Model-Checking CSP

A.W. Roscoe*

October 5, 1995

1 Introduction

It is both inspiring and frustrating to work as closely with Tony Hoare as I have had the privilege to do over the past fifteen years. The source of the inspiration is obvious to all. The frustration comes from his ability to get things so right at the outset that the academic's usual meat and drink – refining and changing an idea, and exploring varieties of theories, until a consensus emerges – is often missing.

I believe this is true of the invention of his that I have spent longest working on, namely CSP [1]. The elegance and expressive power of the notation, and the simplicity of the standard semantic models, have the consequence that exploring other possibilities for these has been far less popular – and fruitful – than with other process algebras. Instead, theoretical work on CSP has concentrated on clear extensions, such as the inclusion of unbounded nondeterminism [2, 3], real time [4], probability [5] and similar. The basic ideas and notation have proved remarkably robust in these contexts.

One of the most stringent tests of a notation or idea is how it stands up to uses which the original designer never imagined. In this paper I describe work carried out over the last two years in building and using FDR[1], a model-checker/refinement-checker for CSP. Both CSP and its theories prove remarkably well-suited for this. Hoare's decisions which have proved helpful include

- basing the semantics, and equivalence, around the idea of refinement;
- separating the ideas of parallel composition and hiding, so that multiple processes can synchronise on events and enforce constraints, and hiding can be used as abstraction;
- the inclusion of a wide range of operators, both ones representing real modes of constructing processes and ones which, while pointless or difficult to implement, are useful in building specifications.

These have meant that it has been possible to build a fast refinement checker which can be used for the great majority of correctness proofs one is likely to want, and that the language is able to represent complex systems succinctly and clearly.

*Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

[1] FDR (Failures-Divergence Refinement) is a product of Formal Systems (Europe) Ltd.

2 Understanding refinement

The greatest stylistic difference between CSP and most other process algebras, at least to a theoretician, is Hoare's decision to base its semantics on mathematical models remote from the language itself. In CSP, a process is identified with its set of possible behaviours chosen from some class, whereas other process *algebras* have either been exactly that – with their notion of equality defined by some set of algebraic laws – or based on operational semantics.

The model-based approach has a considerable influence on the design of a language, primarily because we can only use operators with respect to which the chosen model is a congruence. The clearest consequence of this in CSP is in its choice constructs: since internal actions (τ) are invisible to the model, they cannot, for example, be allowed to resolve alternatives. Thus, CSP has a pair of choice constructs: $\Box$ which is only resolved by external events and $\sqcap$ where the choice is arbitrary, playing the role of the single operator $+$ (afforced by τ) in CCS.

Arguably the greatest difference appears in the treatment of recursion, since we are obliged to have a fixed-point theory over the abstract model which corresponds to our understanding of how a recursively-defined process is meant to behave[2]. This contrasts with an operationally-based theory, for example, where recursion is trivial to define using an unwinding/re-writing rule. Getting recursion to 'work' in an abstract theory is often the hardest thing to get right and a place where plausible-seeming congruences break up. It is largely the problems of recursion that led to the severe treatment of divergence in CSP.

The model-based approach thus imposes additional constraints. Whether the effect of these is positive or negative depends on one's viewpoint. One direct benefit is that models of the sort described above automatically supply a theory of *refinement*. Given that each process is identified with the set of all behaviours that it is allowed to display, it is natural to say that reducing this set corresponds to refinement. The fact that the whole language is compositional, and almost certainly monotone, with respect to the behavioural model means that the refinement order has the properties one might expect.

The standard model for (untimed) CSP is the *failures/divergences* model, where each process is represented by two sets of behaviours:

- *failures* are pairs (s, X) where s is a finite trace of the process and X is a set of events it can refuse after s;
- *divergences* are finite traces on which the process can perform an infinite sequence of consecutive internal actions.

For various technical reasons it is more difficult to model accurately the behaviour of a process after the possibility of divergence than on traces where no such possibility has arisen. Therefore, after each minimal (under the prefix order) divergence trace, the model identifies a process with $\perp$, the most nondeterministic process. The clearest way to understand accurately what the model represents is that

[2]The correctness of the representation of recursion or any other language construct is usually judged via congruence with an operational semantics. Since the abstract model is based on the set of all possible behaviours of some type(s) of a process, the 'real' value of a process can be extracted by seeing which behaviours are possible for its operational semantics. This generates a natural *abstraction* map Φ from the operational model to the behavioural model. An abstract operator op and the corresponding concrete/operational version $\mathbf{op}$ are congruent if, for all operational processes $\mathbf{P}$, we have $\Phi(\mathbf{op}(\mathbf{P})) = op(\Phi(\mathbf{P}))$. The operational and denotational semantics of a language are congruent if all constructs in the language have this property, which implies that the behaviours predicted for any term by the denotational semantics are always the same as those that can be observed of its operational semantics. That the standard semantics of CSP are congruent to a natural operational semantics is shown in, for example, [2].

- the failures on non-divergent traces are precisely those (s, X) where the process can, after s, come into a *stable* state (one with no internal progress possible) that has no transition in X;

- the minimal divergences are the actual least traces after which the process can perform an infinite sequence of τ's;

- all failures on divergent traces, and non-minimal divergences, are present because of our decision not to model this type of behaviour: the fact that they are there carries no information (positive or negative) about what the process can actually do. It is a deliberate obfuscation, introduced to get the theory to work better.

What this statement actually defines is the abstraction mapping Φ from the labelled-transition-system operational semantics of CSP (or, indeed, any other labelled transition system where internal actions are labelled τ) to the model.

In this paper we will be considering largely this model, though the techniques in this paper can be closely related to recent work on seeing beyond the first divergence [6] and, perhaps less expectedly, to the extensions of the model [2] by infinite traces to handle unboundedly nondeterministic constructs and specifications. We will discuss these possibilities briefly in Section 5. We will also only deal with the case where the overall alphabet of possible actions is finite, since this makes the model a little more straightforward, and is an obvious prerequisite to model-checking.

Given the philosophy described above of what the failures and divergences mean, the following properties, which define the model, should be self-evident. The *failures/divergences model* $\mathcal{N}_\Sigma$ over a given finite alphabet Σ of communications is the set of all pairs (F, D), $F \subseteq \Sigma^* \times \mathcal{P}(\Sigma)$, $D \subseteq \Sigma^*$ satisfying

$$\begin{array}{lrcl}
(F1) & (F \neq \{\}) & \wedge & ((st, \{\}) \in F \Rightarrow (s, \{\}) \in F) \\
(F2) & (t, X) \in F \wedge Y \subseteq X & \Rightarrow & (t, Y) \in F \\
(F3) & ((t, X) \in F \wedge (t\langle a \rangle, \{\})) \notin F & \Rightarrow & (t, X \cup \{a\}) \in F \\
(D1) & s \in D & \Rightarrow & st \in D \\
(D2) & s \in D & \Rightarrow & (st, X) \in F.
\end{array}$$

The process $P = (F, D)$ *refines* $P' = (F', D')$, written $P' \sqsubseteq P$, if and only if

$$F \subseteq F' \qquad \text{and} \qquad D \subseteq D'\,.$$

The semantics of CSP over this model may be found in many places, including [7, 1]. Important facts that we will need in this paper include the following:

1. The least process under the refinement order, $\perp$, equates to any process that can diverge immediately (i.e., without performing any visible actions first). The refinement order, under the restrictions given, is complete and its maximal elements are the *deterministic* processes: divergence free and, after each trace s, only able to refuse those events that it cannot communicate after s.

