P\,\mathbf{sat}\,R(tr) \Rightarrow STOP\,\mathbf{sat}\,R(tr)$$

In other words, *any* behavioural trace specification satisfied by *any* process is satisfied by $STOP$.

This type of specification can ban a process from *doing* any incorrect action, but it cannot ban it from *not doing* anything.

Nevertheless, perhaps a majority of practical specifications applied to CSP processes are pure trace specifications.

Example

To prove $\mu\, p.a \to b \to p \,\mathbf{sat}\, tr \downarrow b \leq tr \downarrow a$.

- $STOP \,\mathbf{sat}\, tr = \langle\rangle$ and $tr = \langle\rangle \Rightarrow tr \downarrow b \leq tr \downarrow a$.

  Hence $STOP \,\mathbf{sat}\, tr \downarrow b \leq tr \downarrow a$.

- Assume $P \,\mathbf{sat}\, tr \downarrow b \leq tr \downarrow a$. Then
  - $b \to P \,\mathbf{sat}\, tr = \langle\rangle \vee tr = \langle b\rangle\hat{}tr' \wedge$
    $tr' \downarrow b \leq tr' \downarrow a$

    Hence $b \to P \,\mathbf{sat}\, tr \downarrow b \leq tr \downarrow a + 1$.
  - $a \to (b \to P) \,\mathbf{sat}\, tr = \langle\rangle \vee tr = \langle a\rangle\hat{}tr' \wedge$
    $tr' \downarrow b \leq tr' \downarrow a + 1$

    Hence $a \to b \to P \,\mathbf{sat}\, tr \downarrow b \leq tr \downarrow a$.

- Thus $\mu\, p.a \to b \to p \,\mathbf{sat}\, tr \downarrow b \leq tr \downarrow a$.

Trace refinement

Recall that $P \sqsupseteq Q$ if $P \sqcap Q = Q$. We can interpret this in traces as *trace refinement*:

$$P \sqsupseteq_T Q \equiv traces(P) \subseteq traces(Q)$$

(Note the reversed containment: the *more* traces a process has, the *less* refined it is.)

Three vital facts about refinement:

1. $P \sqsubseteq_T P' \wedge P' \sqsubseteq_T P'' \Rightarrow P \sqsubseteq_T P''$ *Transitivity*
2. $P \sqsubseteq_T P' \Rightarrow C[P] \sqsubseteq_T C[P']$ for any process context $C[\cdot]$ (syntax with a slot to insert a process). *Monotonicity*
3. $P \sqsubseteq_T P'$ and $P \,\mathbf{sat}\, R(tr)$ implies $P' \,\mathbf{sat}\, R(tr)$.

Together these justify *step-wise* and *compositional* development.

The last one helps to explain the use of reverse containment.

Characteristic processes

Let R be any specification such that $STOP \,\mathbf{sat}\, R(tr)$.

If $P \in S \Rightarrow P \,\mathbf{sat}\, R(tr)$, then $(\sqcap S) \,\mathbf{sat}\, R(tr)$ because

$t \in traces(\sqcap S) \Rightarrow \exists\, P \in S.t \in traces(P)$.

It follows that $P_R = \sqcap\{P \mid P \,\mathbf{sat}\, R(tr)\}$ is the most nondeterministic process satisfying $R(tr)$:

$$P \sqsupseteq_T P_R \Leftrightarrow P \,\mathbf{sat}\, R(tr)$$

Thus satisfaction of any behavioural trace specification (other than $false$) can be tested by refinement against its *characteristic process*.

Of course we can use any process that is trace equivalent to $\sqcap\{P \mid P \,\mathbf{sat}\, R(tr)\}$. (This is just as well.)

FDR makes finding characteristic processes an important skill.

Examples of characteristic processes over $\mathcal{T}$

- RUN_A The process never communicates outside A.
- $B^{\infty}_{\langle\rangle}$ The traces buffer specification

$$Buff\,(tr) \equiv tr \in (\{|\ left, right\ |\}^* \wedge tr \downarrow right \leq tr \downarrow left$$

- $COPY$ The one-place buffer specification

$$Buff\,(tr) \wedge \#(tr \downarrow left) \leq \#(tr \downarrow right) + 1$$

Using FDR

Traces checks are inserted into files thus:

```
assert Spec [T= Impl
```

In order to run, both specification and implementation must be *finite state*. Namely, their transition pictures (operational semantics) must be finite.

Thus $COUNT_0$ and $B^{\infty}_{\langle\rangle}$ may not be used.

The easiest way to violate this rule is (as in these two examples) via an infinite parameter type.

When a check fails, you can get debugging information back (a trace that the implementation performed which violates the specification).

Afters and initials

If P is any process, $initials(P)$ (sometimes written P^0) is the set of all its initial events

$$initials(P) = \{a \mid \langle a \rangle \in traces(P)\}$$

For example, $initials(STOP) = \{\}$ and $initials(?x : A \to P(x)) = A$.

If $s \in traces(P)$ then P/s ('P *after* s') is P after the trace s is complete. Over the traces model, P/s can be computed

$$traces(P/s) = \{t \mid s\hat{}t \in traces(P)\}$$

Status of "after" operator

P/s is not an ordinary part of the CSP language, because it is not implementable:

$$(STOP \sqcap a \rightarrow P)/\langle a \rangle = P$$

but the process on the left hand side cannot be forced to accept the a.

It is used for discussing and describing behaviour, e.g.

$$(P \,\Box\, Q)/s = P/s \quad \text{if } s \in traces(P) \setminus traces(Q)$$