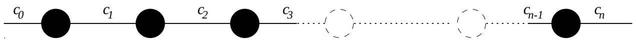
Why we need hiding

When we put processes in parallel it is often natural to regard the communications on internal channels as implementation detail.

There is no good reason to distinguish between N $COPY'(c_r, c_{r+1})$ processes arranged in parallel



and a parameterised n-place buffer definition using only the channels c_0 and c_n.

$$
\begin{array}{rcl}
BN(\langle\rangle) & = & c_0?x \to BN(\langle x\rangle) \\[6pt]
BN(s\hat{}\langle a\rangle) & = & c_n!a \to BN(s) \\
 & & \Box\ (c_0?x \to BN(\langle x\rangle\hat{}s\hat{}\langle a\rangle) \\
 & & \quad \lhd \# s < n-1 \rhd STOP)
\end{array}
$$

We need to be able to *hide* the internal events $\{\mid c_1, \ldots, c_{n-1} \mid\}$.

The hiding operator

Whenever P is a process and $X \subseteq \Sigma$

$$P \setminus X$$

behaves like P except that events from X become τ (internal) actions.

The natural combination of parallel and hiding is then

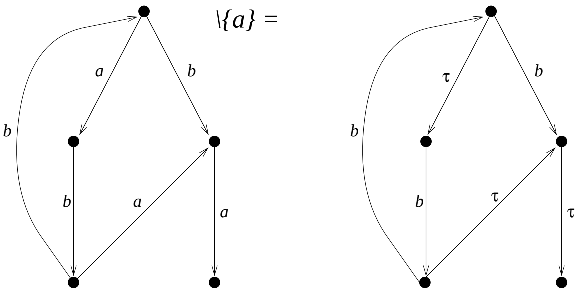
$$(P\ {}_X\|_Y\ Q) \setminus (X \cap Y) \quad \text{or} \quad (P \underset{Z}{\|} Q) \setminus Z$$

CSP models of realistic parallel systems usually combine parallel and hiding in this way.

Some other languages contain the combination of parallel and hiding, but neither operator individually.

Hiding in pictures

Hiding leaves the shape of the transition graph of any process alone, but turns appropriate labels into τ's:



If a τ action is enabled, then some action must happen quickly.

Thus no process ever gets stuck in a state with a τ.

States with and without τ's are respectively called *unstable* and *stable*.

Laws of hiding

This a one-place (unary) operator, so the laws look different:

$$(P \sqcap Q) \setminus X = (P \setminus X) \sqcap (Q \setminus X) \qquad \langle\text{hide-dist}\rangle$$

$$(P \setminus Y) \setminus X = (P \setminus X) \setminus Y \qquad \langle\text{hide-sym}\rangle$$

$$(P \setminus Y) \setminus X = P \setminus (X \cup Y) \qquad \langle\text{hide-combine}\rangle$$

$$P \setminus \{\} = P \qquad \langle\text{null hiding}\rangle$$

A prototype step law

$$(a \to P) \setminus X \;=\; \begin{cases} P \setminus X & \text{if } a \in X \\ a \to (P \setminus X) & \text{if } a \notin X \end{cases} \qquad \langle\text{hide-step 1}\rangle$$

Note that the case $a \in X$ requires our principle that τ's are not delayable.

This is not a full step law because they apply to processes with arbitrary ranges of initial actions.

The difficult case is then

$$(?x : A \to P) \setminus X$$

when A has members both inside and outside X.

Hidden versus visible

How does

$$(a \to P \;\Box\; b \to Q) \setminus \{a\}$$

behave?

Our intuition about hiding in transition pictures, and the law $\langle$hide-sym$\rangle$:

$$((a \to P \;\Box\; b \to Q) \setminus \{a\}) \setminus \{b\} \;=\; ((a \to P \;\Box\; b \to Q) \setminus \{b\}) \setminus \{a\}$$

both imply that hiding a cannot completely exclude the b option.