2. Each standard CSP operator can be defined as an operator over $\mathcal{N}_\Sigma$; each is continuous with respect to the refinement order; using that order and least fixed points for the semantics of recursion gives a denotational semantics that is congruent to the operational semantics.

The congruence theorem is vital in underpinning the model checking work, since all of the computations we do to test refinement – a notion based on the abstract model – are in fact carried out over the operational semantics. The rest of this section develops the theory of how the question $P \sqsubseteq Q$? is decided.

Though there are possibilities for relaxing this at the specification (left-hand) side, we choose only to deal with processes whose operational semantics are *finite state.* This is defined to mean that, as the operational semantics is unfolded, only finitely many process terms are generated. Thus,

$$P = a \rightarrow ((b \rightarrow STOP) \square (c \rightarrow P))$$

is finite state – it has 3 or 4 four states depending on whether one adopts the operational semantics in which a recursion is unfolded directly or guarded by a τ[3]. On the other hand,

$$Q = a \rightarrow (Q \parallel\!| (b \rightarrow STOP)$$

is infinite-state, and fundamentally so since it is a process which can always communicate a, and will communicate b provided this will not make the number of b's so far exceed the number of a's. There are some simple rules to help determine what classes of CSP process are likely to be finite state. These are discussed in Section 3.2.1. We will therefore, for the rest of this section, think of the decision question above as being one between finite directed graphs where all edges are labelled with an action, either τ or visible (finite labelled transition systems). It divides into two parts: normalising the specification, and then checking the implementation against the resulting normal form.

2.1 Normalising a transition system

The transition systems arising from CSP descriptions typically contain a high degree of nondeterminism, in the sense that after any trace s of visible actions there may be many states of a system which the process might be in. This can happen both because of the existence of invisible actions and because of the branching that occurs when a node has two identically-labelled actions, whether visible or invisible. Any method for deciding refinement between these systems will have to keep track of all the states reachable at the specification side on a given trace s, since it is merely necessary than every behaviour of the implementation on s is possible for one of these. And any method we devise will need a way of telling when a large enough set of traces have been tried to establish refinement.

Life would be easier if there were exactly one state corresponding to each possible trace. This can be achieved by transforming the original specification transition system to an equivalent *normal form.* The idea of a normal form for CSP processes has its origins in [8], where it is shown that each finite CSP term is equivalent to one in the following normal form:

- $\perp$ is a normal form;

[3] Over the failures/divergences equivalence, a node N in a labelled transition system is indistinguishable from one whose only transition is τ to N. The full operational semantics of CSP requires that the rule for a recursive term be

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

in order to give a semantics to action-unguarded terms such as $\mu\, p.p$ (which the reader will note is correctly given a divergent operational semantics). However, since terms like this are trivially detectable syntactically, and are always semantically equivalent to $\perp$, we actually choose to disregard them and *not* introduce the τ guard. This gives transition systems with the same abstract semantics and fewer states.

- all others take the form

$$\sqcap\{(x : A \to P(x)) \mid A \in \mathcal{A}\}$$

for $\mathcal{A}$ a *convex* subset of Σ such that $\bigcup\mathcal{A} \in \mathcal{A}$, and each $P(x)$ a normal form which depends only on x, not on A

Except for trivial re-orderings (which are invisible in the above presentation in any case), two normal forms are equivalent semantically if and only if they are equivalent syntactically. Thus, except for the special case of $\perp$, each process is determined by

- its initial actions ($\bigcup\mathcal{A}$);
- its minimal acceptance sets, the minimal elements of $\mathcal{A}$, which are in direct correspondence with the maximal refusal sets;
- its behaviour after each initial action.

The most vital property, for us, of this normal form is that it branches uniquely and on visible actions only: given a starting normal form and one of its traces, we can follow this trace and get the unique normal form state the process is in after it.

The normal form was initially conceived as a target for algebraic transformation: if we can give enough laws so that any finite term can be transformed to normal form, then the laws can reasonably be said to be *complete*. However, since the normal form has a very clear relationship with behaviour, it is possible to compute it from the operational semantics, or indeed to find a normal form for any member of a labelled transition system: for any trace s of a process,

- if P can diverge on some proper prefix of s, there is no normal form node corresponding to s;
- if P can diverge after s but on no proper prefix, then the normal form node is $\perp$;
- otherwise, the initial actions are just the actions a such that $s\langle a\rangle$ is a trace of P (namely, there is some node of P that has trace s and can perform a); the minimal acceptances are the smallest sets of visible actions possible for stable nodes with trace s; the successor nodes are just those computed this way from the $s\langle a\rangle$.

For a process that has infinitely many traces, this formulation generates an infinite normal form. To be usable in our context we need to close the graph by re-using nodes that have identical behaviour (both on the first step and forever after). However, for finite-state processes there is a finite-state version of the normal form, computed using the following two-stage process.

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 $\mathcal{P}(V)$ as follows

- The initial node is $\tau^*(v_0)$, where $\tau^*(v)$ is defined to be $\{w \mid v \xrightarrow{\tau} {}^*w\}$, the nodes reachable under some sequence of τ's from v_0.
- For each node generated we have to decide whether it is divergent: this is true if and only if it contains a divergent node of L. A method for deciding this is described in Section 2.2. A divergent normal form node has no successors.

- 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.

The resulting graph will be termed the *pre-normal* form of L.

This generates a transition system where there are only visible actions and where each node only has a single successor for each of its actions. What we produce by this process is the set of all the sets of nodes of L that are those reachable on any trace. Given any trace of v_0, the set of nodes of L that can be reached is one of the nodes of this new graph, except in the case of a divergence. The possible refusals of one of these nodes v are just the sets refused by the stable $v \in N$ (note that each non-divergent N must have such v).

As examples, we will normalise the processes B_3 and Q_0 defined:

$$B_3 = COPY \gg COPY \gg COPY$$

$$\begin{aligned} Q_i &= (a \to STOP) \,\Box\, (a \to Q_{i+1}) \quad \text{for } i = 0, 1 \\ Q_2 &= (a \to Q_0) \,\Box\, (a \to Q_1) \end{aligned}$$

where $COPY = left \to right \to COPY$. ($\gg$ is a parallel operator that connects the *right* channel of the left-hand argument to the *left* channel of the other, and hides the result.) The transition systems of each of these processes, and the corresponding pre-normal forms, are shown in Figure 1. B_3 has 8 states, corresponding to each of the component processes being in state $COPY = C$ or $right \to COPY = F$. Its pre-normal form has 4 states, one of each possible number of 'items' this 'buffer' is holding (**0**,**1**,**2** or **3**). Though Q_0 has only 4 states and one possible action, its pre-normal form has 6 states. These are shown superimposed on the base transition system as sets of states: the numbers indicating how many a's are taken to reach it. The pre-normal form is also shown below, with the solid and empty circles respectively denoting states that can, and cannot, refuse a.

