Basic operation

Understanding how FDR (like probably every tool) works will help you use it effectively. So here we try to explain it!

Just about everything it does is based on running the operational semantics of CSP. These provide recipes for calculating a *labelled transition system* (LTS) giving a picture of how each process can run.

An LTS is a set of nodes linked by actions from $\Sigma^{\checkmark,\tau}$, where τ represents an internal, invisible, uncontrollable action the process can choose to do itself.

The formal operational semantics treats (closed) CSP terms as the nodes of a big LTS, giving a recipe for calculating

- the initial actions of each node, and
- the possible result(s) of performing each action (each result also being a closed CSP term).

Note that it is the actions that determine how an LTS behaves: the structure of the nodes themselves is irrelevant.

Inferring actions

The most general formulation derives these actions via a natural deduction style inference system, for example:

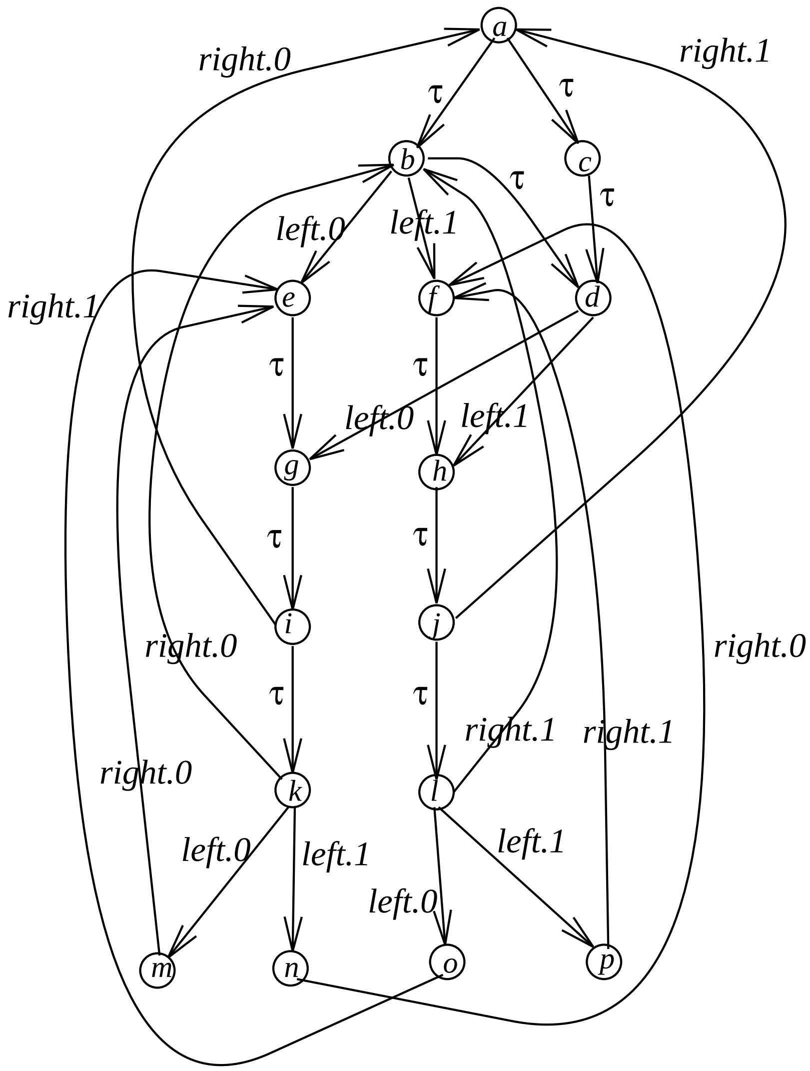
$$\frac{P \xrightarrow{\tau} P'}{P \mathbin{\Box} Q \xrightarrow{\tau} P' \mathbin{\Box} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \mathbin{\Box} Q \xrightarrow{\tau} P \mathbin{\Box} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \mathbin{\Box} Q \xrightarrow{a} P'}(a \neq \tau) \qquad \frac{Q \xrightarrow{a} Q'}{P \mathbin{\Box} Q \xrightarrow{a} Q'}(a \neq \tau)$$

$$\frac{}{\mu\, p.P \xrightarrow{\tau} P[\mu\, p.P/p]}$$

Example

Even a simple process like $COPY \gg COPY$ can give a machine that is complex to look at!



Making it efficient

It is relatively inefficient (both in space and time) to perform these inferences literally on symbolic representations of processes. (We want to handle multi-million state systems!)

But there seems no real option for the most general syntax of CSP, in particular when recursion is involved.

FDR therefore uses a two-level approach to calculating operational semantics:

- *Low level* "compilation" is fully general (using the operational semantics in just about their original form). In a typical system FDR will identify a number of component processes and compile these to pure state machines.
- *High level* "running" is based on calculating a set of rules for turning (usually parallel) combinations of state machines into a single one.

These phases have very different appearances when performing a check!

The dividing line

The dividing line between high and low level is determined automatically, approximately as follows:

- All true recursions are low level. A true recursion is a process name (possibly parameterised) whose value depends (possibly through other recursive expansions) on itself (where parameterised, with *the same* arguments). Thus the recursion

  ```
  P(n) = a -> P(if n==0 then N else n-1)
  ```

  is low level (for $0 \leq n \leq N$) but

  ```
  Q(n) = a -> if n==0 then STOP  else Q(n-1)
  ```

  is not.
- Subject to the above over-riding principle, as many 'high-level' operators (all parallel operators, renaming and hiding) as possible are high-level, but any other operators are treated at low level unless they are forced to high level by being outside one of these.

