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 $\Sigma \in refusals(P)$ if 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

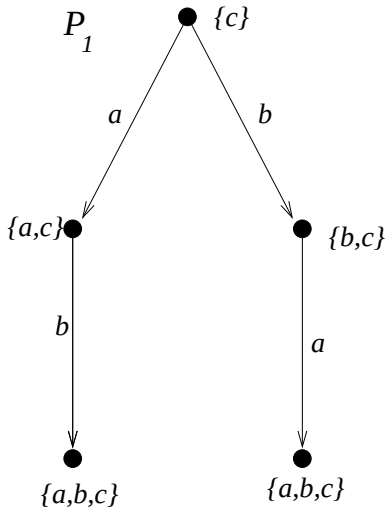
$failures(P)$ can be calculated inductively over the syntax of P (a rule for each operator), and details can be found in Chapter 10 of the book.

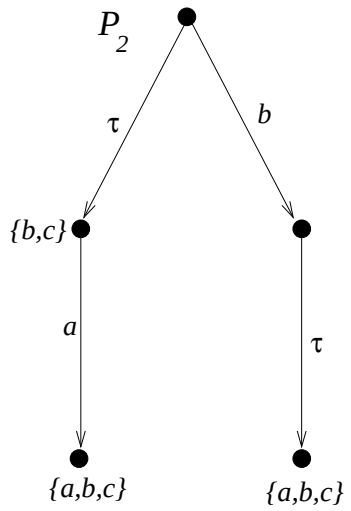
They can also be calculated from the transition diagram of P.

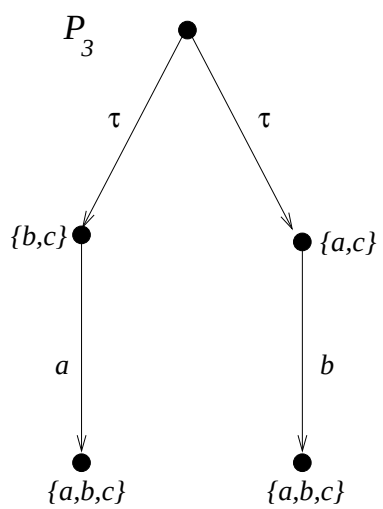
- This means you don't have to remember a series of semantic rules,
- and is basically how FDR does it.

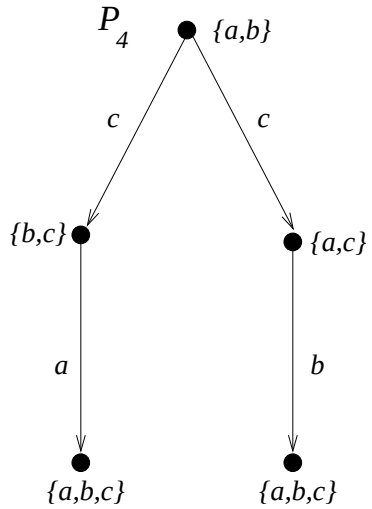
Failures appear only at *stable* (τ-free) nodes of the graph, since we can't be sure of *permanent refusal* when a τ is available.

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 $(a \to b \to STOP) \,\Box\, (b \to a \to STOP)$ 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 $((c \to a \to STOP) \,\Box\, (b \to c \to STOP)) \setminus c$. 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 $(c \to a \to STOP) \Box (c \to b \to STOP)$ 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

$$Q_1 = (a \rightarrow STOP) \,\Box\, (b \rightarrow STOP)$$

$$Q_2 = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$$

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 $\mathbf{div} = (\mu\, p.a \rightarrow p) \setminus a$ 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:

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

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

$$failures_{\perp}(P) \supseteq failures_{\perp}(Q)$$

$$\wedge\ divergences(P) \supseteq divergences(Q)$$

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, $divergences(P) = \{\}$ and

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

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.

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

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

$$DF_A = \prod\{a \to DF_A \mid a \in A\}$$

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

Example failures specifications

Refining...

- $DF_A \,|||\, Chaos_{\Sigma \setminus A}$ means P always offers a member of A
- $RUN_A \,|||\, Chaos_{\Sigma \setminus A}$ means P always offers all members of A
- If $P(a) = a \rightarrow P(a) \,\Box\, (STOP \sqcap ?x : A \setminus \{a\} \rightarrow P(x))$, refining $?x : A \rightarrow P(x)$ says that initially P will accept the whole alphabet A and then will never refuse the most recent communication.

Buffer specification: traces

The strongest trace specification of a buffer from channel $left$ to channel $right$ is

$$tr \subseteq \{|\ left, right\ |\}^* \wedge tr \downarrow right \leq tr \downarrow left$$

This allows some unlikely processes such as

- $STOP$
- $SINK = left?x \rightarrow SINK$
- $STOP \sqcap COPY$

There is nothing to force the process to input or output when it should.

Buffer specification: failures

We can add:

- $tr \downarrow left = tr \downarrow right \Rightarrow ref \cap \{|\ left\ |\} = \{\}$
  An empty buffer must accept any input.
- $tr \downarrow left > tr \downarrow right \Rightarrow \{|\ right\ |\} \not\subseteq ref$
  An non-empty buffer must allow some output.

Note the use of (tr, ref) to denote an arbitrary failure (extending earlier style for trace specifications). Note also the asymmetry between the requirements for an arbitrary input and merely some output.

Catches all the examples on the previous slide.....

Buffer specification: failures-divergences

... but the simply divergent process **div** satisfies it as a failures specification. So to get a buffer that must perform necessary inputs and outputs we must add the requirement that it is divergence free.