Stage 2 Examining the pre-normal forms generated by our two examples, there is a real sense in which the nodes from B_3's pre-normal form are all essentially different, for they allow different numbers of *left* events before this is refused. On the other hand, nodes **4** and **5** from Q_0's pre-normal form are actually indistinguishable: each of them can perform as many a's as it pleases, with the ability to $STOP$ (refuse everything) at any time. Thus the behaviour if this system is equivalent to one with five nodes, formed by identifying the last two in the pre-normal form. This is why we have termed $\mathcal{P}_L$ a pre-normal form rather than a normal form, for in a true normal form we would not expect there to be two essentially different representations of the same thing. To make it into a normal form we have to identify semantically identical nodes such as these.

The question of which nodes should be identified is easily determined by first marking each with *either* $\perp$ *or* its initial actions and maximal refusals, as appropriate, and then computing the fixed point $\sim$ of the following sequence of equivalence relations:

- $N \sim_0 M$ if, and only if, they have the same marking.

- $N \sim_{n+1} M \Leftrightarrow (N \sim_n M) \wedge \forall\, N', M'.(N \xrightarrow{a} N' \wedge M \xrightarrow{a} M' \Rightarrow N' \sim_n M')$.

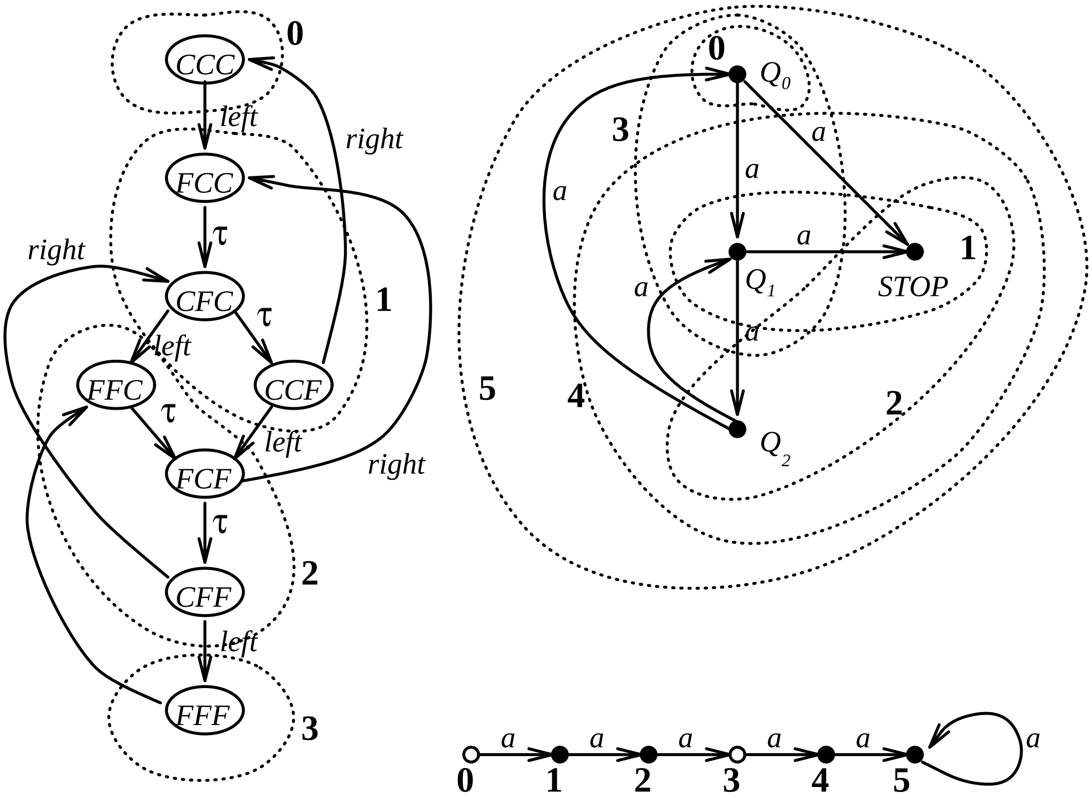

Figure 1: Pre-normalising two transition systems

Note that in the example above the initial marking corresponds to the 'full' or 'empty' marking in Figure 1. This equivalence over pre-normal form states is just a simple form of bisimulation with 'coloured' nodes. It is made particularly straightforward by the fact that two identically marked nodes have the same initial actions, and exactly one successor node for each such action.

The *normal form* of the labelled transition system L is thus its pre-normal form under the equivalence relation $\sim$, where the initial node is the equivalence class of $\tau^*(v_0)$, the transition relation is the obvious one and the only detail we record about each node is the marking as described above.

The complexity of normalisation

Given that the pre-normalisation process builds a transition system over the powerspace of the original one, there is the possibility that the normal form will be exponential in the size of the original system. This can indeed occur, as is shown by the example below. This possibility is to be expected given the result [9] that checking failures/divergence refinement of transition systems is PSPACE-hard. This result, which has been known for some time, can be blamed for the failure of the tool-building community to tackle failures/divergence refinement hitherto.

Fortunately there are two mitigating factors that work in our favour, making this particular obstacle more-or-less disappear.

1. 'Real' process definitions simply do not behave as badly as the pathological example below. In practice it is rare for the normal form of a naturally-occurring process to have more states than the original transition system. Indeed, the normal form is frequently significantly *smaller*, offering scope for intermediate compression. It seems that, at least tackled using our algorithms, deciding failures/divergence refinement is one of that large class of NP-complete and similar problems where the hard cases are rare.

2. It is only the *specification* end of the refinement that we have to normalise. In practice the simpler process is usually that one rather than the implementation. Frequently, indeed, the specification is a representation of an abstract property such as two events alternating or deadlock-freedom, and has a trivial number of states.

   One would usually expect that a process playing the role of a 'specification' is reasonably clearly and cleanly constructed, with understandable behaviour. These aims are more-or-less inconsistent with the sort of nondeterminism that leads to an explosion in the normal form.

Example The potential for state explosion on normalisation is shown in extreme form by the following pathological system, defined for any $n > 0$. We construct a transition system with $n + 1$ states and n events $\{1, \ldots n\} = B$. If the kth state is P_k we define

$$
\begin{array}{lcll}
P_0 & = & STOP & \\
P_k & = & r : B \to P_{k+1} & k \in \{1, \ldots, n-1\} \\
P_n & = & r : B \to P_0 & \\
 & & \Box\, r : B \to P_1 & \\
 & & \Box\, r : B \to P_r &
\end{array}
$$

This system is illustrated in Figure 2. The pre-normal and normal forms of this system (with initial state P_1) both have precisely $2^n - 1$ states (one for each nonempty subset A of the states $P_1, \ldots, P_n$ since, as can be proved by induction on the size of A, there is for each such subset a trace s_A such that the states reachable on s_A are A and perhaps P_0). The role of the state P_0

