Fairness analysis through priority

A.W. Roscoe, Philip Armstrong and Philippa Hopcroft

No Institute Given

Abstract. We report on the extension of the CSP-based refinement checker FDR to encompass a prioritisation operator as envisaged in [23]. This is embedded into the tool using similar technology to the well-known *chase* operator. We show how it can be used to analyse systems under what we term *unstable failures*, in which the usual notion of failure is augmented by a *fair* notion of acceptance along what would previously have been characterised as a divergence. This is important in practical application to the operation of Verum's ASD:Suite.

1 Introduction

Hoare's process algebra CSP, as explained in [11, 22, 23] treats the actions a process can perform alike, except that while ordinary visible communications in the alphabet Σ require the agreement of the external environment to occur, the two special actions τ and $\checkmark$ (the latter a visible signal of successful termination) do not. In particular, processes in these versions of the language cannot express a preference between whatever actions may be enabled in a given state.

Priority means preferring some actions that a system can perform to others. This preference can be based solely on the event name. In other cases it might depend on which process is performing the event, or on some operator like a prioritised version $\leftarrow\!\!\!\!\!\Box$ of $\Box$ within a process.

Over the years there have been a number of versions of CSP with added priority, for example [7, 15, 14], and languages such as occam [12] which have incorporated priority operators into the CSP model of interaction. Priority can be introduced either by prioritised analogues of existing process constructs such as $\Box$ (external choice) and $\|$ (parallel composition), or by assigning priorities to action labels in $\Sigma^{\tau\checkmark} = \Sigma \cup \{\tau, \checkmark\}$.

In this paper we will consider only the second of these options in detail: an operator $\mathbf{Pri}_{\leq}(P)$ where $\leq$ is a representation of an allowable priority order on $\Sigma^{\tau\checkmark}$ and P is a process. In every state of the process P it only allows actions that do not violate $\leq$'s priority specification: in general $\leq$ is partial order on $\Sigma^{\tau\checkmark}$ where the special actions τ and $\checkmark$ are incomparable with each other and greater than all members of Σ that are not maximal in the order. $\mathbf{Pri}_{\leq}(P)$ never allows P to perform action x in a state where there is a greater one available. We will discuss this operator in more detail in Section 3, after first recalling some background.

While that operator limits what the process P can do based on priority, it neither expects P itself to be a prioritised object nor exports any prioritisation amongst the actions it allows: at least at the level of operational semantics it makes sense to consider it in a world of ordinary, unprioritised, semantics.

The same is not true of a prioritised external choice that we might write $P \boxminus Q$. The semantics of $(a \to P) \boxminus (b \to Q)$ has to understand that a is preferred to b, but both remain available. As a result $\boxminus$ would be much harder to incorporate into a tool like FDR [21, 1] that has a lot invested in a particular model of transition system. This difficulty coincides with the fact that any operational or behavioural semantic model has to be changed to include $\boxminus$ in the language.

It is in fact possible to implement prioritised parallel operators in terms of $\mathbf{Pri}_{\leq}(\cdot)$. This is easy to see for the standard CSP parallel $P\ {}_{A}\|_{B}\ Q$ in which P and Q must respectively communicate all events in A and B, synchronising on $A \cap B$: we simply apply the priority operator outside the parallel operator (perhaps a many-way one) giving desired precedence to the component processes' actions. This would probably be done inside any hiding of the network's internal actions.

In Section 4 we describe the implementation of $\mathbf{Pri}_{\leq}(P)$ operator within FDR and give examples of its use and performance. We discuss the interaction with FDR's state compression functions.

There are many different notions of fairness in the literature of concurrent systems, but one that is particularly relevant to CSP with its emphasis on handshaken communication and failures-style model is that of *fair acceptance* discussed in Section 6: if a process has an infinite sequence of τ events possible, but while it is performing then it is offered some set of visible events A such that infinitely many of the states it passes through could perform as alternatives to τ, then one of them is accepted, so the divergence does not occur. While at first sight fairness and priority seem to be at odds with one another, we show that priority is key to building acceptance fairness into FDR checks through an extension of the usual stable failures model to include *unstable failures* that encompass this idea, as discussed in Section 7. Mechanisms for checking unstable failures refinement using priority are introduced in Section 8.

We illustrate this style of reasoning in Section 9 via an application to an industrial tool: Verum's ASD:Suite for the development of embedded software, into which FDR is integrated.

2 Background: CSP and its semantics

The CSP process algebra is based on the concept of instantaneous actions handshaken between a process and its environment, whether that environment consists of processes it is interacting with or some notional external observer. It enables the modelling and analysis of patterns of interaction. The books [11, 22, 23, 26] all provide thorough introductions to CSP. The main constructs that we will be using in this paper are set out below.

- The constant processes STOP, SKIP and **div** which respectively do nothing, terminate immediately with the signal $\checkmark$ and diverge by repeating the internal action τ.
- $a \to P$ *prefixes* P with the single communication a which belongs to the set Σ of normal visible communications. Similarly $?x : A \to P(x)$ offers the choice A and then behaves accordingly.
- CSP has several *choice* operators. $P \mathrel{\Box} Q$ and $P \sqcap Q$ that respectively offer the environment the first visible events of P and Q, make an internal decision via τ actions whether to behave like P or Q.
  The asymmetric choice operator $P \rhd Q$ that offers the initial visible choices of P until it performs a τ action and opts to behave like Q. In the cases of $P \mathrel{\Box} Q$ and $P \rhd Q$, the subsequent behaviour depends on what initial action occurs.
- $P \setminus X$ behaves like P except that all actions in X become (internal and invisible) τs.
- $P[\![R]\!]$ behaves like P except that whenever P performs an action a, the *renamed* process must perform some b that is related to a under the relation R.
- $P \underset{A}{\parallel} Q$ is a *parallel* operator under which P and Q act independently except that they have to agree (i.e. synchronise or handshake) on all communications in A. A number of other parallel operators can be defined in terms of this.

There are also other operators such as $P;\ Q$ (sequential composition), $P \bigtriangleup Q$ (interrupt) and $P\,\Theta_a\,Q$ (throwing an exception) that do not play a direct role in this paper.

It is always assumed that the meaning, or semantics, of a CSP process is the pattern of externally visible communication it exhibits. As shown in [22, 23], CSP has several styles of semantics, that can be shown to be appropriately consistent with one another. The two styles that will concern us in this paper are *operational* semantics, in which rules are given that interpret any closed process term as a labelled transition system (LTS), and *behavioural* models, in which processes are identified with sets of observations that might be made from the outside. By convention these behavioural models are chosen to be compositional under all operators including recursion.

An LTS models a process as a set of states that it moves between via actions in $\Sigma^{\tau\checkmark}$, where τ cannot be seen or controlled by the environment. ($\checkmark$ is regarded as visible but is not controlled by handshaking.) An external observer interacting with a process cannot always tell what state it is in. Note that LTSs are a sequential model in which all processes, even ones built from parallel operators pass through one state at a time. There may be many actions with the same label from one of their states, in which case (even if the label is in Σ) the environment has no control over which is followed.

The best known behavioural models of CSP are based on combinations of the following types of observation.

- *Traces* are sequences of visible communications a process can perform.
- *Failures* are combinations (s, X) of a finite trace s and a set of actions that the process can refuse in a *stable* state reachable on s. A state is stable if it cannot perform either of the actions τ and $\checkmark$ that the process can perform without the cooperation of the environment.[1]
- *Divergences* are traces after which the process can perform an infinite sequence of uninterrupted τ actions, in other words diverge.