Hidden and visible options

The τ action in

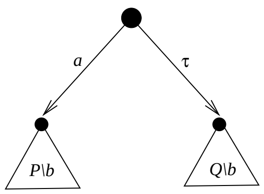
$$(a \to P \;\Box\; b \to Q) \setminus \{b\}$$

will occur soon after the start if the a is not communicated quickly: the answer is

$$(a \to P \setminus \{b\}) \rhd Q \setminus \{b\}$$

which is equivalent to

$$((a \to (Q \setminus \{b\})) \sqcap STOP) \;\Box\; (Q \setminus \{b\})$$



The step law of hiding

$$(?x : A \to P) \setminus X \;= \qquad\qquad \langle\text{hide-step}\rangle$$
$$\begin{cases} ?x : A \to (P \setminus X) & \text{if } A \cap X = \{\} \\ (?x : (A \setminus X) \to (P \setminus X)) \\ \rhd \bigsqcap\{(P[a/x]) \setminus X \mid a \in A \cap X\} & \text{if } A \cap X \neq \{\} \end{cases}$$

The nature of hiding means that the 'step' of behaviour this law generates may be internal (τ) rather than a visible action.

Moving hiding around

As a network is built up, we can either hide the internal events at each stage, or wait until all processes are in parallel and do it then.

This makes no difference, as shown by the following laws:

$$(P \;{}_X\|_Y\; Q) \setminus Z \;=\; (P \setminus Z \cap X) \;{}_X\|_Y\; (Q \setminus Z \cap Y) \qquad \langle\text{hide-}{}_X\|_Y\text{-dist}\rangle$$
provided $X \cap Y \cap Z = \{\}$

$$(P \underset{X}{\|} Q) \setminus Z \;=\; (P \setminus Z) \underset{X}{\|} (Q \setminus Z) \qquad \langle\text{hide-}\underset{X}{\|}\text{-dist}\rangle$$
provided $X \cap Z = \{\}$

These laws are important, because sometimes it is useful to leave all events visible till the end, and sometimes it is useful to hide as soon as possible.

Traces of hiding

The traces of $P \setminus X$ are very easy to compute: if we define $s \setminus X$, for any trace s, to be $s \upharpoonright (\Sigma \setminus X)$, then

$$traces(P \setminus X) \;=\; \{s \setminus X \mid s \in traces(P)\}$$

For example,

$$traces(COUNT_0 \setminus \{up\}) \;=\; \{\langle down\rangle^n \mid n \in \mathbb{N}\}$$
$$traces(COUNT_0 \setminus \{down\}) \;=\; \{\langle up\rangle^n \mid n \in \mathbb{N}\}$$

Do you think these two results of hiding are really as similar as the above suggests?

Hiding and nondeterminism

Hiding is the most common source of nondeterminism in CSP models of real systems.

Nondeterminism often appears where we conceal the internal details of parallel processes in contention for some resource.

$$P \;=\; a \to c \to STOP \;\Box\; b \to d \to STOP$$

offers the environment the choice between a and b, with subsequent behaviour depending on which option is chosen. If we give it the alphabet $X = \{a, b, c, d\}$ of all events it can perform,

$$N \;=\; P \;{}_X\|_{\{a,b\}}\; (a \to STOP \;{}_{\{a\}}\|_{\{b\}}\; b \to STOP)$$

does not change its behaviour.

But hiding the internal interactions of this network:

$$N \setminus \{a, b\}$$

produces the same behaviour as

$$(c \to STOP) \sqcap (d \to STOP)$$

Divergence

When internal events continue for ever we say that our process is *diverging*.

This can happen when a parallel/hiding combination can perform an infinite sequence of hidden internal actions without a visible one: this is *livelock*.

$\mathbf{div}$ is the process that does nothing but diverge. Externally it looks rather like $STOP$, but internally and theoretically it is different.

$COUNT_0 \setminus \{up\}$ can diverge, but $COUNT_0 \setminus \{down\}$ cannot.

Example

Imagine the *dithering philosophers*, in which a philosopher can put down the first fork without eating:

$$\begin{aligned} DITH_i \quad = \quad & picksup.i.i \rightarrow \\ & ((picksup.i.i{\oplus}1 \rightarrow eats.i \rightarrow \\ & putsdown.i.i{\oplus}1 \rightarrow putsdown.i.i \rightarrow DITH_i) \\ & \square\; putsdown.i.i \rightarrow DITH_i) \end{aligned}$$

Replacing $PHIL_i$ by $DITH_i$ solves the deadlock problem of the dining philosophers, at the expense of introducing livelock if the $\{|\ picksup, putsdown\ |\}$ events are hidden.