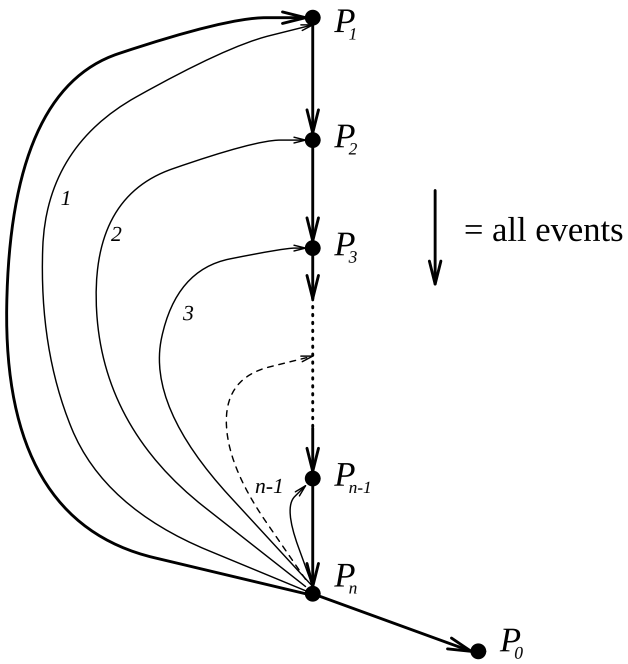

Figure 2: Transition system with pathological normalisation

is to allow us to distinguish between these sets: if it is in the set reachable after a given trace then P_1 can deadlock after that trace, otherwise not. If the state P_0 is removed we get a system with the same number of pre-normal form states but only *one* normal form state, since every one is then semantically equal to

$$RUN = x : B \rightarrow RUN$$

2.2 Checking refinement

Once the specification end of a refinement check has been normalised, the following two phases 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.

A state P is divergent if, and only if, the directed graph formed by considering only τ actions has a cycle reachable from P. There are two standard algorithmic approaches to this problem: computing the transitive closure of the graph or taking advantage of the properties of depth-first search (DFS). The latter is more efficient, since it offers an essentially linear (assuming constant-time set membership test) algorithm, based on the fact that in a DFS, the set of partially explored nodes (ones that have been visited, but not all nodes reachable from them) always forms a chain from the root of the search. The marking process[4] proceeds as follows:

[4]This algorithm was devised by Michael Goldsmith.

- Perform DFS's rooted at successive nodes of the graph until each node has been visited at least once.

- Any node can be marked non-divergent, divergent or unresolved, initially all being unresolved. During each individual search, only the successors of unresolved nodes are explored.

- If, during any DFS, either a node on the current partially explored chain or a marked divergent node is reached, then all of the nodes on the partially explored chain are marked divergent and this particular DFS is terminated.

- When a node leaves the partially explored chain because all its successors have been explored without the above occurring beneath it, then it is marked non-divergent (as must all its successors have been). For the root node this corresponds to the end of the DFS.

In the model checking phase we have to discover whether all the behaviours of each implementation state are allowable in all the normal form states such that they have a trace in common. This is done by exploring the cartesian product of the normal form and implementation state machines as follows.

We maintain a set of checked pairs and a set of *pending* pairs; initially the former is empty and the latter is the singleton set of the pair of initial states of the two systems. Until it is empty, we repeatedly inspect pairs from *pending*. The pair $\langle \nu, w\rangle$ checks if (i) the normal-form state ν is divergent *or* (ii) the implementation state w is non-divergent *and*

- the set of initial actions of w is a subset of those of ν *and*

- either w is unstable (has a τ-action) *or* the set of actions it refuses (the complement of its initials) is a subset of one of the maximal refusals of ν.

The set of all pairs reachable from $\langle \nu, w\rangle$ and not in *checked* is added to *pending*. A pair $\langle \nu', w'\rangle$ is *reachable* from $\langle \nu, w\rangle$ if either

(a) $w \xrightarrow{\tau} w'$ and $\nu = \nu'$, or

(b) $w \xrightarrow{a} w'$ and $\nu \xrightarrow{a} \nu'$ for $a \neq \tau$; noting that whenever $w \xrightarrow{a} w'$, then this ν' must exist and be unique.

If a pair is found which fails to check, then the proposed refinement does not hold. If the above process completes without finding such a pair, then refinement does hold.

The first case (failure of refinement) is easy to see: the implementation has a trace (the sequence of visible actions used to bring the cartesian product to the failing state-pair), which can bring it into a state with behaviours not possible for the normal form. This means there is a divergence, a trace or a refusal of the implementation that is not allowed in the specification. Provided we have a way of finding the trace that led to a specific pair $\langle \nu, w\rangle$ being considered, it is thus possible to report not only the failure of refinement but also a reason.

Conciseness being a virtue, it is as well to report the shortest possible sequence of actions that can lead to an error. This is achieved if we perform a breadth-first search (BFS) during the model-checking phase. We shall see later that this brings other advantages.

The case of successful proof is a little more subtle. One can justify the claim that refinement always holds in these circumstances either operationally or abstractly. Operationally, if refinement fails, then it must hold because the implementation has some behaviour that is banned by the normal form. There is therefore some sequence of actions which exhibit this, potentially bringing the implementation into some state w where it can (a) diverge illegally, (b) perform an event that takes the traces outside the set permitted by the normal form or (c) refuse a set not refused by the normal form. It is clear that the unique normal form state ν corresponding to this trace is such that $\langle \nu, w \rangle$ will be found in the search above, leading to a failure to check.

Abstractly, one can show that the refinment checking process is simultaneously formulating and proving a sort of mutual recursion induction over the state-space of the implementation. This style of proof is discussed in [10], though one also needs the result of [11] that a divergence-free component of a CSP mutual recursion is identical in any fixed point of the recursive function.

3 Implementation Issues

3.1 Machine-Readable CSP

CSP has been, in the best tradition of 'blackboard' languages, a rather variable notation, with some parts being used differently in different schools and some parts being rather under-defined. Since it is written, essentially, as a series of algebraic expressions using a variety of peculiar-looking operators, it does not look like the sort of programming language one usually types into a computer.

FDR emerged as part of a general effort to make tools available for CSP. There was thus an obvious need for a standardised syntax, parser and type theory/checker for the language. The work[5] to develop these has been led by J.B. Scattergood [14], and has gone on hand-in-hand with that on FDR which, together with several other tools, uses them.

The objective of this work has been to preserve as much as possible of the form, spirit and flexibility of the blackboard language, while bringing in structure to allow for building the larger programs we can expect to see with the availability of tools and practical use. For portability, it was decided to do all of this within the confines of an ASCII syntax – where the chief compromise is in the representations of the operators – though this does not preclude the building of more elaborate display and printing facilities on top of it.

Aside from trivial issues such as the shapes of operators, most of the decisions in the design of the syntax revolved around the sub-process objects such as events, their atomic components and parameters to processes (which can be thought of as process state). Unless we change the CSP semantics of termination and sequencing, the treatment of these objects is necessarily declarative: there is no assignment or similar construct that can change the value of an existing variable within its scope. This makes CSP unlike occam, which requires a more elaborate semantics of termination [12, 13], and also contrasts with Hoare's earlier 'CSP 1978' [15]. It also affects the style of CSP programs, which become *scripts* of definitions (of a mixture of processes and other objects) in the style of functional programming languages such as Haskell and that of [16].

More specific features of the machine-readable syntax include:

1. In [1], every process has an intrinsic alphabet, with these being important for the semantics of the parallel operator. A pair of processes must synchronise on events in the intersection of their alphabets. We have found it more convenient to supply either these alphabets directly when using the parallel operator or, more usually, to define the interface set of each parallel composition: $P \underset{A}{\|} Q$, written `P [|A|] Q` in ASCII, makes P and Q synchronise on elements of A. Thus, processes in our standard syntax *do not* have intrinsic alphabets.