These well-known models are

- $\mathcal{T}$ in which a process is identified with its set of finite traces;
- $\mathcal{F}$ in which it is modelled by its (stable) failures and finite traces;
- $\mathcal{N}$ in which it is modelled by its sets of failures and divergences, both extended by all extensions of divergences: it is *divergence strict*.

Note that traces, failures and divergences are all observations that can be made of a single behaviour of a process in *linear time*: unlike with LTS

[1] The action $\checkmark$ is sometimes included in refusal sets, and notional failures added corresponding to states where $\checkmark$ is possible: this issue is discussed in [23] but is irrelevant to the present paper.

semantics, the way that a process's behaviour branches as it evolves is not recorded. As described in [23], there is a range of other models based on other, usually richer, forms of linear behaviours. One that is important to this paper is *refusal testing*, in which we record not just one stable refusal at the end of a trace, but have the option to record one before each event of the trace as well as at the end. Refusal testing models have long (see [18]) been recognised as being relevant to priority, for reasons we will discuss in the next section. However we show in this paper that (unexpectedly) refusal testing models are not always sufficient, and the sometimes one needs to look at the yet more refined models in which the refusal information during and at the end of traces is replaced by *acceptance* or *ready* sets: the actual sets of events made available from stable states. These latter models, which sit at the extreme of what a CSP-style behavioural model can record, are sometimes called *acceptance traces* models.

There is an interesting relationship between priority and the modelling of timed systems. Semantic models for versions of CSP incorporating time – either continuous [20] or discrete [16, 17, 23] – usually bear a close relationship to refusal testing, since they involve recording what visible events at each time instant that the model records. It has long been recognised that in order to simulate discrete timed systems accurately on FDR it is necessary to give τ actions priority over the event (typically *tock*) representing the passage of time. With that goal, FDR has for several years (initially in prototype versions and then as officially released functionality) implemented a checking mode[2] specifically for that:

```
assert P [T= Q :[tau priority over]: A
```

analyses this trace refinement under a modified operational model in which no event in `A` can happen when the process can perform a τ. This is closely related to applying the $\mathbf{Pri}_{\leq}(\cdot)$ operator with an order in which the events in A are less than $\{\tau, \checkmark\}$, and all other members of Σ are

[2] That mode was designed for the CSP dialect *tock*-CSP, in which the event *tock* is used by the programmer, explicitly modelling the passage of time. More recently explicit support has been added for Timed CSP [2, 20, 26], where a real-time semantics is given for the CSP language with a $WAIT\ t$ construct which takes precisely t time units before terminating. That support is provided via a translation to *tock*-CSP as discussed in [16, 17, 23], and priority of τ over *tock* is still required there. The priority construct used there is, however, couched in terms of the one described in the present paper, which essentially supersedes the `[tau priority over]` mode. This is so particularly as (i) it works for all models and not just traces and (ii) can be used at any level in the process that is outside recursions.

incomparable to all events. However it can only be applied at the top semantic level as it is imposed by the checking mode.

3 A priority operator

The $\mathbf{Pri}_{\leq}(P)$ operator discussed in the introduction is slightly more general than the one described in Chapter 20 of [23]. The one described there was parameterised by a partial order on a *subset* of Σ, in which all events in that subset were assumed to be strictly less than τ and $\checkmark$.[3] It behaves in exactly the same way as our operator in which the order is extended to the whole of $\Sigma^{\tau\checkmark}$ by placing the events not in the domain of the original order as additional maximal elements all incomparable to every other action in $\Sigma^{\tau\checkmark}$ (as discussed for the τ priority model above). This order meets the restrictions described earlier, and which we will explain shortly.

The operational semantics of $\mathbf{Pri}_{\leq}(\cdot)$ are easier to understand than its abstract behavioural semantics. They do not, however, fit into the framework described as "CSP-like" in [23, 24], because they require negative premises: an action cannot occur unless all actions of higher priority are impossible.

The following two SOS (Structured Operational Semantics) rules are necessary unless our new operator is going to turn the philosophy of CSP upside down, since the process P is allowed to perform $\checkmark$ and τ without the co-operation of its environment, which in this case is our new operator:

$$\frac{P \xrightarrow{\tau} P'}{\mathbf{Pri}_{\leq}(P) \xrightarrow{\tau} \mathbf{Pri}_{\leq}(P')} \qquad \frac{P \xrightarrow{\checkmark} P'}{\mathbf{Pri}_{\leq}(P) \xrightarrow{\checkmark} \Omega}$$

It is these rules that imply that the actions τ and $\checkmark$ must be maximal in the order $\leq$ that any instance of $\mathbf{Pri}_{\leq}(P)$ uses: no other action can prevent them. In fact we would expect the following rule to be the only operational semantic rule needed to govern this operator's behaviour, provided that we augment it in the case $x = \checkmark$ by the convention that the result of any $\checkmark$ action is the terminated process Ω:

$$\frac{P \xrightarrow{x} P' \wedge \forall\, y \neq x . x \leq y . P \not\xrightarrow{y} \cdots}{\mathbf{Pri}_{\leq}(P) \xrightarrow{x} \mathbf{Pri}_{\leq}(P')}$$

So this rule must generate the same τ and $\checkmark$ actions as the two specific rules above. It implies that any action maximal in the partial order, including τ and $\checkmark$, can proceed unfettered.

[3] So, unlike here, a member of Σ with priority greater than another must itself have lower priority than τ.

We assumed a further property of $\leq$, namely that every non-maximal event is less than τ and $\checkmark$. We can see the need for this from two of the "laws" of CSP and the operational semantics of other CSP operators. The first is not a law in itself but is derivable from standard laws: the principle that a process P preceded by a single τ action – we can write this τP – is equivalent to P in every model. This does not in itself imply that τ has to have high priority, since in τP it is not in competition with other events. However, consider the processes $P1 = (a \to Q) \square (b \to Q)$ and $P2 = (a \to Q) \square (\tau(b \to Q))$ where the identity of Q is unimportant. Our principle implies they are equivalent. The first state of the operational semantics of $P2$ has two actions: a leads to Q and τ to $P1$.

Suppose we were to apply our priority operator to $P1$ and $P2$ in a context where $a < b$. $\mathbf{Pri}_{\leq}(P1)$ cannot perform a as its first visible action, so we can infer that $\mathbf{Pri}_{\leq}(P2)$ cannot either. It is this that allows us to infer that if a process can perform a τ action together with a second action a that b, not immediately available, would preclude, then it is not appropriate to allow a. Thus if a has a lower priority than b then it must have a lower priority than τ since that might enable b.

The principle that explains the priority of $\checkmark$ is the law which explains that it is not controllable:

$$P \square SKIP = P \rhd SKIP$$

named $\square$-$SKIP$-resolve in [22, 23]. If P is an initially stable process such as $P1$ which has no initial $\checkmark$, then the operational semantics of the left-hand side of this equation has a range of visible initial actions and $\checkmark$, while the right-hand side has exactly the same initial actions in Σ and the τ introduced by the operational rule.

If applying $\mathbf{Pri}_{\leq}(\cdot)$ to the left- and right-hand sides is going to give equivalent results, we can infer that τ and $\checkmark$ must have equal priority.

While it would make perfect sense *operationally* to drop the restrictions on $\leq$, the above arguments show that the $\mathbf{Pri}_{\leq}(P)$ operator would not make sense in any of CSP's existing abstract models despite our aim that it would make sense over refusal testing models. Another way of viewing the restrictions is to say that τ and $\checkmark$ are unaffected by the priority order, and that non-maximal events can only happen from stable states (ones with no τ or $\checkmark$).