Unbounded nondeterminism

Hiding an infinite set of events X, or the use of $\sqcap S$ for infinite S, can create *unbounded nondeterminism*. Our process might have an infinite number of possible options to pick via τ's from a single state.

Unbounded nondeterminism is sometimes regarded as unrealistic, and it creates some theoretical difficulties. While none of these is insuperable, you should at least be aware

(i) that this is an issue, and

(ii) which operators can create unbounded nondeterminism.

Hiding *versus* guardedness

For obvious reasons, when we recurse through hiding the notion of guardedness (essential for UFP) becomes a lot more difficult:

$$P = a \rightarrow (P \setminus \{a\})$$

is not guarded, and has many fixed points over $\mathcal{T}$.

No recursion in which $\setminus X$ is applied (directly or indirectly) to a recursive call should ever be deemed guarded without very careful justification.

This applies equally to operators we will see later that are combinations of hiding with parallel.

Therefore the recursions

$$X = F(X)$$

we apply UFP to should not usually involve hiding, but there is nothing to stop the definition of the process Y using hiding, where Y is the one proved to be a fixed point of F.

They often do: see book for an example proof.

Injective renaming

A function is 1–1, or *injective*, if $f(x) = f(y)$ implies $x = y$.

If $f : \Sigma \rightarrow \Sigma$ is injective, then the process

$$f[P]$$

which performs $f(a)$ whenever P performs a, is just a copy of P using different events.

f may be a *partial* function, so long as every event P uses is in its domain.

The transition system of $f[P]$ is just that of P with f applied to all visible events.

Injective $f[P]$ has many laws: see book.

Examples

If f swaps $down$ and up, then $f[COUNT_0]$ acts like a counter through the *negative* numbers.

Recall

$$\begin{aligned} COPY &= left?x \rightarrow right!x \rightarrow COPY \\ COPY'(a,b) &= a?x \rightarrow b!x \rightarrow COPY'(a,b) \end{aligned}$$

If, for all x, $f(left.x) = a.x$ and $f(right.x) = b.x$ then

$$f[COPY] = COPY'(a,b)$$

Similar maps would relate the behaviours of the different $FORK_i$ processes and the different $PHIL_i$ processes, because these two collections both behave identically except for the names of events.

Of course, no f exists such that $f[FORK_i] = PHIL_j$.

Thus injective renaming provides an alternative to some uses of parameterisation (though the latter is generally preferable in these cases).

Non-injective functions

Uses of renaming in building realistic implementation models are usually injective functions (see especially Chapter 4), but we can achieve powerful and useful effects by examining the concept of "alphabet transformations" more generally.

The first case is $f[P]$, where f is a non-injective function: some events we could distinguish in P get identified in $f[P]$.

There is no difficulty in seeing how this generalisation works, but obviously it allows subtler effects.

Forgetful renaming

For example, it allows us to forget some details of a process:

$$\begin{aligned} SPLIT = \; & in?x : T \rightarrow \\ & ((out1.x \rightarrow SPLIT) \lessdot x \in S \gtrdot (out2.x \rightarrow SPLIT)) \end{aligned}$$

To forget about the contents of the messages, apply $forget$ which remembers only the channel name. $forget[SPLIT]$ is equivalent to

$$SPLIT' = in \rightarrow (out1 \rightarrow SPLIT' \sqcap out2 \rightarrow SPLIT')$$

Nondeterminism appears because we have lost the detail which allows to decide which way to send each message.

Why forget?

The transformation on the previous slide appears odd, but

- it allows us to demonstrate that some aspect of the correctness of the system does not depend on precisely how decisions are made, and
- in cases where this is true the details of decision making frequently clutter proofs.
- It will often save states on FDR.

Like hiding, forgetful renaming is a form of *abstraction*.