Example

Thus in the unlikely combination

```
LevelExample = a -> ((b -> STOP)
                     |~|
                     ((c -> STOP) ||| Q))
```

(where the identifier `LevelExample` does not appear in `Q`) the processes `b -> STOP` and `c -> STOP` are both compiled at low level, while the internal choice `|~|` and the `a ->` construct are both high level because they are outside the `|||`.

For most practical examples this approach produces just one high-level structure, for example any combination of low-level processes using only hiding, renaming and parallel.

However, high-level prefix choice, sequencing, etc., can create multiple ones:

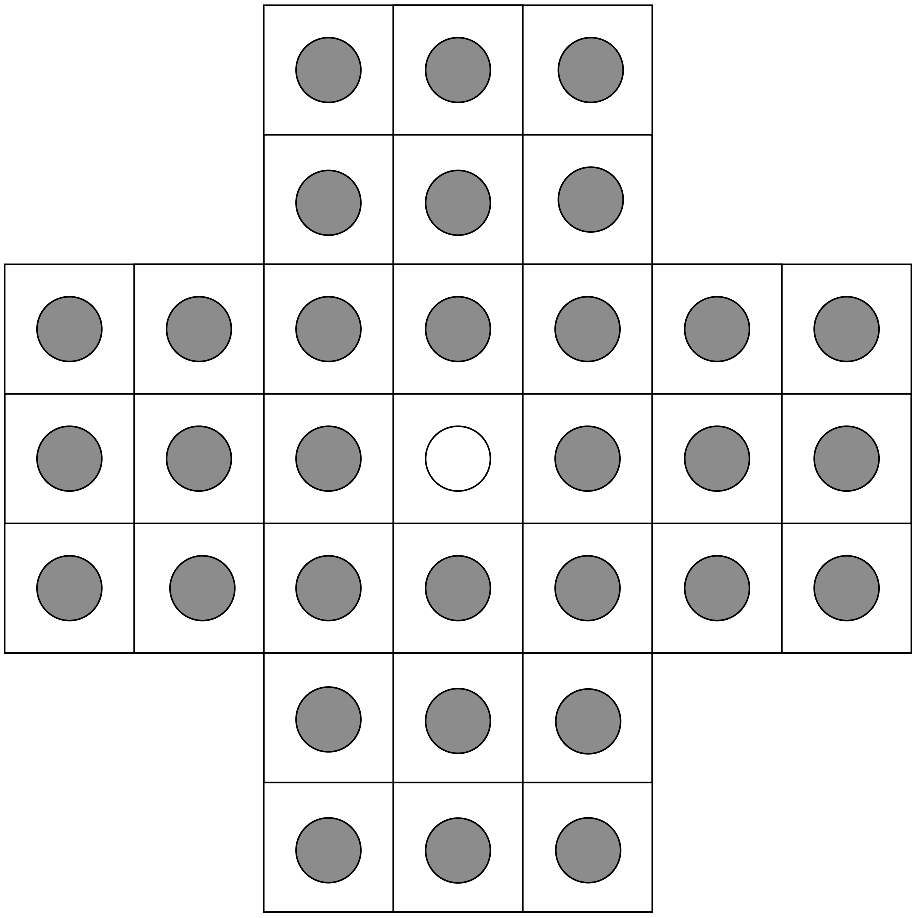
```
c?x -> (P(x) ||| Q(x))
```

gives one *configuration* for each `x`, plus an initial one.

For simplicity, we will concentrate on the one-configuration case.

Solitaire: one configuration, many states

Recall our model of a solitaire board as a parallel system of 33 slot processes plus STOP:



Every state has the same parallel structure, it is only the component states that change.

Combining actions

Any action of a configuration will come from one of a fixed set of rules for combining actions of subprocesses.

When running a high-level process FDR computes actions by seeing which rules are enabled (because all the required actions are possible).

The result state has all components the same as before, *except* those that took part in the action.

For solitaire we would have:

- A rule (each) to promote a τ action from any given component (rules that will never be needed in this case).
- A rule for each move, involving three the three processes with it in their alphabets: the combination can do the event if they all can.
- A rule for `done`, involving *all* the slots.

Hiding, renaming, and interleaving can create more complex rules, but on the same principle!

Supercombinators

FDR calls these rules *supercombinators*: they are automatically generated rules for complex combinations of processes which mimic the standard operational semantic rules such as

$$\frac{P \xrightarrow{\tau} P'}{P \underset{X}{\|} Q \xrightarrow{\tau} P' \underset{X}{\|} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \underset{X}{\|} Q \xrightarrow{\tau} P \underset{X}{\|} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P[\![R]\!] \xrightarrow{b} P'[\![R]\!]} \quad (a\,R\,b)$$

which form the operational semantics.

The difference with simply applying the basic rules hierarchically is efficiency: speed and compactness – the rules are stored so they can be computed by bit manipulation.

You will see the difference if you turn "supercompilation" off!

State storage

During a run using supercombinators, each state is stored as an indication of which configuration it is in, plus a packed bit-representation of the component states ($\lceil \log_2 N \rceil$ bits for an N-state component). Thus each state of solitaire requires $2 = \lceil 33/32 \rceil$ words for each state (plus perhaps a little housekeeping), as all the slot processes have 2 states.