We cannot expect our prioritisation operator to respect a CSP model that does not tell us which events a process performs happen from stable states, and whether all Σ-events less than a given event are then refused. The traces model certainly does not do this because its observations are completely independent of whether the process is stable or not. While failures-based models would seem to satisfy this requirement – as failures occur in stable states and tell us what these states refuse – they do not. Consider the pair of processes $(a \to b \to STOP) \rhd (a \to STOP)$ and $(a \to STOP) \rhd (a \to b \to STOP)$. These divergence-free processes have identical failures, but imagine applying a priority operator to them where $a < b$. In each case the $a \to \cdot$ that appears to the left of $\rhd$ is prevented because τ is an alternative. So only the other a is allowed, meaning that the results of the prioritisation are different: one can perform b and one cannot. We conclude that it is not enough to know information about stable states only at the ends of traces; rather we need to know about stability and the refusal of high-priority events earlier in traces as well.

The refusal testing model described briefly in the introduction does distinguish these two processes, because they have different behaviours beginning $\langle \Sigma \setminus \{a\}, a\rangle$, so the fact that $\mathbf{Pri}_{\leq}(\cdot)$ maps them to processes with different traces is OK if we are testing that refusal testing is respected. Several variations on the refusal testing model, and a richer one in which exact *ready* or *acceptance* sets are recorded on the stable states in a trace, are detailed in Chapters 11 and 12 of [23]. In the simplest of these, the *stable refusal testing model* $\mathcal{RT}$, the behaviours recorded of a process are all of the forms

- $\langle X_0, a_1, X_1, \ldots, X_{n-1}, a_n, X_n\rangle$ and
- $\langle X_0, a_1, X_1, \ldots, X_{n-1}, a_n, X_n, \bullet, \checkmark\rangle$

where $n \geq 0$ and each X_i is either a refusal set (subset of Σ) or $\bullet$ (indicating that no refusal was observed).

The refusal testing value of a process P can tell us what traces are possible for $\mathbf{Pri}_{\leq}(P)$: P can only perform an action a that is not maximal in $\leq$ when all greater actions (including τ) are impossible. In other words the trace $\langle a_1, \ldots, a_n\rangle$ is possible for $\mathbf{Pri}_{\leq}(P)$ if and only if

$$\langle X_0, a_1, x_1, \ldots, X_{n-1}, a_n, \bullet\rangle$$

is a refusal testing behaviour, where X_i is $\bullet$ if a_{i-1} is maximal, and $\{a \in \Sigma \mid a > a_{a-1}\}$ if not (even if that set is empty so a_{n-1} is less than only τ and $\checkmark$).

It came as a surprise to us, however (particularly given what the first author wrote in [23]) to discover that there are cases where the refusal components of refusal testing behaviours of $\mathbf{Pri}_{\leq}(P)$ can not be computed accurately from the corresponding behaviour of P. This is because

$\mathbf{Pri}_\leq(P)$ can refuse larger sets than P: notice that if P offers *all* visible events, then the prioritised process refuses all that are not maximal in $\leq$.

Consider the processes

$$DF1(X) = \bigsqcap\{a \to DF1(X) \mid a \in X\}$$

$$DF2(X) = \bigsqcap\{?x : A \to DF1(X) \mid A \subseteq X, A \neq \emptyset\}$$

These are equivalent in the refusal testing models: each has all possible behaviours with traces in Σ^* that never refuse the whole alphabet Σ.

Now consider $P1 = DF1(\{a, b\}) \mid\mid\mid CS$ and $P2 = DF2(\{a, b\}) \mid\mid\mid CS$ where $CS = c \to CS$. Clearly these are also refusal testing equivalent. Now suppose $\leq$ is the order in which $b > c$ and a is incomparable to each of b and c. We ask the question: is $\langle\{c\}, a, \bullet\rangle$ a refusal testing behaviour of $\mathbf{Pri}_\leq(Pi)$?

When $i = 1$ the answer is "no", since whenever $P1$ performs the event a the set of events it offers is precisely $\{a, c\}$ (it can also offer $\{b, c\}$). On the other hand, $P2$ can choose to offer $\{a, b, c\}$: in this state the priority operator prevents c from being offered to the outside, meaning that $\mathbf{Pri}_\leq(P2)$ can be in a stable state where a is possible but c is not: so in this case the answer is "yes". This demonstrates that we need more information than refusal testing of Pi to calculate the refusal testing behaviours of $\mathbf{Pri}_\leq(Pi)$.

On close inspection this example tells us that $\mathbf{Pri}_\leq(\cdot)$ is only compositional for refusal testing when the structure of $\leq$ is such that whenever a and b are incomparable events in Σ and $c < b$ then also $c < a$. This means that the order has to take one of two forms:

- A linearly ordered list of collections of equally prioritised events, the first of which contains $\{\tau, \checkmark\}$.
- A linearly ordered list of collections of equally prioritised events, the first of which is exactly $\{\tau, \checkmark\}$, together with a further collection of events that are incomparable to the members of the first two of these collections and greater than the rest.

The second of these includes the order used for the timed priority model, in which the only prioritisation is that $\{\tau, \checkmark\}$ have greater priority than the time event(s), typically $\{tock\}$.

Both exclude some interesting applications of priority, including the main example in this paper.

These issues disappear for the acceptance traces model $\mathcal{FL}$ and its variants, which are therefore the *only* CSP models with respect to which our priority operator can be defined in general.

With respect to this model, the semantics of $\mathbf{Pri}_{\leq}(P)$ are the behaviours

$$\{\langle A_0, a_1, A_1, \ldots, A_{n-1}, a_n, A_n\rangle \mid \langle Z_0, a_1, Z_1, \ldots, Z_{n-1}, a_n, Z_n\rangle \in P\} \cup$$
$$\{\langle A_0, a_1, A_1, \ldots, A_{n-1}, a_n, \bullet, \checkmark\rangle \mid \langle Z_0, a_1, Z_1, \ldots, Z_{n-1}, a_n, \bullet, \checkmark\rangle \in P\}$$

where in every case one of the following holds:

- a_i is maximal under $\leq$ and $A_i = \bullet$ (so there is no condition on Z_i except that it exists).
- a_i is not maximal under $\leq$ and $A_i = \bullet$ and Z_i is not $\bullet$ and neither does Z_i contain any $b > a_i$.
- Neither A_i nor Z_i is $\bullet$, and $A_i = \{a \in Z_i \mid \neg \exists\, b \in Z.b > a\}$,

and in each case where $A_{i-1} \neq \bullet$, $a_i \in A_{i-1}$.

Notice how we are able, when P offers the set Z, to calculate the set that $\mathbf{Pri}_{\leq}(P)$ offers: the set $\{a \in Z \mid \neg \exists\, b \in Z.b > a\}$.

The above definition can easily be adapted to all the variants set out in [23] of the model $\mathcal{FL}$ in which the exact sets of events offered from stable states are recorded through the trace, rather than refusal sets.

4 Implementation in FDR

As described, for example, in [22, 23], FDR adopts a two-level implementation strategy for CSP. Its compiler typically identifies a number of component processes which run in parallel combinations in the complete system. The compiler then reduces each of these components to an *explicit* state machine: a list of states and transitions. It also devises sets of rules, called *supercombinators* by which combinations of actions of these components become actions of the whole. Full details can be found in [23]. Aside from these, it also handles operators on state machines which are declared as `transparent` or `external` within CSP scripts: the former are operators that are not intended to change the behavioural semantics of the machine they are applied to, the latter may change the semantics. Examples are state compression operators such as `normal` and `diamond`, and the `chase` operator which sits somewhere between `transparent` and `external` since while it can change semantics, it is usually used in places where it does not. (It can be declared as either.) `transparent` and `external` operators thus transform the state machine representing a process by direct manipulation of state machine representations, the only difference being whether these transformations are intended to leave the abstract semantics unchanged or not.