[5] The CSP parser and typechecker were produced under an ONR-sponsored project, and are freely available to anyone interested in producing tools related to CSP.

2. Each event is made up from a constructor representing its name and a number of 'data' components from a fixed list of types (which may be empty). Each 'channel' thus has a possibly trivial cartesian product type. The events are formed using infix dots, sometimes by `?` representing input and binding the identifiers to their right until the occurrence of a `!` representing output. Thus the event `a.1?x!y` is a communication over channel `a`, whose first and last components are fixed and the middle one is open and will bind `x` to the value input. FDR demands that all channels used are specifically declared.

3. The syntax has specific support for booleans, numbers, sequences, tuples and sets. Any undefined identifiers (ones not bound to anything specific) are treated as tokens or constructors introduced by the user.

3.2 Implementing checking and debugging

3.2.1 The front end

FDR has a Standard ML front end (itself sometimes hidden by an X Windows interface) which calls on optimised routines in C to perform computationally-intensive tasks. The front end's main role is to compile the high-level CSP notation into a form that can be efficiently manipulated at the lower level, and to interpret the results.

FDR makes a distinction between *low-level* processes and *high-level* processes. The former are intended to be compiled into a raw state-machine representation by the ML. The latter are typically the composition of several low-level processes: FDR deals with these by compiling these components in the front end and using the back-end routines to combine them. At the time of writing this distinction is made by dividing the operators of CSP into two classes.

- Low-level operators are prefixing, the various forms of choice, conditional, sequencing and all parameterisation of processes.

- High-level operators are all forms of parallel, hiding and renaming.

No low-level construct may be used syntactically outside a high-level one, and the low-level components of a system are just the maximal components composed of low-level operators.

This hierarchy is generally consistent with the way CSP is used, and indeed what it in fact enforces is the notion of a CSP program as a system of *communicating sequential processes*! It is also close to the syntactic restriction necessary to enforce finite state spaces. Infinite state spaces can be created in CSP through the effects of parameterisation, and recursion through the left-hand argument of ; (sequencing) or through parallel, hiding and renaming in various combinations. Of these, only the first two remain possible in our restricted syntax. While not forbidden in FDR, use of recursion through the left-hand-side of sequencing should be used only with care for obvious reasons. Similarly, recursions parameterised by infinite data types need to be treated with care.

3.2.2 The back end

There are three main functions carried out by the back end in the current version of FDR: expanding and normalising the specification process, divergence-checking the implementation, and doing the model-checking of the two. The essential features of these algorithms have already been set out in an earlier section, so we will restrict attention to one important issue. This is how to deal efficiently with the problems that appear in managing the very large sets of states that typically appear in checking a large parallel system.

Because more elaborate computations are needed over the specification system, the simple enumeration of its states is typically not a limiting factor. Thus the size of state-space we can deal with at the implementation end of refinement is currently two to three orders of magnitude larger than that at the specification end. This has rarely proved a problem, for essentially the reasons discussed in Section 2.

The most interesting issue is the difference between the DFS used for divergence checking and the BFS used for model checking. Both of these algorithms rely on the maintenance of sets of which states have been visited before. In a DFS we need a constantly updated representation of this set so that we can discover whether the state we are looking at has been visited before (and in our case, whether it is on the current chain). Since we immediately look at the children of the most recently visited node, there is no potential for grouping these tests. Our preferred representation for the set in this case has thus been a hash table.

In performing a BFS there is a high latency between generating a pending state and actually visiting it, if it is new. The test of whether it is new can be performed at any time during this interval. A natural way of doing this is to look at all successor states generated during a particular 'layer' of the search. If the current list of visited states is stored as a sorted list, then we can find out which of the successors are new by sorting the successors (naturally, removing duplicates) and then merging them into the established list. Obviously the successors of each genuinely new state can be generated as it is merged in.

This latter method has been found to be faster and more memory-efficient than the former. The greatest benefit appears when the set of states becomes too large to fit within the physical memory of the computer. Hash table representations of sets map so badly onto virtual memory that they are effectively unusable at this level. The sorted list representation maps onto virtual memory extremely well: all access into the lists is sequential (depending on which sorting routine is used during that phase). We have found very little degradation in performance in checks when they are forced to use virtual memory.

This fact has meant that it is often significantly quicker to perform the model-checking phase than the divergence-checking, and indeed the former can frequently be performed on systems when the latter is not within reach. FDR thus provides the option of 'failures-only' checking, which establishes whether all the traces and stably-observable failures are allowed by the specification. Of course this proves full refinement on the assumption that the implementation is divergence-free, something which is frequently either obvious or can be established by other methods. The notion of failures-only checking also has independent interest in connection with [6].

3.2.3 Debugging

When a state-pair is found which fails to check, we would like to be able to help the user find why things went wrong. To do this, FDR discovers a shortest sequence of actions that bring the system into the errant state, and helps the user to see the state and actions each component process of the implementation contributed to the problem via its graphical interface.

The simplest way of reconstructing the path taken by the BFS to the error state is by storing, with each state, a record of its parent. Unfortunately this essentially doubles the storage requirement. It is more efficient (for the overall operation of the system) to store with each state-pair the level of the BFS on which it was generated. The path can then be reconstructed straightforwardly by essentially running the transition system backwards starting from the erroneous one: if we have currently got a state-pair at level $N+1$, the next one is any pair at level N of the original BFS from which the $N+1$ one can be reached.

As indicated above, it is possible for FDR to give a detailed analysis of why the implementation performed an illegal behaviour. An interesting problem that has arisen in practical use is that it is sometimes the *specification* that is 'wrong', not saying what was really intended. A 'missing' behaviour is, of course, output. But it is much harder to attribute blame within the specification for why it did *not* allow it, both because of the large transformation the specification has gone through on normalisation, and because the lack of a behaviour is a much harder thing to pin down than why one is there.

4 FDR in practice

The title of this section could equally be 'CSP in practice', since the utility of FDR derives from the expressive power of the notation and of CSP refinement. The former is important in the effective way we can express a wide range of systems, the second because of the wide range of correctness questions that can be resolved using only one well-sharpened tool.

FDR actually has its origins in a real, practical problem: the specification and design of the VCP and other communications hardware of the Inmos T9000 transputer family [17, 18]. Inmos identified the need for a tool to support failures/divergence refinement on realistic-size problems (then of order 10^4–10^5 states) with debugging facilities.

Since those early prototypes both the algorithms, and in particular the interface, have become a great deal more sophisticated and the tool has been used in a variety of studies, both practical and academic. At the time of writing (June 1993) it is capable of dealing with implementation state-spaces of order 10^7 on a moderate-sized workstation (where a check of size 10^6 might take 20 to 60 minutes) and order 10^4 normal form states. We expect considerable improvements, by a variety of methods such as those detailed in the next section.

Two things about CSP work hand-in-hand to make FDR useful: the theory of refinement and the expressive power of its operators. Refinement both brings a theory of program development, though its transitivity and monotonicity, and also, since it is based on nondeterminism, gives a uniform way of expressing correctness conditions. Any condition R on processes that takes the form of specifying that each behaviour of a process satisfies some predicate (a *behavioural specification*) – and most practical conditions take this form – has a most nondeterministic process P_R that satisfies it. A general process Q then satisfies R if, and only if, $P_R \sqsubseteq Q$.