When $f^{-1}(x)$ ($= \{y \in \Sigma \mid f(y) = x\}$) is infinite (e.g. because the types of forgotten data are) you have to be particularly careful because $f[P]$ might introduce unbounded nondeterminism.

Relations

The function in $f[P]$ can only map each event of P to a single target.

It is surprisingly often useful to map events to several alternatives.

This can be done using *inverse functions*: see Hoare's book, or (more generally) using *relations*. We discuss the latter because it is what CSP$_M$ (in essence) uses.

A relation between X and Y is a set of pairs $(x, y) \in X \times Y$.

If R is a relation and $(x, y) \in R$ we write $x\,R\,y$.

If R is a relation, its *domain* and *range* are respectively

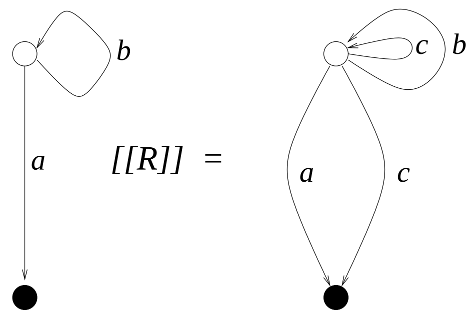
$$\begin{array}{rcl} dom(R) & = & \{x \mid \exists\, y . x\,R\,y\} \\ ran(R) & = & \{y \mid \exists\, x . x\,R\,y\} \end{array}$$

Relational renaming

We are interested in relations R between Σ and itself whose domain includes all the events of P.

$P[\![R]\!]$ is a process which, when P performs a and $a\,R\,b$, can perform b.

If a has several images under R, then each arc labelled a in P's transition diagram gets multiplied in $P[\![R]\!]$'s: if $R = \{(a, a), (a, c), (b, b), (b, c)\}$ then



Laws of renaming

Relational renaming is the most general, so we give laws and traces rules for it:

$$(P \sqcap Q)[\![R]\!] = P[\![R]\!] \sqcap Q[\![R]\!] \qquad \langle [\![R]\!]\text{-dist}\rangle$$

$$(P \sqcap Q)[\![R]\!] = P[\![R]\!] \sqcap Q[\![R]\!] \qquad \langle [\![R]\!]\text{-}\Box\text{-dist}\rangle$$

If the initials of P' are A, then those of $P'[\![R]\!]$ are $R(A) = \{y \mid \exists\, x \in A. (x, y) \in R\}$:

$$\begin{array}{l} (?x : A \to P)[\![R]\!] \;=\; \qquad \langle [\![R]\!]\text{-step}\rangle \\ \quad ?y : R(A) \to \bigsqcap\{(P[z/x])[\![R]\!] \mid z \in A \wedge z\,R\,y\} \end{array}$$

Notice how this last law makes the introduction of nondeterminism clear.

Traces of $P[\![R]\!]$

We just apply R in the obvious way to traces:

$$\begin{array}{rcl} \langle a_1, \ldots a_n\rangle R^* \langle b_1, \ldots, b_m\rangle & \Leftrightarrow & n = m \wedge \forall\, i \leq n . a_i\,R\,b_i \\ traces(P[\![R]\!]) & = & \{t \mid \exists\, s \in traces(P) . s\,R^*\,t\} \end{array}$$

Notation

We can use a substitution-like notation:

- $P[\![^{a}/_{b}]\!]$ means b is replaced by a, and everything else (including a) is unchanged.
- $P[\![^{a,\,b}/_{b,\,a}]\!]$ means a and b are swapped.
- $P[\![^{b,\,c}/_{a,\,a}]\!]$ means a is mapped to both b and c.
- $P[\![^{a,\,b}/_{a,\,a}]\!]$ means a is mapped to both a and b.

The CSP$_M$ notation is essentially the same (though order of events reversed):

```
P[[b <- a]]  P[[b <- a, a <- b]]  P[[a <- b, b <-c]]
```


Beyond traces

We already know that traces give an incomplete description of processes.

Since $traces(P) = traces(P \sqcap STOP)$, it is clear we need a way of telling not only what events a process *can* do, but also what it can *refuse* to do.

A *refusal set* is a set of events that a process can *permanently* refuse.