There is no reason in principle why $\mathbf{Pri}_{\leq}(P)$ should not be implemented as a low-level operator within a particular component. We have yet to find a convincing example of why this is useful, however, and at the time of writing no such implementation exists. There would, however, be practical problems[4] if the operator was recursed through: a recursive definition of a process P in which an instance of P occurs within the new operator.

It would not, however, be possible to embed $\mathbf{Pri}_{\leq}(P)$ within the supercombinator framework without a complete revision of the latter. For this only enables FDR to calculate actions of a multi-component systems from the actions that the components positively have and does not allow for the sort of negative premises that we saw in the operational semantics of our new operator above. In the language of [23, 24], this is because $\mathbf{Pri}_{\leq}(P)$ is not a *CSP-like* operator as described in [24, 23]

The natural mode of implementing our new operator is therefore as `external`, and this is what we have done. It acts in a way similar to `chase`: as a wrapper or environment for the state machine it is acting on, which calculates the entire set of initial actions of its arguments before deciding what to do with these.[5]

In common with `chase`, the result of applying our `priority` operator can be treated like any other state machine within FDR, for example becoming a component of a supercombinator-based parallel combination or having other `external` or `transparent` operators applied to them.

Because of the relative complexity of representing a general partial order to FDR, we took the decision to restrict the form of order allowed to specifying a list $\langle A_0, \ldots, A_n \rangle$ of disjoint subsets of Σ, individually groups of equal-priority events, with decreasing priority through the list. Any members of the first subset A_0 have equal priority to $\checkmark$ and τ, and any member of Σ that lies outside $\bigcup_{i=0}^{n} A_i$ is incomparable to all other members of the order. We have not yet found a practical example that requires more generality.

FDR 2.93 therefore implements the new `external` operator `prioritise`, which has two modes of use:

[4] The problem alluded to that it would be difficult to "close off" the exploration of a low-level component with this type of recursion. It is similar to documented existing problems with recursing through some standard CSP operators such as parallel and hiding [8, 24].

[5] `chase` determines if the argument has any τ actions, and if it does simply follows one without making this τ visible on the outside. Thus the result of applying `chase` only has visible actions, and it only has these from states of the argument with no τ.

```
  prioritise(P,A0,A1,..,An)     prioritise(P,As)
```

where `As` is a list of subsets of Σ. The advantage of the latter is that `As` can be any expression for such a list, for example a list comprehension calculated within the program under consideration.

There are in fact two versions implemented, namely `prioritise` and `prioritise_nocache`, the difference being identical to the existing `chase` and `chase_nocache`. The cached version remembers the states it has applied prioritisation to and the transitions it has calculated for them, while the `nocache` version recalculated every time. We recommend that the `nocache` version is normally used, particularly when applied to a process with a large state space, since amount of memory consumed by the caching can be considerable.

4.1 Priority and compression

FDR implements a number of operators that take an LTS and attempt to construct a smaller LTS or *Generalised* LTS (GLTS) with the same semantic value. A GLTS is like an LTS except that information such as divergence, refusals and acceptances is included as explicit annotations to nodes rather than being deduced only from transitions.

The compressed (G)LTS can then replace the original in higher-level compositions as a way to address the state explosion problem. Most of these were designed primarily for failures and traces style model, and so are not necessarily accurate in refusal testing or acceptance traces. In other words, $compress(P)$ is not necessarily equivalent to P in $\mathcal{RT}$, $\mathcal{FL}$ and similar models. `diamond`, one of the most frequently used compressions, falls into this category. Indeed, the form of GLTS produced by some of the compressions does not contain enough information for the richer models.

Furthermore `normal`, which reduces a process to its normal form in the current semantic model, would be extremely complex in refusal testing and is not presently implemented: out initial impression is that it would probably be slow and ineffective in many cases. [It would be more natural in $\mathcal{FL}$ but has not yet been implemented.]

Nevertheless, all compressions applied in the definition of P in $\mathbf{Pri}_{\leq}(P)$ must preserve acceptance traces or refusal testing equivalence (dependant on $\leq$), for otherwise there is the danger that applying the priority operator to the revised P might not produce the same answer.

In part as a remedy for this problem, we have implemented the compression *divergence-respecting weak bisimulation* as defined in [23]. (This factors an LTS by the maximum weak bisimulation relation that does not identify any pair of states, one of which is immediately divergent and the other one not). This respects all CSP models and has the added advantage that, unlike some other compressions, turns an LTS into an LTS rather than a GLTS, whose present structure gives insufficient information for the implementation of priority. We will report separately on this implementation and weak bisimulation's place in the family of CSP compression functions.

5 Using `prioritise`

In this section we consider ways in which the new operator can be used in the modelling and implementation of some types of system, either extending what can be modelled in CSP or making modelling more natural and efficient.

5.1 Priority and time

As detailed in [23] and elsewhere, it is usual to adopt the principle of *maximal progress* when building process algebra models of timed systems that have the possibility of internal τ actions. This principle says that whenever a τ action becomes available, *some* action must happen immediately. So processes never hang around for any non-zero time interval with a τ available.

A good example of why this principle is necessary is provided by the case in which the outputs of one timed process are fed into the inputs of the other. If both of these processes are designed to wait for a communication if none is immediately available, we have to ask what happens when they are both willing to proceed. A simple case of this is provided by the following *tock*-CSP model of a one-place buffer, taken from Chapter 14 of [23]:

$$\begin{aligned} TCOPY2 &= \mathit{left}?x \to \mathit{tock} \to TCOPY2'(x) \\ &\quad \Box\ \mathit{tock} \to TCOPY2 \\ TCOPY2'(x) &= \mathit{right}!x \to \mathit{tock} \to TCOPY2 \\ &\quad \Box\ \mathit{tock} \to TCOPY2'(x) \end{aligned}$$

The natural way of composing two of these processes is via a timed analogue of the CSP chaining operator $\gg$: connect the output channel *right* of one to the input channel *left* of the other, hide the resulting communications and synchronise the two processes on the event *tock* that represents the regular passage of time.

Without the maximal progress assumption this process can perform as many *tock* events as it likes before the τ transferring a data item from left to right occurs, even though that τ has been available since after the first *tock*. This is unrealistic. Maximal progress can be implemented in this example via the operator `prioritise(P,{},{tock})`: `tock` is given lower priority than τ and $\checkmark$, with all other events being unaffected by priority. As stated in the introduction, this has the same effect as running the combination as the right-hand side of the assertion that FDR implements specifically for this purpose.

Having `prioritise` as an operator rather than a checking mode allows more generality, since for example we can create timed components that each have independent clocks, each driving its own version of maximal progress. Or we can take a timed component which is built using maximal progress and then hide *tock* or similar events so we get an *untimed* view of the process, or *rescale* the clock by arranging to hide all but the Nth, $2Nth$, $3N$th etc *tock*s.

5.2 Priority and tactics

Sometimes one builds a CSP of a system to test if certain sorts of state are reachable. Good examples of this are the models of puzzles such as peg solitaire, Sudoku and the knight's tour that can be found in [22, 23]. It is often desirable to establish strategies for how such state spaces are explored, and priority can help us do this.

One example of this is provided by Warnsdorff's algorithm for the knight's tour, which instructs the experimenter, when confronted with a choice of next moves, always to choose a place to move to with a minimum of still-free neighbours.[6] A natural way of implementing this in CSP is to have a process for each square on the board which always knows if it has ever been occupied and, if not, how many free neighbours it still has. In the coding we allude to here there is also a single process representing the knight, which is parameterised by its present position. We then choose events $move.n.p$ to represent the act of moving the knight to point p on the board (p will be a pair of integer co-ordinates) where n is the number of free neighbours of p. Each square then monitors not only moves to it, but also moves to its neighbours so it can maintain its state correctly.