CSP is fortunately well-equipped with notation for expressing nondeterminism, so that it is usually possible to find a concise representation for P_R in the language. Such a representation may be finite or infinite state. 'Safety' and 'liveness' properties can both be written in this form, as can ones that mix these two ideas. For example,

$$P \underset{\{a,b\}}{\|} CHAOS(\Sigma) \qquad \text{where} \quad P = a \rightarrow b \rightarrow P$$

specifies that a and b alternate, without expressing any condition on other events. $CHAOS(A)$ is the most nondeterministic divergence-free process using the events in A:

$$CHAOS(A) = STOP \sqcap (x : A \rightarrow CHAOS(A))$$

This is a *safety* specification because it allows any refusal. *Liveness* specifications usually allow any trace, but put limitations on what can be refused. A typical – and the most commonly used – example is

$$DF = \textstyle\bigsqcap_{a \in \Sigma} a \rightarrow DF$$

which states that the process is deadlock-free. Note that DF can perform any trace, and select any event at any time, but may not refuse all events.

Safety and liveness properties tend to represent desirable facets of a process, rather than attempting to capture the full essence of intended behaviour. One that does this is likely to contain both safety and liveness elements. A particularly simple example which might apply to a communications protocol is that it should behave like a one-place buffer:

$$COPY = \mathit{left}?x \rightarrow \mathit{right}!x \rightarrow COPY$$

(Note that this simply adds data to the version of $COPY$ seen earlier.) This will apply to most implementations of the alternating bit protocol, for example. $COPY$ is actually a *deterministic* process, so that $COPY \sqsubseteq P$ implies $COPY = P$: refining it makes a very precise statement indeed. In many protocols a certain amount of *buffering* is introduced, and we may well not care how much. We would therefore want to generalise the above specification to that of the most nondeterministic buffer:

$$\begin{array}{rcl} B_{\langle\rangle} & = & \mathit{left}?x \rightarrow B_{\langle x\rangle} \\ B_{s\langle y\rangle} & = & ((\mathit{left}?x \rightarrow B_{\langle x\rangle s\langle y\rangle}) \sqcap STOP) \\ & & \Box\ \mathit{right}!y \rightarrow B_s \end{array}$$

This process cannot refuse to input when empty, or to output when nonempty. It never loses any input and preserves output. The nondeterminism comes in because it *may*, but is not *obliged to* accept further input when non-empty. This specification is infinite-state, since the buffer can hold any number of items. In practical use it is thus necessary to introduce some maximum permitted buffering.

The ability of CSP to describe nondeterministic behaviour is also useful at the implementation side, both in cases when some part of the system (such as an unreliable communications medium) is outside the control of the programmer, and also in cases where it is desired to abstract away from the way decisions are made. The former was illustrated in [19], which describes how FDR can be used to verify communications protocols. It has also been used to good effect in the development of fault-tolerant systems. The latter can be valuable both by cutting down state-spaces and by proving more general results. These ideas are discussed in [20], for example.

CSP is unusual among process algebras in the way it makes multi-way synchronisation natural, Hoare pointing out that parallel composition acts like conjunction over trace specifications. A fascinating illustration of the power of this notion is the way FDR can be used to solve the well-known puzzle *peg solitaire*. A move in this game, illustrated in Figure 3, consists of a peg hopping up, down, left or right over one peg, which is removed, into an empty hole. Starting with 32 pegs and a hole in the centre, the objective is to have left only a single peg, in the centre hole. The reason why we have coloured eight pegs differently will become apparent shortly.

Thinking of each of the 33 squares as a separate process, each move becomes a 3-way synchronisation (between the from-, over- and to-squares). A typical square, at co-ordinates (i, j) has two states (*Full* and *Empty*) and can be written:

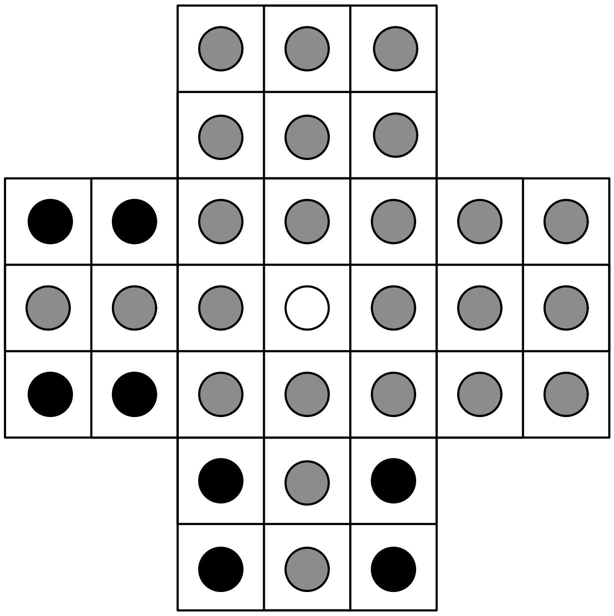

Figure 3: Peg solitaire showing tactically fixed pegs

$$
\begin{array}{rcl}
Empty(i,j) & = & left.i.j \rightarrow Full(i,j) \\
& & \Box\ down.i.j \rightarrow Full(i,j) \\
& & \Box\ up.i.j \rightarrow Full(i,j) \\
& & \Box\ right.i.j \rightarrow Full(i,j) \\
Full(i,j) & = & left.(i-2).j \rightarrow Empty(i,j) \\
& & \Box\ down.i.(j-2) \rightarrow Empty(i,j) \\
& & \Box\ up.i.(j+2) \rightarrow Empty(i,j) \\
& & \Box\ right.(i+2).j \rightarrow Empty(i,j) \\
& & \Box\ left.(i-1).j \rightarrow Empty(i,j) \\
& & \Box\ down.i.(j-1) \rightarrow Empty(i,j) \\
& & \Box\ up.i.(j+1) \rightarrow Empty(i,j) \\
& & \Box\ right.(i+1).j \rightarrow Empty(i,j)
\end{array}
$$

Note that there are four 'channels', corresponding to a move in any of the four directions. The move *direction.i.j* represents the move *to* square (i,j) from the appropriate source.

We form the complete starting position from 32 *Full* squares and 1 *Empty* one, synchronising on appropriate events. Or nearly, since the use of the generic square process means that most (all but the nine middle ones) of the processes have non-real events possible: ones relating to squares off the edge of the board. This problem can be removed at once by putting the combination in parallel with the process *STOP*, synchronising on all these false events, thereby banning them. We then have a process, which we can call *BOARD*, which allows precisely the sequences of moves allowable in solitaire.

A solution to the puzzle is thus a trace of 31 moves, with the last one a jump to the centre square ($(3,3)$ in the co-ordinate system where the left and bottom squares have x and y co-ordinate 0). In principle we can find this by specifying it is impossible, checking the result (which will fail) and using the debugging information to find the winning sequence of moves.

The specification for this is

$$\begin{aligned} TEST &= MOVE(30) \underset{\Sigma}{\|} CHAOS(\Sigma), \quad \text{where} \\ MOVE(0) &= x : (\Sigma \setminus Middle) \to STOP \\ MOVE(n+1) &= x : \Sigma \to MOVE(n) \end{aligned}$$

where *Middle* is the set of moves to (3,3).