$refusals(P)$ is the set of P's refusal sets after the empty trace.

Failures

A *failure* is a pair (s, X) where $s \in traces(P)$ and $X \in refusals(P/s)$.

Thus $\Sigma \in refusals(P)$ is and only if P can deadlock before communicating anything, and $(s, \Sigma) \in failures(P)$ if and only if P can deadlock after s.

Failures are the key to understanding nondeterminism.

Calculating failures

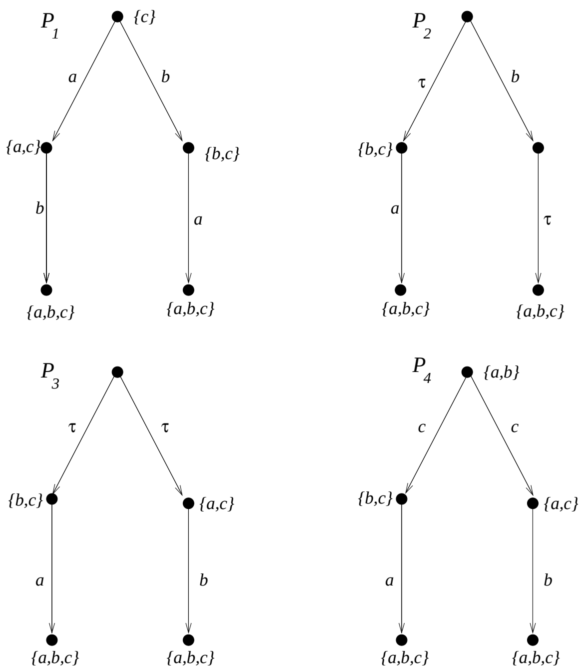
$failures(P)$ can be calculated inductively over the syntax of P (a rule for each operator), and details can be found in Chapter 8 of the book.

They can also be calculated from the transition diagram of P.

- This means you don't have to remember a series of semantic rules,
- and is basically how FDR does it.

Failures appear only at *stable* (τ-free) nodes of the graph, since we can't be sure of *permanent refusal* when a τ is available.

Failures from pictures



The results

Assuming $\Sigma = \{a, b, c\}$, each node is labelled with its maximal refusal.

Note that if a node can refuse X, then it can also refuse every subset of X.

- P_1 is $(a \to b \to STOP) \square (b \to a \to STOP)$ It has a *deterministic* transition system: no τ's, no ambiguous branching visible actions). Hence a unique path through the tree for any trace, so just one maximal refusal for each trace s: $\Sigma \setminus initials(P/s)$. Example failures: $(\langle\rangle, \{\})$, $(\langle\rangle, \{c\})$, $(\langle a\rangle, \{a, c\})$.
- P_2 shows how internal actions can introduce nondeterminism. It could have arisen as $((c \to a \to STOP) \square (b \to c \to STOP)) \setminus c$. Its failures are

$$\{(\langle\rangle, X) \mid X \subseteq \{b, c\}\} \cup$$
$$\{(\langle a\rangle, X), (\langle b\rangle, X) \mid X \subseteq \{a, b, c\}\}$$

More results

- P_3 could be $(a \to STOP) \sqcap (b \to STOP)$. It has two initial τ's to choose from. Its initial refusals are $\{X \mid \{a, b\} \not\subseteq X\}$. It can refuse either a or b separately but must accept something if $\{a, b\}$ is offered.
- P_4 which could be $(c \to a \to STOP) \square (c \to b \to STOP)$ shows how ambiguous branching on a visible action can lead to nondeterminism. Its refusals after the trace $\langle c\rangle$ are $\{X \mid \{a, b\} \not\subseteq X\}$.

Failures refinement

One process *failures*-refines another: $P \sqsubseteq_F Q$ if and only if

$$traces(P) \supseteq traces(Q) \quad \text{and} \quad failures(P) \supseteq failures(Q)$$

Q can neither accept an event nor refuse one unless P can.

$P_2 \sqsupseteq_F P_3$ and this is the only failures refinement among $P_1, \ldots, P_4$.