[6] There may sometimes be a number of such minimum choices, and there are choices that follow Warnsdorff and do not find a complete tour. So it is more of a heuristic than an algorithm. However for square boards that have a knight's tour, there is one that can be found with this heuristic.

The obvious way of implementing this is by giving priority to $move.n.p$ events according to n: the smaller it is, the higher priority.

Section 20.1 of [23] explains an alternative coding of this in which the squares have to agree that there is none amongst those neighbouring the knight that is unused and has (successively) $0, 1, \ldots, r-1$ free neighbours before one with r can be moved to. In effect this simulates a prioritised execution without itself using priority. However the coding using `prioritise` is certainly more efficient and natural. For example, in an experiment using FDR's standard breadth-first search,[7] the version with `prioritise` solved an 8×8 board in 16 seconds, as opposed to 62 seconds for the version with the [23] coding. If the board gets much bigger than this, the number of states possible following Warnsdorff using BFS becomes too great to search conveniently, and at this point a second advantage of using `prioritise` becomes apparent. It is very simple to experiment with refinements of the original order, meaning that FDR less frequently has a choice of equally-ranked options to follow.

Files exploring all these options are available for download via the web site of [23][8], along with others analysing the problem of *long and skinny* knight's tours (of a $M \times N$ board where M is small) as discussed in [13].

Note that Warnsdorff's algorithm strictly reduces the extent to which one searches for a knight's tour. The fact that FDR finds solutions using it (and even considerably tightened versions of it) is a testament to the effectiveness of the heuristic that it represents. If you reverse the priority order – so that you choose a target with the *largest* number of free neighbours – then there is no solution in the 8×8 case if the knight starts in a corner.

A similar and yet subtly different case is given by Sudoku. Here we have a choice of which square to fill in next, and this can also be made by priority: perhaps one of those with the least number of options that are not immediately blocked by the same symbol residing in the same row, column or box. The difference here is that we can make a choice of square without restricting the range of solutions that may be found: a solution is simply an assignment of symbols to squares, not a path. So here we can use priority on choice of square without any danger of making the puzzle insoluble.

[7] In fact, depth-first search is significantly more effective in conjunction with versions of Warnsdorff's algorithm.

[8] `www.cs.ox.ac.uk/ucs`

5.3 Explicitly prioritised models

The `prioritise` operator can potentially be used to improve the CSP models (and hence FDR verification) of models of concurrent system that depend on priority for their semantics. Several examples of this are provided by *statecharts* (of which there are a number of variants with different priority assumptions). Some examples are given below.

- Statemate statecharts [10, 25] have a two-level timing model in which small time steps continue for as long as necessary until no more progress occurs, at which point a big time step occurs. This is closely analogous to the maximal progress assumption discussed above, and can be modelled by giving the actions associated with progress priority over one that triggers a big time step.
- Statecharts provide a hierarchical model of state machine, in which individual states of a high-level machine can contain subsidiary machines. One generally wishes to prioritise high-level actions over low-level ones where both are possible.
- Some models of statechart provide very precise rules for selecting which action or actions of a state or parallel ("and") collection of states perform, the motivation presumably being to make the behaviour of a statechart model deterministic.

We have yet to make use of priority for significant examples of this type.

6 Fairness and priority

There are many different concepts of *fairness* in the literature on semantics and verification. Broadly speaking it means the assumption that if a machine has the option to do something infinitely often during an infinite execution, then it will eventually take this opportunity. This may or may not be a reasonable assumption depending on circumstances.

Historically speaking, FDR has not concentrated on the analysis of fairness as much as some other tools which typically analyse fairness through the medium of *Büchi Automata*. That is because general fairness is not necessary to calculate the refinement relations that FDR is built for. Fairness is highly relevant, however, to properties specified in languages like LTL which allow a range of eventuality properties to be expressed.

Fairness is often expressed in terms of state spaces, for example a given state must be visited infinitely often in an infinite behaviour. There are two features of CSP that limit this type of reasoning. Firstly, semantic models cast in terms of observed behaviours give us no direct knowledge of what states a process passes through: for any process there is a semantically equivalent one where no state is ever visited twice. Secondly, processes only perform normal visible actions when their environments permit, meaning that any judgement of what is and is not fair must be made relative to what offers the environment was making through time.

In this paper we will concentrate on a particular sort of fairness that matches well to the CSP model, *acceptance fairness*, as described below.

When we observe a CSP process we see it perform events, joined together to form traces. We see what events our process refuses when we offer them, and we sometimes choose to record when it diverges by performing an infinite sequence of internal actions. The traditional view is that when a process is diverging we do not see what it offers or refuses. After all, it is usual to regard divergence as a disastrous behaviour that swallows all others in CSP models that record it, and a diverging process does not sit still and offer a single set of events.

However, from the standpoint of fairness, if a process has a set of visible actions A available alongside the τ's making up a divergence infinitely often as the divergence progresses, and the environment is offering actions from A throughout this time, then it is reasonable to postulate that one of these must actually occur. In other words, the divergence does not actually occur at all, and in some sense our process cannot refuse sets that intersect with A during the course of this divergence.

In other words, we might say that any particular divergent execution has an acceptance set associated with it: the events that are available from infinitely many of its states. It therefore refuses any set that does not intersect with this set. It is reasonable to term any such set an *unstable refusal*, and to get a corresponding notion of *unstable failure*. Of course we would want to include traditional stable failures in the representation of a process, and therefore define the union to be the *extended failures* of a process. We can then choose to compare this against a failures specification, and correspondingly reduce the set of divergence traces for a failures-divergences specification.

It is this that we term acceptance fairness. As we will see, it is possible to assume acceptance fairness only relative to certain accepted events A, and even differentiate to some extent between different sorts of τ actions in the associated choices. But first of all assume that we want to analyse this concept of unstable failures relative to all Σ events and all τs, and that we are trying to decide if the extended failures of a process P are contained in the stable failures allowed by a specification S:

$S \sqsubseteq_{UF} P$

A similar, and yet subtly different, notion of fairness has been examined in [6]. In this, a process does not refuse a set X if, from any state P along a chain of τs, there is further state P' reachable after zero or more further τs from which a member of X can be performed. This therefore requires not only that events offered in conjunction with τs infinitely often are fairly accepted, but that the τs leading to offers are also treated fairly. To illustrate this, consider the process $ND = ND \sqcap a \rightarrow ND$ whose operational semantics begins with a choice of two τs: one leading to ND and one to $a \rightarrow ND$.

Under our concept of acceptance fairness, this process can unstably refuse a since it can always choose the left-hand τ and therefore never reach a state where a is available. In the equivalences investigated in [6] this process cannot refuse a because it can always reach the state where a is offered.

In the next two sections we will examine how unstable and extended failures are computed, and how we can formulate FDR checks of them using priority.

In the rest of this paper we make two simplifying assumptions: firstly, and like FDR, we assume that the set Σ of all ordinary events is finite. There is little prospect of dropping it practically; to drop it theoretically one would have to resolve an ambiguity that arises in the definition of unstable refusal when one considers an infinite set. Secondly, to avoid some special casing for the termination event $\checkmark$, we assume that the constructs ; and *SKIP* are not used, so processes do not perform $\checkmark$. There should be no problem in later extending our work to encompass these constructs.