Unfortunately there are too many states in this pure version of the puzzle (order 10^9-10^{10}). Here, multi-way synchronisation comes to our rescue again, for we can put *BOARD* in parallel with processes that put restrictions (*tactics*) on the moves we allow. One simple one that works is to forbid 8 specified pegs (shown in Figure 3) from moving until the 20th move. This finds a solution in 729,833 states.

A number of other similar puzzles have been solved similarly, including one example where FDR beat the best-known solution by some way!

This example is interesting because it shows both how CSP can be used to represent state-, and rule-based systems outside the domain one would at first expect, and how multi-way synchronisation can be used to carry out restricted tests in cases where a full check is either unnecessary or unacceptably slow.

5 Prospects

Even though FDR has already shown itself to be a practical tool, there are numerous ways in which it can be developed. These can be broken into the categories of improvements to the basic structure, and of widening its scope. All of the following possibilities have either been experimented with or are under consideration.

5.1 General Improvements

5.1.1 Inference and infrastructure

The present tool provides a powerful calculator for deciding refinement questions. The properties of refinement (monotonicity, transitivity, etc.) mean that there are potentially many inferences one can make once one has established a few refinements between processes in a system. There is the potential to build a proof/development management system on top of the calculator to record what facts have been established in a session (or longer), record any assumptions that the user might wish to assert, and make appropriate deductions (perhaps by integrating an established logical inference tool).

Once one has such an infrastructure, it will be possible to build in other tools, such as algebraic transformations, and support for deadlock or divergence analysis using analytic rather than model-checking techniques (though probably supported by them). Tools for establishing divergence-freedom are particularly desirable, given the difficulties discussed in Section 3.2.2.

5.1.2 Compression

Currently FDR expands the state-spaces of its processes fully and *explicitly*. Since the state-spaces of parallel compositions typically grow exponentially with the number or processes involved, this limits the size of problem that can be dealt with: while we can deal with many realistic problems, there are also many which are too large and others where considerable ingenuity has to be exercised to keep the size manageable. Methods which do not exhaustively list all reachable states, rather proving results of the state space in blocks, are known as *implicit*.

Three general types of method are known to me: Binary Decision Diagrams (BDDs), exploiting compositionality, and the symbolic representation of state. The first of these has attracted a considerable amount of work, most notably from Clarke [21], and it is possible to 'cover' enormous state-spaces provided the problems have sufficient regularity. Since the usual problem-domain of this type of system (both in terms of the type of property proved and the type of system analysed) is somewhat remote from that of FDR we have no immediate plans to incorporate this type of technology, though in future, in a large integrated tool, there will almost certainly be a role for it.

Like BDD's, compressing state-spaces via compositionality is a technique which is likely to work well for some examples and badly for others. The basic idea is to attempt to compress the state-spaces of the intermediate syntactic components of a system as it is built up, rather than letting it just multiply. One would thus expand the states of a component, compress it, and then use the compressed version in building up the system from there on. What we need of the compressed version is that it should have the same semantics as the original (failures, or failures and divergences depending on the sort of check being done).

The obvious techniques for compression include direct forms of surgery on transition systems, cutting out sequences of internal actions for example, and normalisation, both discussed earlier and possibly also in 'failures-only' and 'traces-only' forms. In our experience, normalisation can be expected to reduce the size of a state space of a system if that system includes hiding of internal actions *providing* the system does not include explicit nondeterminism ($\sqcap$) or many nondeterministic arbitrations that are hidden. (In other words, the internal actions should essentially be 'making progress' rather than 'making choices'.)

Either of these will make it desirable that we extend the basic notation of labelled transition system to allow nodes to have sets of maximal refusals: any stable node must have at least one, and nodes with τ-actions may now have them. In other words, we want a notion of transition system which includes both the traditional one and the additional information contained in the normal form systems.

One of the most interesting and challenging things when incorporating these ideas will be preserving the debugging functionality of the system. The debugging process will become hierarchical: at the top level we will find erroneous behaviours of compressed parts of the system; we will then have to debug the pre-compressed forms for the appropriate behaviour, and so on down.

When doing failures-only checking (i.e., disregarding the possibility of divergence) it will be possible to hide any events which can be made irrelevant to the specification, greatly increasing the scope for compression of this sort. The most spectacular example of this is deadlock-freedom, where the whole alphabet can be hidden: a general process P can deadlock if and only if $P \setminus \Sigma$ can (necessarily on the empty trace).

Consider the case of the N dining philosophers (in a version, for simplicity, without a Butler process). A natural way of building this system up hierarchically is as progressively longer chains of the form

$$PHIL_0 \| FORK_0 \| PHIL_1 \| \ldots \| FORK_{m-1} \| PHIL_m$$

In analysing the whole system for deadlock, we can hide all those events of a subsystem that do not synchronise with any process outside the subsystem. Thus in this case we can hide all events other than the interactions between $PHIL_0$ and $FORK_{N-1}$, and between $PHIL_m$ and $FORK_m$. The failures normal form of the subsystem will have very few states (typically 4, though the number may vary for small m and different versions of the system). Thus we can compute the failures normal form of the whole hidden system in time proportional to N, even though an explicit model-checker would find exponentially many states. (At a higher level, this particular example offers the possibility of inductive proof, proving results independent of N. This is a separate issue.)

A mode of state-space compression that is possible in the present system relates to systems of the form $C[P]$, where $C[\cdot]$ is a *context* and P is a subprocess whose states we wish to compress to allow reasoning with $C[P]$. If we can find a specification S with fewer states than P such that $S \sqsubseteq P$ and S either is equivalent to P or is thought to capture all the properties of P required by $C[\cdot]$, then it makes sense to analyse $C[S]$ rather than $C[P]$. An example of this might be where P is a communications protocol and S is an appropriate buffer specification.

This suggests a further possibility for compression in cases where the normal form of some subsystem is larger than we might wish (offering negative or little saving). A normal form where this occurs is likely to contain a number of (powerspace) nodes containing the initial node system being normalised. The process represented by any of these will be an approximation to the initial node, and the larger the set (i) the smaller the set of normal form states reachable from it and (ii) the worse approximation one gets. In the second example of Figure 1, apart from node **1**, with 5 normal form states, we could choose **3** or **4/5** with 2 and 1 states respectively. Clearly the use of such approximate compressions will sometimes give false negative results (assuming the compression occurs only on the implementation side), though these will be detected in attempting to debug the 'error'. Pragmatically, unless the context prevents the subprocess from performing any trace that would get it into the approximate state, one would normally expect such approximations to be valid. For otherwise the specification would be expecting less of the initial state of the system when it is returned to than at the beginning.

5.1.3 Symbolic manipulations

Though CSP is primarily a notation for describing and reasoning about how processes interact, it can also describe how the internal states of these processes evolve. In other words, CSP processes sometimes contain, and communicate, data. We have seen one fairly typical example of this in buffer processes.

At the time of writing, FDR deals with this data by expanding any input on a channel into an alternative choice with one communication for each element of the underlying type. Thus the version of $COPY$ with data has $N + 1$ states where there are N members of the type of *left* and *right*. In the general case there seems no alternative to this approach, since conditional constructs, etc., allow one to create complex patterns of behaviour on a single channel. However it is frequently the case, as in the buffer case, that data (or at least most data) does not alter the control-flow of a process: it is treated as tokens, perhaps with operations carried out on it.