It is possible (though not yet implemented in FDR2) to get this down further by data-compressing the sorted lists of states FDR uses.

Explicit and implicit state machines

The initial state of a supercombinator machine, plus the rules, gives an *implicit* representation of a state machine.

It stays (somewhat) implicit even when all the states have been computed, because we don't remember the transitions and may have to do some work to find whether and where a state is stored.

An *explicit* state machine is a list of states, each with all its actions and pointers to next states. Once you have *explicated* a machine there is no need to remember the states' structures: we can simply label them $1, \ldots, M$.

Explicit machines are faster, but usually much larger, and of course are slower to produce: *there is no point in explicating a machine if all you have to do is visit each state once.*

The compiler produces explicit machines, and there are other functions that produce them, including all the *compression* functions.

Pragmatics of running

Bear in mind that

- High-level running is much more efficient in time and space, and can deal with much larger machines.
- FDR is optimised for dealing with systems with relatively few configurations, and relatively simple (< 4000 state, say) low-level components.
- Low-level components are completely compiled, even when some of the states will never be visited in the eventual run. *A very common error is including infinite-state components via, for example, an integer or sequence parameter.*

The need for normalisation

FDR's main role is deciding refinement between processes.

We now know how it compiles them into transition systems.

There is no obvious way of deciding whether one transition system refines another over any of $\mathcal{T}$, $\mathcal{F}$ or $\mathcal{N}$, since the relationships can be very subtle:

We have to decide whether,

- for each state σ of the implementation, then
- for every trace t that the implementation can be on in σ,
- then, for each observable behaviour b of σ (in the appropriate model) there is some state ρ that the specification can be in on t that allows b.

If dealing with arbitrary machines, this is very complex. Life would be much simpler if the specification could only be in one state for each trace. FDR achieves this by *normalising* the specification.

Normal form state machines have a unique state for each trace: once we are told the present trace, there is only one state it can be in....

Normal form state machines

In order to achieve our aims, these must have

- No τ actions, and
- No ambiguous branching on any visible action.

For either of these would lead to ambiguity over what state it is in after at least one trace.

If these were ordinary transition systems, they could neither diverge nor show any nondeterminism.......which is not reasonable since we want one equivalent to an arbitrary (specification) process.

Therefore (where needed for the chosen model) we allow nodes to be annotated with divergence information and minimal acceptance sets.

Such annotations turn an LTS into a Generalised *LTS (GLTS). These are also produced by some of FDR's compression operators.*

Normalisation

Stage 1 Given a finite labelled transition system $L = (V, E, v_0)$, we form a graph $\mathcal{P}_L$ whose nodes are members of $\mathbb{P}(V)$ as follows

- The initial node is $\tau^*(v_0)$, where $\tau^*(v)$ is defined to be $\{w \mid v \overset{\langle\rangle}{\Longrightarrow} w\}$, the nodes reachable under some sequence of τ's from v_0.
- For each node generated we have (if using the failures/divergences model $\mathcal{N}$) to decide whether it is divergent: this is true if and only if it contains a cycle under $\xrightarrow{\tau}$.
- If a node N is not divergent, we determine the set of non-τ actions possible for $v \in N$. For each such action a we form a new node, the set $\bigcup\{\tau^*(w) \mid \exists\, v \in N . v \xrightarrow{a} w\}$, the set of all nodes reachable after action a and any number of τ's from members of N.
- The search is completed when all 'new' nodes generated are ones that have been previously expanded.

This is called the *pre-normal form*.

The pre-normal form

Because the nodes of the pre-normal form are sets of nodes of the original, the procedure on the last slide is, *in the worst case*, exponential in the size of the original.

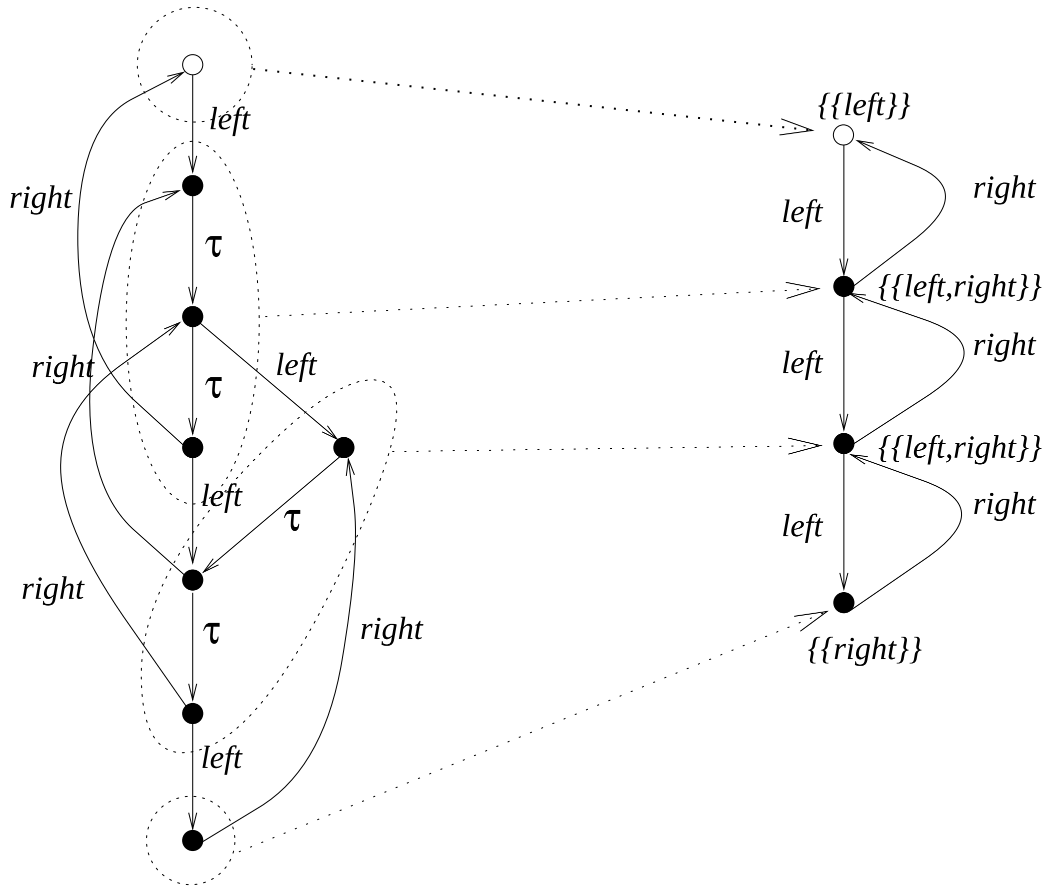
Fortunately the worst case is rare, *especially in processes designed to be specifications*, but can happen. See the book for an example.