7 Calculating unstable failures

Deciding whether $S \sqsubseteq_{UF} P$ will normally be done in two parts: firstly performing the check $S \sqsubseteq_F P$ which tests whether the traces and stable failures of P conform to S. All we therefore have to do is work out whether the unstable failures of P conform to S. Before we show how to do this we need to examine the concept of an unstable failure more closely.

The unstable failures (s, X) of P can expected to manifest themselves as divergences in a state exploration: divergences during which X is in some sense offered P during an infinite final period after the conclusion of the trace s. To be an unstable failure, there would have to be a divergence after s in which members of X are only available (as alternatives to τ) in finitely many states.

The unstable failure $(s, \emptyset)$ would correspond to any actual divergence, and (s, Σ) would correspond to a divergence on which (using the finite alphabet assumption) eventually there are no Σ actions offered through a tail of the behaviour.

It is clear how we can extract the unstable failures of any process from its operational semantics: simply inspect the divergent trajectories of our process P. These are the infinite sequences of states P_i of P's operational semantics, with $P_0 = P$ and actions $P_i \xrightarrow{x_i} P_{i+1}$ such that only finitely many of the x_i are not τ. The trace associated with this trajectory is the sequence of these non-τ x_i, and an unstable refusal is any subset X of Σ whose members are only directly possible for finitely many of the P_i.

The only fairness we are expecting arises from interactions with the environment: every finite and infinite trajectory of the operational semantics remains possible under some circumstances. It is this case that corresponds well both to our industrial example and the treatment with priority below. As stated above, we are only considering *acceptance* fairness.

An obvious question is whether we can turn unstable failures (either by themselves or in combination with traces and/or stable failures) into a compositional semantic model.

The answer to this appears to be "no". At least two of the standard CSP operators cause problems here. The first – and more obvious – is hiding. It is clear that in order to know the offers that are made along divergences of $P \setminus X$, we need to know what offers are made alongside X events that constitute a divergence of $P \setminus X$.

Perhaps the most telling example to illustrate this is provided by

$$QQ = (b \rightarrow STOP) \rhd (a \rightarrow QQ)$$

$QQ \setminus \{a\}$ does not have the unstable failure $(\langle\rangle, \{b\})$ because b is possible infinitely often along the only divergent trajectory. However in every CSP model detailed in [23], QQ is equivalent to the following RR (both being divergence-free and hence without *unstable* failures themselves), where

$$RR = ((b \rightarrow STOP) \rhd AS) \sqcap (a \rightarrow RR)$$

where $AS = a \rightarrow AS$. But $RR \setminus \{a\}$ does have the unstable failure $(\langle\rangle, \{b\})$ as the only divergent trajectory on the empty trace never has any visible alternative to τ.

The obvious conclusion to draw from this example is that in order to get a compositional model showing unstable failures, under hiding, we would need to record infinite behaviours including all unstable offers that occurred on the way – a much richer model than anything resembling failures.

The second, and less expected, problem operator is parallel. Consider the processes

$$\begin{aligned} P &= a \to (b \to P \mathbin{\Box} a \to P) \\ R &= (P \setminus \{a\}) \underset{\{b\}}{\parallel} (P \setminus \{a\}) \\ P' &= b \to P' \mathbin{\Box} a \to P' \\ R' &= (P' \setminus \{a\}) \underset{\{b\}}{\parallel} (P' \setminus \{a\}) \end{aligned}$$

Plainly $P \setminus \{a\}$ cannot unstably refuse $\{b\}$, and has the same unstable failures as $P' \setminus \{a\}$. If our supposed model were compositional, we would therefore expect R and R' to be equivalent also. It is natural also to expect these processes to be equivalent to $P \setminus \{a\}$ and $P' \setminus \{a\}$. This is so for R', because the only state of its operational semantics offers b.

Imagine, however, that the first P in R does two hidden as, the second does two, the first does two again, and so on. The parallel composition R is then never in a position where the two processes can agree on b, and so this trajectory gives the unstable failure $(\langle\rangle, \{b\})$ in contradiction to what we are expecting. It is interesting to note that the offending trajectory is, in the most obvious sense, fair since both processes perform an infinite number of actions.

The problem with parallel does not arise with the alternative view of unstable failure discussed above, but that still does not solve the problem with hiding.

We conclude that unstable failures are an interesting way of looking at the operational behaviour of a given process, but fall well short of a conventional-style model for CSP.

8 Unstable failures checking via priority

At first sight priority and fairness seem to be mutually exclusive: a prioritised system, by its very definition, does not behave fairly between its options.

Nevertheless priority has frequently been used (e.g. [3]) to implement fairness in practical examples: if we have a point in a program where there is an external choice that we want to be made fairly, simply give the various options priority in turn. That way, if the environment continuously offers one, say C of these choices and the crucial point in the program is reached infinitely often (actually, the number of options in the choice) then C is bound to be picked. The most obvious approach is to rotate priorities in some round-robin manner.

This technique will make an implementation satisfy abstract fairness assumptions, but it is far too specific to represent a simulation of such assumptions themselves. If we prove a property of a system whose fairness is implemented this way, there is no way of knowing that the property will still hold of systems which make fair choices in a different order. One could achieve this, or at least something a lot closer, by taking a nondeterministic choice over the infinity of processes representing all fair schedules, but that would be far from practical model checking.

As already stated, our aim is to model the particular property of acceptance fairness. Given a failures specification S and an LTS implementation of a process P, we want to know if the unstable failures of P are contained in the stable failures of S. We can, without loss of generality, assume that S implies deadlock freedom because, in general, $S \sqsubseteq_F P$ if and only if $AS \mid\mid\mid S \sqsubseteq_F AS \mid\mid\mid P$ where $AS = a \rightarrow AS$ as above and a is some event not occurring in S or P.

It follows that a process P that can diverge without offering some event infinitely often will fail the $S \sqsubseteq_{UF} P$ we consider, because that sort of divergence constitutes the unstable refusal of the whole alphabet – the direct analogue of the stable concept of deadlock.

Suppose that S is the specification of being deadlock-free, namely $DF = \bigsqcap\{a \rightarrow DF \mid a \in \Sigma\}$, and $P = Q \setminus M$ for a τ-free process Q and some set of actions M. This structure $P \setminus M$ is extremely helpful because it allows us to see the resulting τ actions by reference to P itself.

In unstable failures, P will satisfy this provided (i) Q is deadlock free in the usual stable failures model and (ii) it can never perform an infinite sequence of M events along which no event outside M is ever available.

Priority can tell us if there is such an infinite sequence starting from the beginning of Q's execution. Prioritise all events outside M higher than all those in M. Then $\mathbf{Pri}_{\leq}(Q)$ can perform an infinite sequence of M events if and only if none or them is offered from the same state as a higher priority, non-M event in Σ. Since we have assumed that Q has no τ actions, this means that the trajectory Q performs here has no non-M event on offer from any state. Thus $\mathbf{Pri}_{\leq}(Q) \setminus M$ has divergence $\langle\rangle$ if and only if there is a behaviour of the sort described above. Similarly it has minimal divergence $s \neq \langle\rangle$ if it can perform the trace s and then, immediately after the last event in s, engage in an infinite sequence of hidden M events with no non-M alternative.

Proving divergence freedom of this process does not, however, prove that $P \setminus M$ can never unstably refuse the whole alphabet. If $m \in M$ and $a \notin M$ then, for any n, the process $NA(n) \setminus \{m\}$ can unstably refuse Σ, where

$$\begin{aligned} &NA(0) = m \to NA(0) \\ &NA(n) = (a \to STOP) \,\Box\, (m \to NA(n-1)) \quad (n > 0) \end{aligned}$$

is the process that performs an infinite sequence of ms with a offered as an alternative to the first n. Clearly, for $n > 0$, $\mathbf{Pri}_{\leq}(NA(n))$ is equivalent to $a \to STOP$, so hiding m will leave it divergence free.