Failures and choice

We can now distinguish internal and external choice: consider

$$\begin{array}{rcl} Q_1 & = & (a \to STOP) \Box (b \to STOP) \\ Q_2 & = & (a \to STOP) \sqcap (b \to STOP) \end{array}$$

Q_2 can initially refuse $\{a\}$ and $\{b\}$, but Q_1 cannot. In general,

$$P \sqcap Q \sqsubseteq_F P \Box Q$$

Divergence

Failures refinement works well, but gives some strange answers on processes like $\mathbf{div} = (\mu\, p.a \to p) \setminus a$ that diverge.

This process is not doing anything useful, but it has no failures since there are no stable states. In fact, for all P,

$$P \sqsubseteq_F \mathbf{div}$$

This is just as unfortunate as the fact that $P \sqsubseteq_T STOP$.

A process which can choose not do behave in the ways we are recording but do something else instead will seem to refine ones that do the opposite.

The answer: build divergences into our model and record the set of traces on which P can diverge: $divergences(P)$.

Divergence as a black hole

The theory has difficulty handling potential distinctions in a process's behaviour after it might have diverged.

The *failures-divergences* model therefore takes the decision that any two processes that can diverge immediately are

- equivalent, and
- completely useless.

Specifically, we assume that if $s \in divergences(P)$ then $s\hat{\ }t \in divergences(P)$ (whether or not P can even perform $s\hat{\ }t$).

Also use extended sets of traces and failures:

$$\begin{array}{rcl} traces_\perp(P) & = & traces(P) \cup divergences(P) \\ failures_\perp(P) & = & failures(P) \cup \{(s, X) \mid s \in divergences(P)\} \end{array}$$

The failures-divergences model

Since every trace is followed by divergence or a stable state, there is no need to record them separately, so P is identified with

$$(failures_\perp(P), divergences(P))$$

This is the mathematical model for CSP most often used to define what it means for two processes to be equivalent.

Because of the closure under divergence, over this model *any* process that can diverge immediately (i.e., without any visible communication) is equivalent to $\mathbf{div}$, no matter what else it may also be able to do.

Failures-divergence refinement

One process *failures-divergences-refines* another, written $P \sqsubseteq_{FD} Q$, if and only if

$$\begin{array}{l} \mathit{failures}_\perp(P) \supseteq \mathit{failures}_\perp(Q) \\ \wedge\ \mathit{divergences}(P) \supseteq \mathit{divergences}(Q) \end{array}$$

The position of $\mathbf{div}$ under refinement has been reversed: $\mathbf{div} \sqsubseteq_{FD} P$ for all P.

FDR stands for **F**ailures-**D**ivergences **R**efinement.

Why this model?

Most correct programs are divergence free so

- we have to model divergence to know this is so, and
- what happens after divergence is usually of little practical interest.

If P has performed trace s and we know it can neither refuse X nor diverge after s, then if offered X it will certainly accept a member of X.

Thus this model allows us to constrain not only what events a process can perform, but also allows us to specify what it *must* accept.

Determinism

P is defined to be deterministic if, and only if, $\mathit{divergences}(P) = \{\}$ and

$$s\hat{\ }\langle a\rangle \in \mathit{traces}(P) \Rightarrow (s, \{a\}) \notin \mathit{failures}(P)$$

In other words, it cannot diverge, and never has the choice of both accepting and refusing any action.

Deterministic processes

- are predictable, and
- are precisely the most refined processes under $\sqsubseteq_{FD}$ (which can be read as 'is more nondeterministic than').

Failures-divergences specifications

We now have richer languages for specifying processes.

Failures and failures/divergences specifications can be presented as conditions on the individual behaviours or via characteristic processes and refinement.

$$\forall\, s.(s, \Sigma) \notin \mathit{failures}(P)$$

is the specification of deadlock freedom (over failures or failures-divergences), and it has characteristic process DF_Σ, where

$$DF_A \;=\; \sqcap\{a \to DF_A \mid a \in A\}$$

You can test if a process is deadlock free by failures(-divergence) refinement of DF_Σ. (Over failures-divergences you will also catch divergence.)

A more interesting example of a failures-divergences specification – being a buffer – will be studied later.