More usually the pre-normal form is about the same size or smaller than the original. (It is perhaps the most-used *compression* function.)

The minimal acceptances of a pre-normal form node are the minimal ones from its members (as a set of nodes of L).

The pre-normal form has the properties required for refinement checking. However it may have a number of behaviourally equivalent nodes: these are factored out in **Stage 2** of the normalisation process by calculcating the maximal *bisimulation* that respects the node labellings.

Example: 3 data-less $COPY$'s chained



In this case the pre-normal form nodes are all semantically distinct, and so the final bisimulation makes no further identification.

Checking refinement

Once the specification end of a refinement check has been normalised, the following are necessary to establish failures/divergence refinement.

- Establish which states of the implementation transition system are divergent, marking them as such.
- Model-check the implementation, thus marked, against the normal form.

FDR presently interleaves these phases.

For any given node P, the first of these is done by a depth-first search (DFS) algorithm over $\tau^*(P)$: it can diverge if and only if the search ever encounters the spine of partially-explored nodes.

The model-checking phase does a breadth-first search (BFS) over the pairs of (normalised) specification and implementation nodes which have a trace in common. All relevant behaviours (initials, refusals, divergence) of the implementation node must be allowed by the specification.

Checking refinement II

If all pairs found are OK, refinement holds.

If one is found that does not, refinement fails and a counter-example is produced. (BFS guarantees a shortest.)

Note that one *may* explore more pairs of nodes than there are nodes in the implementation, because some may correspond to more than one specification node. *This is relatively rare.*

The BFS of the model-checking phase can be done more efficiently than the DFS of divergence-checking *especially for very large systems.* See the book for details – the difference can make itself felt in the difference between `[FD=` and `[F=`.

Why compress?

Many or most variable-sized networks one can analyse on FDR have state-spaces that grow exponentially with the number of components.

Such systems quickly reach limits beyond which it is unattractive or impossible to go beyond if using an *explicit model-checking* algorithm (one that visits each state separately).

These limits may or may not be smaller than what represents an *"interesting"* system. They are almost always frustrating.

Implicit model checking

Implicit model checking still covers all states, but only nominally. Somehow we prove that all are OK without visiting each separately. A variety of approaches have been developed:

- BDD's (Binary decision diagrams) turn transition relations and satisfaction of specifications into propositional formulae with a clever representation. *This is the most different, conceptually, from exploring states one at a time.* FDR doesn't presently incorporate this, though there are several CSP tools that do.
- Partial order methods use arguments to avoid visiting all states. *There is one important application of this to CSP: the* `chase` *function (see Section 15.3) but the nature of CSP models and refinement restricts them.*

Implicit model checking II

- Hierarchical compression: build up a process in steps, replacing appropriate components with semantically equivalent ones with smaller state-spaces. *This is the main method employed in FDR2.*
- In appropriate examples: data independence, symmetry exploitation and symbolic operational semantics.

Note that the first and last are both called "symbolic model checking" in literature, though they have nothing really in common.

Unlike explicit checking, which is by-and-large predictable and reliable within its limits, all of these are, comparatively, "black arts". They can sometimes produce spectacular results, but equally can be disappointing.

The principles of compression

You can calculate the value of any CSP combination

$$C[P_1, \ldots, P_n]$$

in each of the models $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{N}$ from the semantic values of P_1,. . .,P_n and the structure of $C[\cdot]$.

Thanks to the congruence theorem between operational and denotational semantics, the semantic values calculate operationally from the context depend only on the semantic values of the P_i.

Thus you can replace the operational values of the P_i with any other LTS's with the same denoational values without changing the overall denotational value.

If, for example, P and Q are processes with 1,000 states (each) that can be compressed to 100 (each), this may have reduced the states of the complete system from 10^6 to 10^4.