We can solve this problem and find the unstable refusal in $NA(n) \setminus \{m\}$ if we introduce a second copy of m by renaming, say m', and make it incomparable with both a and m in the priority order.

$\mathbf{Pri}_{\leq}(NA(n)[\![^{m,\,m'}/m,m]\!])$ can now perform any number of m' events whatever the value of n because the possibility of a in initial states does not exclude m'. After a trace of n or more m' events, this prioritised process will also be able to perform m, which is excluded in the initial states. Therefore $\mathbf{Pri}_{\leq}(NA(n)[\![^{m,\,m'}/m,m]\!]) \setminus \{m\}$ can diverge after sufficiently long traces of m's.

These divergences simply reflect $NA(n)$'s ability to perform an infinite trace of m's with only finitely many offers of a along the way: by the time a particular divergence appears there are *no* further offers available.

We can generalise this construction to one that enables us to decide the absence of unstable failures (s, Σ) for $Q \setminus M$ for the case alluded to above of a τ-free deadlock-free LTS Q. Simply choose a new event m' outside M and not used by Q, and make it incomparable with all other elements of the order in which members of $\Sigma \setminus (M \cup \{m'\})$ have higher priority than members of M (with no other orderings amongst these events). Then

$$\mathbf{Pri}_{\leq}(Q[\![^{m,\,m'}/m,m \mid m \in M]\!]) \setminus M$$

is divergence free if and only if $Q \setminus M$ can never perform an infinite sequence of hidden M actions with no alternative from outside M, or in other words if it does not have an unstable failure of the form quoted above.

Note that the process checked here has the same number of states as Q: every state of Q is reachable because of the role of m', but there is only one state of this construct for each of Q.

As a variation on this, suppose we partition $\Sigma \setminus (M \cup \{m'\})$ into two parts, F and U. We could then prioritise F above M as above, and make the events of U incomparable alongside m'. A divergence of the resulting system would then coincide with an unstable failure (s, F). Thus absence of divergence means that in any infinite sequence of hidden M events, events from F must be offered infinitely often. This gives an efficient way to extend from simple deadlock freedom to some other failures specification, but we need some further trick to extend to general deadlock-free failures specifications S.

Suppose S is a general deadlock-free specification process that a process P trace refines. Then we can define $NR(S)$ to be the set of those X that are subset minimal with respect to $(\langle\rangle, X)$ not being a failure of S. $NR(S)$ is nonempty because Σ is finite and $(\langle\rangle, \Sigma) \notin S$.

Choose a new event d that is outside the set αS of all elements of Σ that are possible for S. (Note that $\alpha P \subseteq \alpha S$ because we are assuming that $S \sqsubseteq_T P$.) For a set of refusals R, let $T(R) = \Box_{X \in R}\, d \to (?x : X \to DS)$ where $DS = d \to DS$. Note that $T(R) \underset{\alpha S}{\parallel} P$ can deadlock if and only if, when one of the sets $X \in R$ is offered to P on $\langle\rangle$, P refuses it. This parallel composition is therefore deadlock free if no member of R is an initial (stable) refusal of P.

Now let

$$\mathit{Test}(S) = (?x : S^0 \to \mathit{Test}(S/\langle x\rangle))\ \Box\ T(NR(S))$$

The parallel composition $\mathit{Test}(S) \underset{\alpha S}{\parallel} P$ is then deadlock free if and only if $S \sqsubseteq_F P$, given that we know that $S \sqsubseteq_T P$: the composition can deadlock if and only if, after one of its traces s, P can refuse a set that S does not permit.

This construction for deciding failures refinement is very similar to the "watchdog" one set out in [9]. The main difference is that ours is constructed with no τ actions: the visible action d replaces τ. The above analysis applied, of course, to the standard stable failures model.

In the case where $P = Q \setminus M$ we can assume that $M \cap (\alpha S \cup \{d\}) = \emptyset$, for if not then we can use renaming to make this so without changing the result of the refinement check. Under this assumption it is reasonable to ask whether $Q \setminus M$ meets the specification S with respect to unstable failures if and only if

$$\mathit{Test}(S) \underset{\alpha S}{\parallel} (Q \setminus M) = (\mathit{Test}(S) \underset{\alpha S}{\parallel} Q) \setminus M$$

is deadlock-free in this sense. Note that this equality holds because of the above assumption, and that the process to which hiding is applied on the right-hand side is free of τ actions. If this same process is not conventionally deadlock free, then the stable failures refinement $S \sqsubseteq_F Q \setminus M$ can easily be seen to be false.

Now none of the events of $\mathit{Test}(S)$ is hidden by $\setminus M$, so it follows that any infinite sequence of τ actions by the right-hand side above all come from M events hidden in Q and that $\mathit{Test}(S)$ remains in the same state throughout. Furthermore this state cannot be one that offers a d if this divergence gives rise to the unstable refusal Σ. Thus unstable refusals of Σ correspond to $\mathit{Test}(S)$ being in one of the offer states $?y : Y \rightarrow DS$ of $T(NR(s))$ for s being the current trace of $Q \setminus M$. Thus in such a behaviour, $Q \setminus M$ exhibits the unstable failure (s, Y), contrary to S.

We therefore have a general technique for deciding whether, for τ-free Q, $Q \setminus M$ meets an arbitrary failures specification with respect to unstable failures.

Some of our earlier examples like QQ and RR demonstrate that we cannot expect such success for processes Q that have τ actions that we do not have the ability to see directly in CSP. It is fortunately straightforward, for any CSP process Q and an event d not in its alphabet, to create a process Q' with τ-free operational semantics such that $Q' \setminus \{d\}$ is equivalent as an LTS to Q's operational semantics. This can be achieved by *syntactic* transformation of the definition of Q.[9] This transformation is syntactic and certainly does not respect semantics. The syntax of QQ and R transform to

$$\begin{aligned} QQd &= (b \rightarrow STOP) \,\Box\, (d \rightarrow a \rightarrow QQd) \\ RRd &= (d \rightarrow ((b \rightarrow STOP) \,\Box\, (d \rightarrow AS))) \\ &\quad\ \Box\, (d \rightarrow a \rightarrow RRd) \end{aligned}$$

which are themselves not equivalent.

Of course a similar transformation can be done at the LTS level: each τ is turned into the extra visible action d.

Our conclusion is that it is possible to decide $S \sqsubseteq_{UF} Q \setminus M$ on FDR extended by priority for finite-state S and Q provided P's operational semantics contains no τ actions. To decide $S \sqsubseteq_{UF} P$ for general P with τ actions, we we have to transform it back into the first form.

[9] Our arguments to date show that there is no transformation based on abstract semantics alone.

9 An industrial example: availability checking in Verum's ASD:Suite

This section describes the example which inspired the application of `prioritise` to fairness properties. We first give some background on ASD:Suite and then describe how we use a model of fast and slow τs to capture appropriate fair availability properties using the techniques described above.

9.1 Background on ASD:Suite

Analytical Software Design (ASD) [4] is software design automation platform developed by Verum[10] that provides software developers with fully automated formal verification tools that can be applied to industrial scale designs without requiring specialised formal methods knowledge from the user. ASD was developed for industrial use and is being increasingly deployed by customers in a broad spectrum of domains, such as medical systems, electron microscopes, semi conductor equipment, telecoms and light bulbs. Industrial examples, such as the development of a digital pathology scanner, using ASD can be found in [5].