The most nondeterministic buffer is $Buff_{\langle\rangle}$

$$Buff_{\langle\rangle} = left?x : T \to Buff_{\langle x\rangle}$$

$$Buff_{s\hat{}\langle y\rangle} = (STOP \sqcap left?x : T \to Buff_{\langle x\rangle\hat{}s\hat{}\langle y\rangle}) \\ \Box\ right!y \to Buff_s$$

P is a buffer if and only if it refines $Buff_{\langle\rangle}$.

Note this is an infinite state process.

Buffer Laws

In TPC a series of laws relating buffers and $\gg$ are given. These can easily be translated to involve link parallel. The most obvious is

BL1. If P is an $(a \Rightarrow b)$-buffer, and Q is $(c \Rightarrow d)$-buffer, with $a \neq d$, then $P\,[b \leftrightarrow c]\,Q$ is a $(a \Rightarrow d)$-buffer.

The following law can be extremely useful for analytic proofs that combinations are buffers:

BL5. Suppose P uses events $\{|\ a, b\ |\}$ and Q uses $\{|\ c, d\ |\}$, with $a \neq d$. If x is free in neither P nor Q, which are such that

$$P\,[b \leftrightarrow c]\,Q \sqsupseteq_{FD} a?x \rightarrow (P\,[b \leftrightarrow c]\,d!x \rightarrow Q)$$

then $P\,[b \leftrightarrow c]\,Q$ is an $(a \Rightarrow d)$-buffer.

Example of BL5

Consider $S = left?x \rightarrow right!x \rightarrow right!x \rightarrow S$ and
$R = left?x \rightarrow left?y \rightarrow (right!x \rightarrow R \sqcap right!y \rightarrow R)$.

It is easy to show that

$$R \gg S = left?x \rightarrow (S \gg (right!x \rightarrow R) \sqcap (right!x \rightarrow R))$$
$$left?x \rightarrow (S \gg right!x \rightarrow R)$$

So $R \gg S$ is a buffer by BL5.

Divergence freedom rule

Both deadlock and divergence freedom have specific buttons on FDR, which uses refinement checks to verify them. But it is useful to understand ways of creating networks that are free of them by design. We studied this issue for deadlock in Chapter 4.

Order rule for divergence freedom Suppose we have a partial order on the nodes of a network where every pair of connected processes are comparable. Suppose all internal communication (i.e. all events in the alphabet of at least two processes) is hidden, and no other. Suppose further that each component process is divergence free when all communications with processes less than it in the partial order are hidden. Then the network is divergence free.

For example, any chain of processes where none can output infinitely without inputting infinitely also, is divergence free.

Abstraction

Suppose a divergence-free process P has two separate parts of its alphabet: say H and L, that are controlled by different users.

What does it look like to L when actions in H have been abstracted away.

It is tempting to think $P \setminus H$, but this is usually wrong since it assumes H never refuses a communication.

The right answer for most purposes, if we assume that H uses the CSP model of handshaken interaction, is the value of

$$(P \underset{H}{\parallel} Chaos_H) \setminus H$$

over stable failures (i.e. ignoring divergence).

This is called the lazy abstraction $\mathcal{L}_H(P)$. Lazy because H does not have to do anything.

Example

Consider the processes $P1 = a \to b \to P1$ and $P2 = a \to P2 \,\Box\, b \to P2$:

- $\mathcal{L}_{\{a\}}(P1) = Chaos_{\{b\}}$
- $\mathcal{L}_{\{a\}}(P2) = \mu\, p.b \to p$

Abstracting errors

Recall the controlled error model, in which errors are generated by some demon that has special events to trigger them.

We should look at such systems with these error events abstracted, perhaps after being limited by parallel composition: $FAULTY$ is

$$\mathcal{L}_E(P) \quad \text{or} \quad \mathcal{L}_E(P \underset{X \cup E}{\|} L)$$

where L is a process that never refuses a member of X.

We get a natural specification of fault tolerance:

$$P \underset{E}{\|} STOP \sqsubseteq_F FAULTY$$

In other words, the system behaves no worse with faults permitted than it does with them banned.

Abstraction and security

Suppose that L and H are low and high security level users of the system P. Under what circumstances can we be sure that L can gain no information about H's behaviour from her interaction with P?

When $\mathcal{L}_H(P)$ is deterministic!

This is the lazy independence definition of security and is certainly accurate when P itself is deterministic.

Recall $\mathcal{L}_{\{a\}}(\mu\, p.a \rightarrow b \rightarrow p)$ is nondeterministic: a user who controls as can determine how many bs are available.

$\mathcal{L}_{\{a\}}(\mu\, p.(a \rightarrow p \,\Box\, b \rightarrow p))$ is deterministic: nothing can stop an unbounded sequence of bs being possible.

Covert channels

A covert channel is a way, unintended by the system designer, by which H can pass information to L. Therefore lazy independence is a technique for identifying covert channels or proving them absent.

Typically these arise because of contention for resources: communication, memory, file, CPU etc.

They can be eliminated by (i) giving priority over resources to L (not always attractive) (ii) strictly partitioning resources (potentially wasteful) and sometimes compromises between the two.

An example involving a shared communication resource is illustrated in an example file.

Security

This is just one of many ways in which CSP/FDR can be used to analyse security: see Chapter 15 of TPC, and Modelling and Analysis of Security Protocols: the CSP Approach (Addison Wesley).

The absence of information flow (what lazy independence attempts to capture) is a large and fascinating theoretical topic, with many different interpretations!