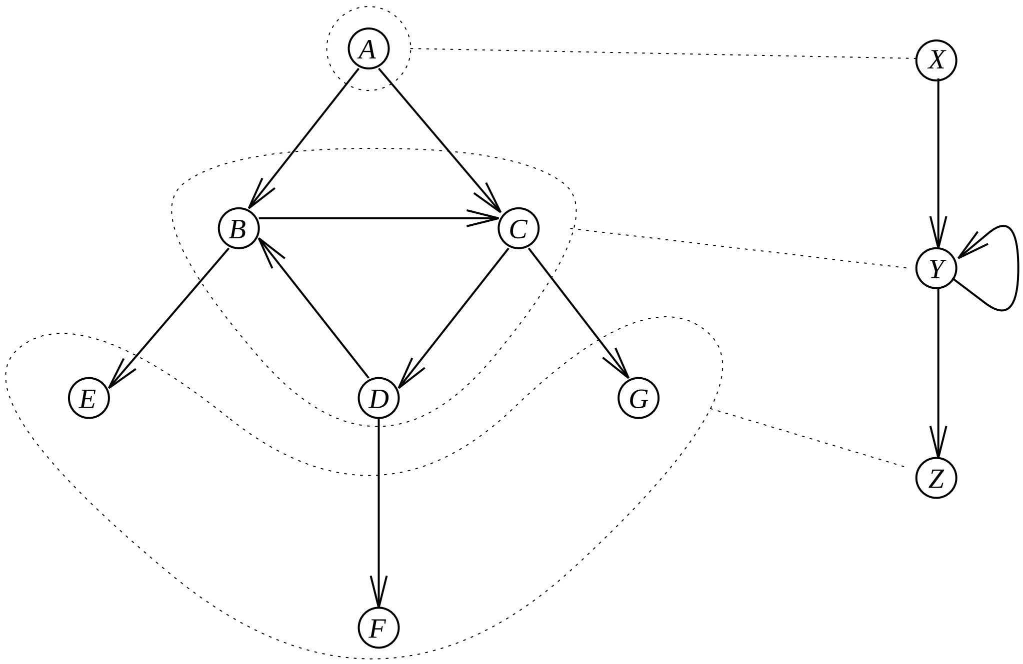
Methods of compression

At the time of writing, FDR2 uses five different methods of taking one LTS and attempting to compress it into a smaller one. Each of these is called by simply applying a function to the process you want to compress. Since these do not change the semantic value of the process they are applied to, FDR terms them `transparent` functions. Each one you want to use has to be declared as such.

- Normalisation (`normal(P)`) is as discussed earlier. *This is often the best, but can be expensive and can expand. Probably the method you try first on processes you would expect to be reasonably sized (*$< 10,000$ *states, say) and without much externally visible nondeterminism.*

Bisimulation

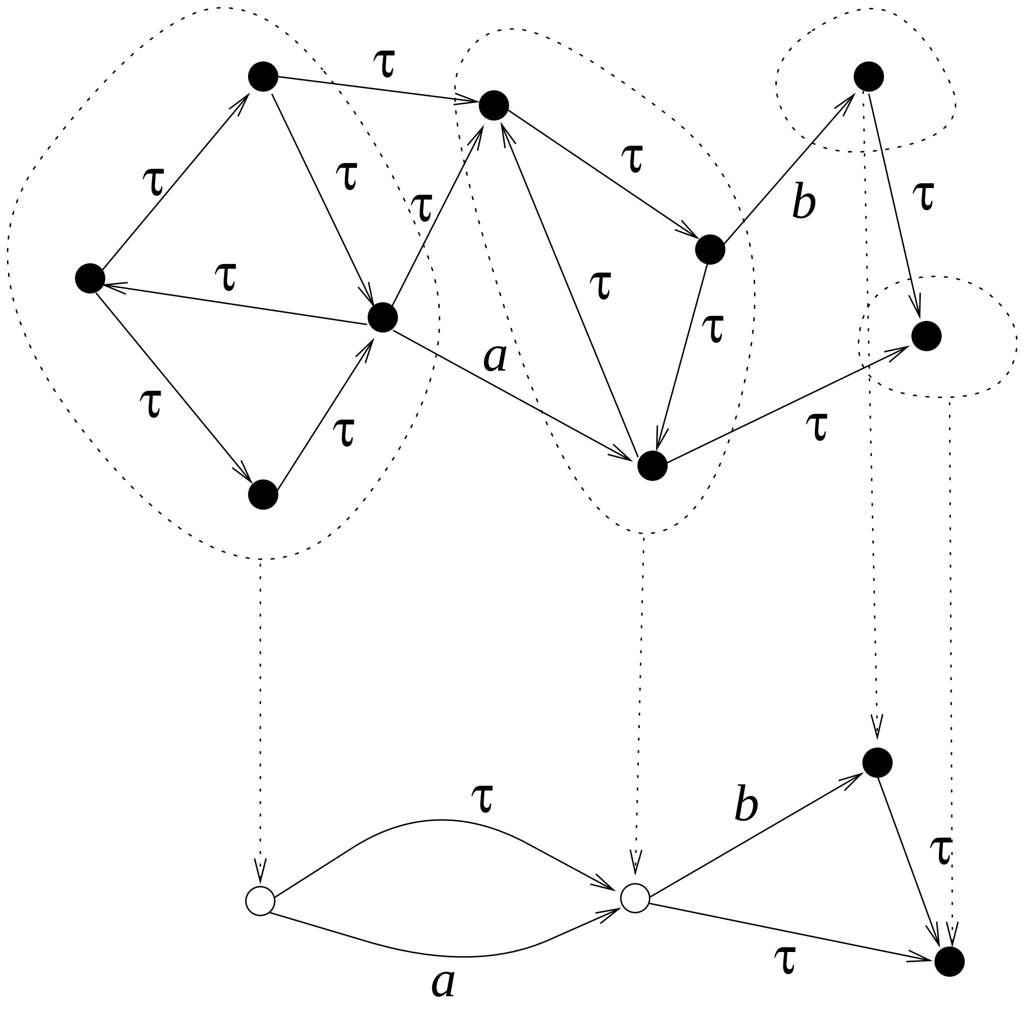
- Strong bisimulation (`sbisim(P)`): two nodes are equivalent iff they have indistinguishable patterns of actions (including τ's) in the strongest sense.
  FDR performs this by computing a series of more and more refined relations until a fixed point is reached (see book: Chapter 7 and Appendix A).