There appears to be considerable potential for reducing state explosion by recognising this fact and adding the capability for including symbolic representations of data in the transition systems used: inputs will not have to be expanded as above, and outputs will become abstract representations of the data they carry.

5.1.4 Specification operators

The CSP operators currently implemented are those that have convenient operational semantics over standard transition systems. Though, as we have remarked, these include some constructs whose main role is to allow us to construct specification-like processes, there are other constructs which it would be useful to have for specifications, that do not fit into this category.

Obvious ones include $\sqcup$ (least upper bound), which can be thought of as *full* conjunction (as opposed to $\|$ which acts as *trace* conjunction) and P/s (after). We have identified various others. These are characterised by being only, or most conveniently, implemented as direct transformations on normal forms. We propose to implement a variety of these in future versions of FDR. For them to be used at the implementation side (and there are good reasons for wanting this facility) it will be helpful to be using the extended form of transition systems identified above. This feature has many of the same needs as compression by normalisation.

A related issue is the possibility of creating finite-state representations of infinitary (typically *fairness*) specifications. The idea here is to create a system where we specify that all infinite traces must pass infinitely often though a specified set of nodes. It should be possible to present such infinitary specifications rather more compactly than most of their conventional finite approximations. This will require a revised form of normalisation.

5.2 Widening scope

5.2.1 Time and priority

The tool we have described deals only with the untimed version of CSP, where we are concerned with the relative order of events but not with exactly when they occur. Many concurrent systems developments are concerned with time: either real, continuous time or clocked time (in synchronous VLSI and similar applications). This has led to a variety of timed process algebras (including Timed CSP) and timed model-checking work.

Applications of Timed CSP have demonstrated the desirability of having mechanisms (such as Timewise Refinement [22]) for moving between the timed and untimed theories – either for different parts of a system, or during phases of a development. This creates the desire that FDR might be extended to deal with timed systems. One can identify three models of time such a version might operate.

- Real, continuous time as in Timed CSP.
- Clocked, or checkpointed, time where a special event is assumed to fire regularly, but where the relative order of events between checkpoints remains relevant.
- Discrete time, where there is a regular clock and process state only exists at the ticks, with multisets of events between them.

The first of these carries the obvious problem that there are infinitely many points in real time to worry about in any interval of non-zero length. It is not tractable in general, though results have been proved [23, 24] which show that, under certain conditions, a sufficiently fine finite granularity will do.

The second can be viewed as approximating the first. We have implemented a prototype system extended by a priority mechanism that makes this type of reasoning sensible. The basic principle we follow is that, at the highest level, most non-time events should have priority over time. A variety of examples have been studied.

The third is what is probably required for reasoning about synchronous VLSI at the cycle level. By treating the events between ticks as bags, and not recording intermediate state, one can expect to generate less states than in the asynchronous cases.

5.2.2 Non-CSP state machines

CSP specifications, and the theory of refinement we have developed, make as much sense for systems developed in other notations than CSP. Indeed the initial uses of the prototype system by Inmos used a variety of input formats for the state machines under consideration. We expect to deal with input derived from a number of non-CSP sources, including VHDL and LOTOS.

Acknowledgements

I have not written a single line of the code of FDR. Most of the credit, for implementing my ideas and many of their own, therefore goes to Dave Jackson, Michael Goldsmith, Bryan Scattergood and Jason Hulance. We owe much to the initial stimulus from Inmos, in particular Geoff Barrett and Victoria Griffiths, and to our various users and customers.

Partial funding for this work has come from ONR, ESPRIT and Inmos.

References

[1] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985.

[2] A.W. Roscoe, *Unbounded Nondeterminism in CSP*, in 'Two Papers on CSP', PRG Monograph PRG-67. Also Journal of Logic and Computation **3**, 2 pp131-172 (1993).

[3] Barrett, G., *The fixed-point theory of unbounded nondeterminism*, Formal Aspects of Computing 3 pp110-128 (1991).

[4] G. M. Reed and A. W. Roscoe, *Metric spaces as models for real-time concurrency,* Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics, Springer LNCS 298, 1987

[5] G. Lowe, *Priorities and probabilities in Timed CSP*, Oxford University D.Phil thesis, 1993.

[6] L. Jategoankar, A Meyer and A.W. Roscoe, *Separating failures from divergence*, In preparation.

[7] S.D. Brookes and A.W. Roscoe, *An improved failures model for communicating processes,* in Proceedings of the Pittsburgh seminar on concurrency, Springer LNCS 197 (1985), 281-305.

[8] S.D. Brookes, *A model for communicating sequential processes*, Oxford University D.Phil thesis, 1983.

[9] P.C. Kanellakis and S.A. Smolka, *CCS expressions, Finite state processes and three problems of equivalence*, Information and Computation **86**, 43-68 (1990).

[10] A.W. Roscoe, *Topology, Computer Science and the Mathematics of Convergence*, in Topology and Category Theory in Computer Science (Reed, Roscoe, Wachter, eds) OUP 1991.

[11] A.W. Roscoe, *An alternative order for the failures model*, in 'Two papers on CSP', technical monograph PRG-67, Oxford University Computing Laboratory, July 1988. Also appeared in Journal of Logic and Computation **2**, 5 pp557-577.

[12] A.W. Roscoe, *Denotational semantics for occam,* in Proceedings of the Pittsburgh seminar on concurrency, Springer LNCS 197 (1985), 306-329.

[13] M.H. Goldsmith, A.W. Roscoe and B.G.O. Scott, *Denotational Semantics for Occam II*, Oxford University Computing Laboratory technical monograph PRG-108 (1993). Also to appear in Transputer Communications.

[14] J.B. Scattergood, *A basis for CSP tools*, To appear as Oxford University Computing Laboratory technical monograph, 1993.

[15] C.A.R. Hoare, *Communicating Sequential Processes*, CACM **21**, 666-677 (1978).

[16] R.S. Bird and P. Wadler, *An introduction to functional programming*, Prentice-Hall 1988.

[17] M.D. May, G. Barrett and D. Shepherd, *Designing chips that work*, in *Mechanised Reasoning and Hardware Design*, M.J.C. Gordon and C.A.R. Hoare, eds (Prentice-Hall, 1992).

[18] A.W. Roscoe, *Occam in the Specification and Verification of Microprocessors* Phil Trans R. Soc. Lond A (1992) **339**, 137-151. Also in *Mechanised Reasoning and Hardware Design*, M.J.C. Gordon and C.A.R. Hoare, eds (Prentice-Hall, 1992).

[19] A.W. Roscoe *Developing and verifying protocols in CSP*, Proceedings of Mierlo workshop on protocols, published by TU Eindhoven.

[20] A.W. Roscoe and Naiem Dathi *The pursuit of deadlock fre edom,* Information and Computation **75**, 3 (December 1987), 289-327

[21] J.R. Burch, E.M. Clarke, D.L. Dill and L.J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Proc. 5th IEEE Annual Symposium on Logic in Computer Science, IEEE Press (1990).

[22] S.A. Schneider, *Timewise refinement for communicating processes*, proceedings MFPS 9, LNCS to appear, 1993.

[23] R.Alur, C. Couroubetis and D.L. Dill, *Model-Checking for real-time systems*, in Proceedings of Symposium on Logic in Computer Science, pp414-425.

[24] D. M. Jackson, *Logical verification of reactive software system*, Oxford University D.Phil Thesis, 1992.