τ-loop elimination

- τ-loop elimination (`tau_loop_factor(P)`): since a process may choose automatically to follow a τ-action, it follows that all the processes on a τ-loop (or, more properly, a strongly connected component under τ-reachability) are equivalent. Therefore all the nodes on such a component can be identified. As discussed in the previous section, this calculation is done via depth-first search algorithms.
  Since this can only compress divergent processes, its stand-alone uses are necessarily rather specialised. It exists mainly as a precursor to `diamond`, described below.

Picture of τ-loop elimination

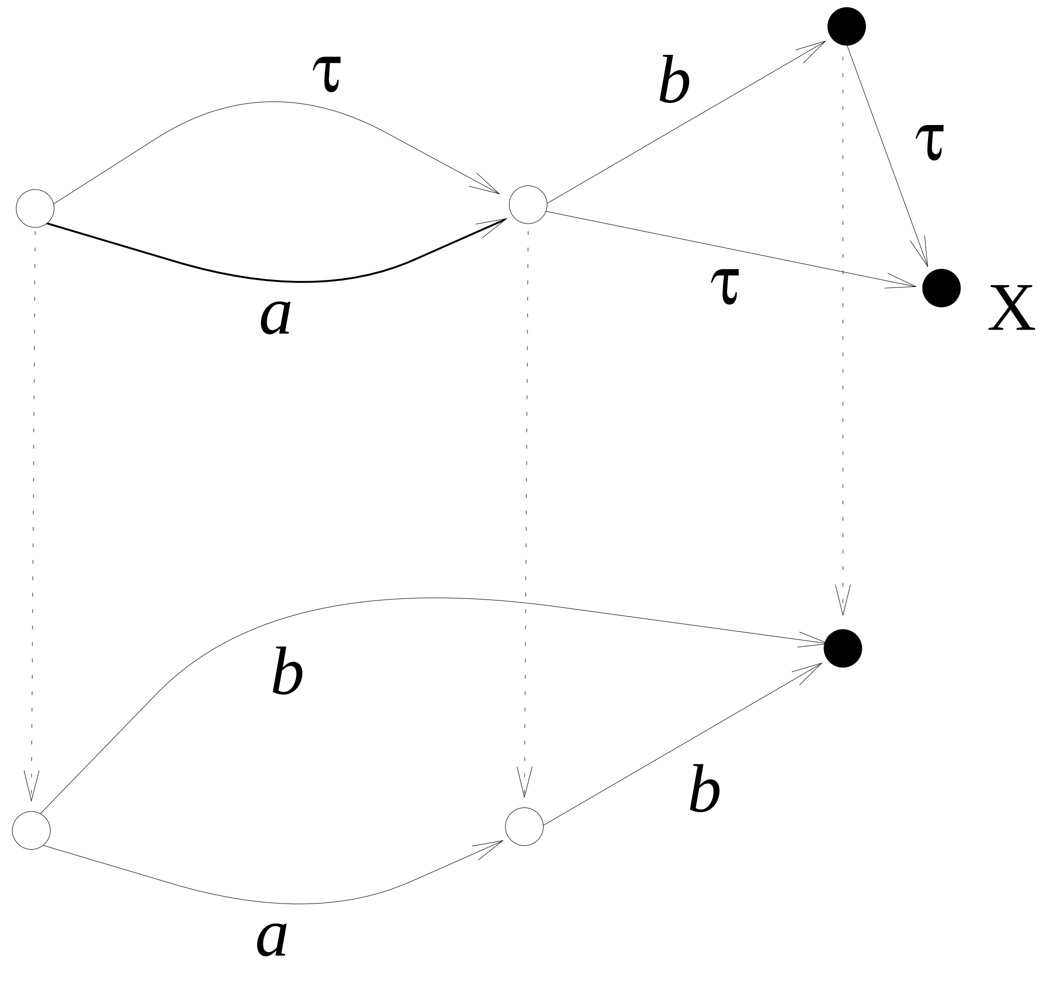


○ Node marked as divergent

Diamond

- Diamond elimination (`diamond(P)`): this carries out a transformation which cuts out all τ actions from an LTS. It does not, however, disambiguate visible actions.
  It applies τ-loop elimination as its first step, and then essentially creates a system with each node equivalent to a $\tau^*(P)$ for some node of the original.
  May be better or worse than normalisation *though on simple examples is often identical*. It can only give compression on systems with τ actions.

Picture of diamond compression



All three nodes get refusal from X

Semantic factoring

- Factoring by semantic equivalence (`model_compress`): the compositional models of CSP we are using all represent much weaker congruences than bisimulation. Therefore if we can afford to compute the semantic equivalence relation over states it will at least as good a compression as bisimulation to factor by this equivalence relation. The algorithm used to compute these equivalences is based on an extension to normalisation. *Significantly slower than bisimulation, though often giving the same results. Only recommended for small processes.*

Some of these functions will return a *Generalised* LTS, an LTS where nodes may be marked with divergence or minimal acceptance information. In fact, therefore, the whole of FDR is programmed to operate with these GLTS's rather than the basic sort.

Pragmatics

I usually try

- `normal(P)`
- `sbisim(diamond(P))`

and use a compression function `compress(P)` that can be easily re-defined.

Note also the `transparent` function `explicate` which turns any process into an *explicit* state machine (list of states and transitions). It is called automatically by the other compression functions.

Using compression

There is no point (except for exploiting other properties of the functions used) in compressing the top-level process which is to be explored in a check.

For compressing a process will take longer than checking it.

Rather, you should choose an appropriate set of levels as you build it up: you can compress at one level or at many.

Broadly speaking it is a good idea to apply compressions after hiding (because τ actions generally give them more scope).

Manipulate hiding to get it at as low a level as possible (i.e., don't do it all at the outside). It's usually better to do this with generic compression functions (see later).

Restructure your network so you can hide actions early, i.e., so that processes that communicate are combined early: *don't* put all philosophers and forks in parallel separately; *do* put them together alternately as a chain.

Be prepared (i) to experiment with different options and (ii) for surprises (both good and bad).

Contrary indications

Compression is less likely to work well:

(a) When you have good reason to believe that the differences between states will still be visible even after hiding. An example: attempting hiding on a standard solitaire board is pointless, though on a long thin one it might not be.

(b) It is less likely to work well if the hidden actions are resolving nondeterminism rather than representing "progress" (but by no means hopeless). process, as opposed to the resolution of nondeterminism.

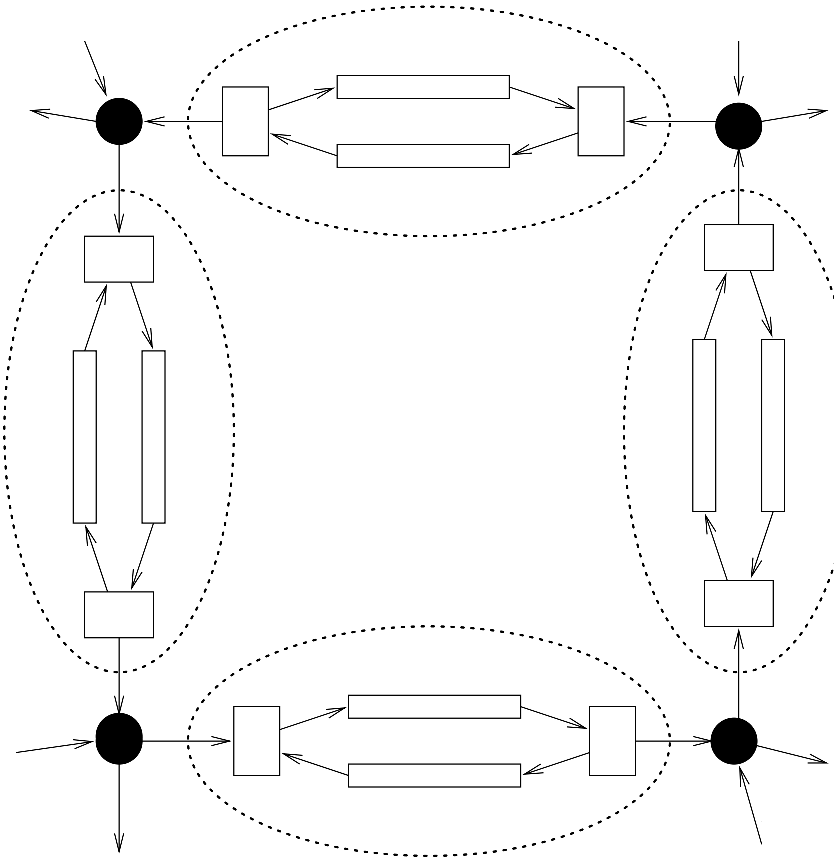
(c) If a large part of the state-space of the process you are compressing is never visited when it is put into the complete system. (The danger here is that you will expend too much effort compressing irrelevant states, or that the irrelevant states will make the subsystems too large to compress.)

Modular compression

Compression is likely to work well if some subcomponent of a network is relatively self-contained: complex internal behaviour, but with relatively simple external functionality.

Imagine, for example, a communication service which uses a number of copies of the alternating bit protocol to overcome an error-prone medium. We might have a ring of nodes N_i ($i \in \{0, \ldots, K-1\}$) with a channel between N_i and $N_{i \oplus 1}$ being implemented by an ABP. There would thus be K copies of this process.

Picture of modular compression



This *modular* compression strategy is particularly close to compositional development (where a complex process is replaced by a simpler specification).

Inductive compression

This strategy recompresses each time a new process (or a defined set of processes) is added.

It is likely to work well on networks that can be built up as chains or near-chains.

Best implemented automatically as a high-level function, on a list of process/alphabet pairs. This can also push hiding as far into a network as possible: *see and use* `compression.csp`*!*

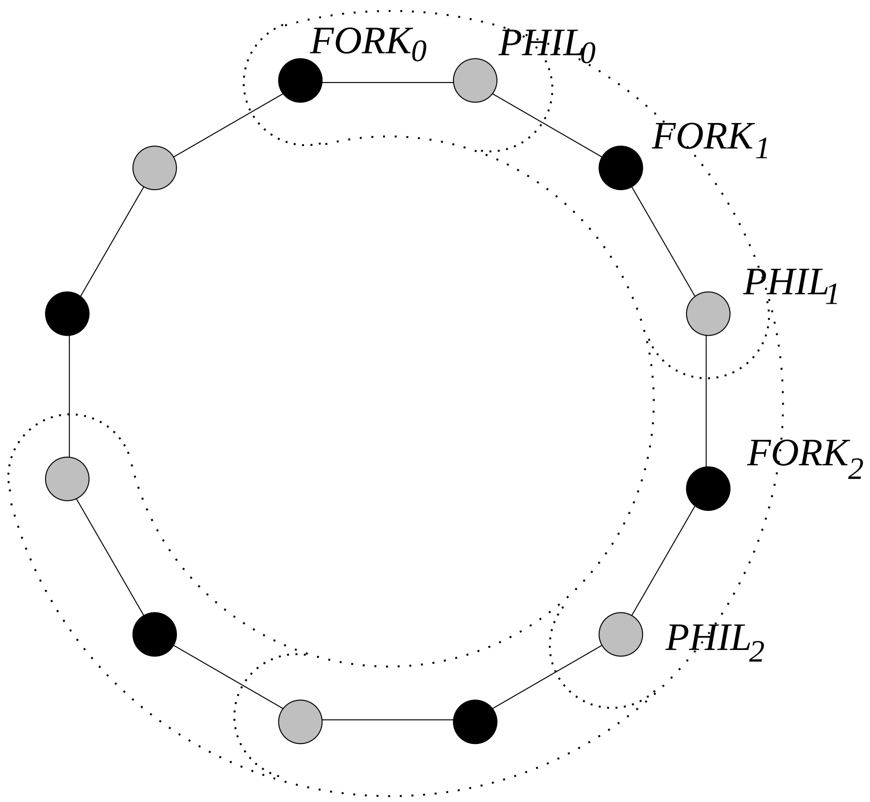
For example consider the dining philosophers: we can check for deadlock by hiding the entire alphabet

$$(\|_{i=0}^{K-1}(PandF_i, AP_i)) \setminus \Sigma$$

where $PandF_i$ is the combination

$$PHIL_i \underset{AF_i}{\|} FORK_i$$

Compressed dining philosophers



The unhidden actions of each combination are just the four *pickup* and *putdown* events from the two ends of a chain.

Each partial system has just four states; so you can check as many philosophers as you like!

Milner's scheduler

A 'scheduler' has the job of arranging two classes of event: $a.i$ and $b.i$ for $i \in \{0, \ldots, N-1\}$. It lets the $a.i$ happen in strict rotation and (for every i) insists on precisely one $b.i$ occurring between each pair of $a.i$'s.

It is a ring of simple cell processes connected using an extra, internal channel c:

$$\begin{array}{rcl} \mathit{Cell}(0) & = & a.0 \to c.1 \to b.0 \to c.0 \to \mathit{Cell}(0) \\ \mathit{Cell}(i) & = & c.i \to a.i \to c.i \oplus 1 \to b.i \to \mathit{Cell}(i) \quad \text{if } i > 0 \end{array}$$

The different case for 0 reflects that the rotation of the $a.i$ starts with $a.0$.

The alphabet AC_i is just the four events $\mathit{Cell}(i)$ uses, in the network description

$$\mathit{Scheduler} = (\|_{i=0}^{N-1} (\mathit{Cell}(i), AC_i)) \setminus \{| \; c \; |\}$$

The uncompressed network has $N \times 2^N$ states.

You can prove this deadlock-free using the same inductive strategy using $O(N^2)$ time.

Leaf compression

To prove that the $a.i$ rotate we use a trace specification with alphabet $\{\mid a \mid\}$: $ASpec(0)$ where

$$ASpec(n) = a.n \rightarrow ASpec(n \oplus 1)$$

leading to the refinement check

$$ASpec(0) \sqsubseteq_T Scheduler \setminus \{\mid b \mid\}$$

In this case, the inductive strategy fails badly with the state spaces of the partially constructed networks growing exponentially. (For reasons, see the book.)

Fortunately, a simpler approach, which we term *leaf compression* works very well. The only compressions are applied to the individual components of the network after distributing the abstraction of irrelevant events.

$$ASpec(0) \sqsubseteq_T (\|_{i=0}^{N-1}(normal(Cell(i) \setminus \{b.i\}), AC_i)) \setminus \{\mid c \mid\}$$

This gives a linear time check (with $2N$ states), and the same approach works better than inductive compression for deadlock freedom, since it is now a linear time check (with N states).

Further specifications of the scheduler

A specification involving *all or most* of the events $b.i$ will not compress well: if we leave all these events visible, we leave too many different states visible.

However a check involving a few will yield to compression just as well as the earlier ones: for example comparing the number of $b.n$'s and $b.m$'s.

We can learn important lessons from the scheduler:

The amount of abstraction you can do in a check may well have a huge effect on the success or otherwise of compression.

Leaf compression is a good, safe strategy to employ when you are able to abstract away any of the events which are in a single process' alphabet.

If you try it you will discover that bisimulation is not a useful compression for exploiting τ actions: it gets no real compression at all of any of these examples.