ASD is a component-based technology: systems are defined in terms of ASD components and foreign components. An ASD component is a software component specified, designed, verified and implemented using ASD and is specified by:

1) An ASD interface model specifying the externally visible behaviour of a component and
2) an ASD design model specifying its inner working and how it interacts with other components.

Corresponding CSP models are generated automatically from design and interface models, and the ASD component designs are formally verified using FDR, though the CSP is not visible to the end user.

The source code that implements an ASD component is generated automatically from its design model in the specified target programming language. A foreign component is one which forms part of the run-time environment that that ASD generated code interacts with and is captured as an ASD interface model. Foreign components are the mechanism by which ASD components are integrated with existing legacy code and off-the-shelf components.

Figure 1 gives an overview of the standard ASD architecture which

[10] `www.verum.com`

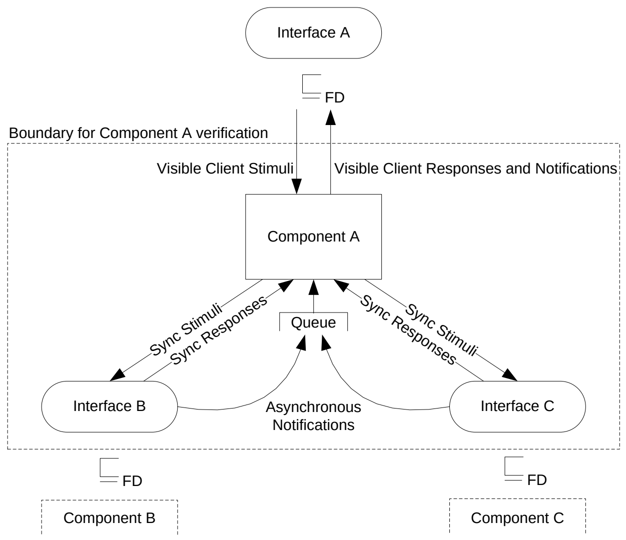


Fig. 1. ASD architecture.

is based on the client-server model. Within an ASD model, system behaviour is specified in terms of stimuli and responses. A stimulus in Component A represents either a synchronous procedure call initiated from a Client above or an asynchronous notification event received from its queue. A response in Component A will either be a response to its Client above or a synchronous procedure call downwards to Interfaces B or C.

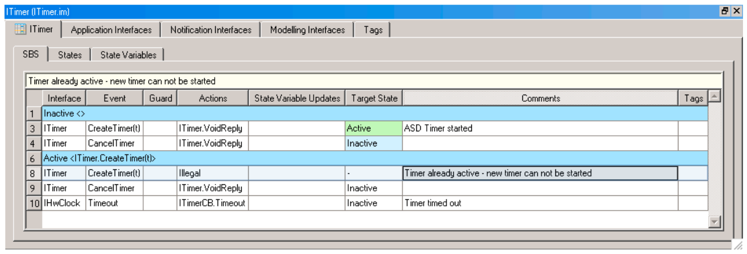
ASD design and interface models are specified in an extended version of the Sequence Based Enumeration specification language [19] and automatically translated into corresponding CSP models. The CSP model not only captures the behaviour in the ASD models as specified by the user, but it also reflects the properties of the ASD run-time environment in which the generated code will be executed. This includes:

1) The externally visible behaviour of the foreign components and ASD components that form the environment in which the ASD design runs;
2) Synchronous procedure calls can only be initiated downwards from a client to a server component and therefore the corresponding synchronous return events only occur upwards from a server component to its client;
3) a client can only invoke 1 synchronous procedure call on the server at a time;

4) a client can only invoke a synchronous procedure call on its server when the server's queue is empty; and
5) the server queue is non-blocking: there is always space in the queue if the used components choose to post a notification event in it.

The CSP models are verified for errors such as deadlocks, livelocks, interface compliance, illegal behaviour, illegal nondeterminism, data range violations and refinement of the design and it's used interfaces with respect to a given specification, known as the design's implemented interface specification. In Figure 1, the implemented interface is that Component A must satisfy is Interface A.

An ASD design model specifies the complete implementation of a component from which the run-time code is generated; it must therefore be deterministic. On the other hand, an interface model is a partial view of a foreign component and captures only the visible communication between itself and its client component that is using it. Therefore, interface specifications are typically more abstract and nondeterministic in nature. As well as stimuli and response events shared with its client, an interface specification can also contain abstract modelling events to represent internal behaviour that is itself invisible to its client but might nevertheless influence the state of the component in a manner which is visible to the client. For example, a simplified version of the standard ASD timer interface specification is defined in Figure 2.



Timer already active - new timer can not be started

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	Inactive <>							
3	ITimer	CreateTimer(t)		ITimer.VoidReply		Active	ASD Timer started	
4	ITimer	CancelTimer		ITimer.VoidReply		Inactive		
6	Active <ITimer.CreateTimer(t)>							
8	ITimer	CreateTimer(t)		Illegal		-	Timer already active - new timer can not be started	
9	ITimer	CancelTimer		ITimer.VoidReply		Inactive		
10	IHwClock	Timeout		ITimerCB.Timeout		Inactive	Timer timed out	

Fig. 2. ASD interface model.

There are 2 canonical states defined in this interface model, namely Inactive and Active. In the Inactive state, this interface offers 2 synchronous procedure calls to its client represented by the stimuli ITimer.CreateTimer and ITimer.CancelTimer. If its client calls ITimer.CreateTimer then the interface immediately returns with the synchronous return event ITimer.VoidReply, thereby passing the thread of control back to its client; the client is now free to carry on executing its own instructions and the interface is now in state Active. In state Active, there is a modelling event called IHwClock that represents in the internal clock triggering an asynchronous notification event, ITimerCB.Timeout, to be put on its client's queue. This modelling event is hidden from its client reflecting the fact that the client cannot see the internal workings of the timer component and therefore doesn't know when it has occurred. Since the client's queue is non-blocking, from its client's point of view the interface might still be in Active or have moved to Inactive with a notification being placed on its queue. The modelling events can also be used to capture a nondeterministic choice over a range of response sequences that depend on internal behaviour abstracted from the interface specification. Typically, a user will select whether modelling events are *eager*, namely that they will always occur if the system waits long enough for them, or *lazy* capturing the case where they nondeterministically might or might not occur. These correspond to the two main modes of abstraction for CSP described in Chapter 12 of [22], which play an important role in formulating ASD's CSP specifications.

A design model with its used interface models and appropriate plumbing, referred to as the *complete implementation*, is refined against its corresponding *implemented interface specification*, that specifies the design's expected visible behaviour by its client. In turn this implemented interface becomes the used interface when designing and verifying the client component using it. In this refinement, the communication between the design model and its used interface models is hidden, since it is not visible to a client using this design. One of the properties that the complete implementation must satisfy is livelock freedom. For example, if a design can invoke a cycle of infinite communication with one or more of its used interfaces without any visible communication being offered with its client, we say the client is starved and this erroneous behaviour must be flagged and corrected. Within CSP such behaviour is captured as divergence.

9.2 Benign and malign divergence

There are divergences that arise during the verification of ASD models that are not regarded as erroneous behaviour in practice due to assumptions of fairness in the notion of 'time passing' at run-time. These are referred to as benign divergences.

An example of how a benign divergence arises in ASD is with the implementation of a timer driven polling loop as follows. An ASD component A is designed to monitor the state of some device by periodically polling it to request its status. In the event that the returned status is satisfactory, component A merely sets a timer, the expiry of which will cause the behaviour to be repeated. In the event that the returned status is not satisfactory, an asynchronous notification is sent to A's client and the polling loop terminates. Thus, A is not interested in normal results; it only communicates visibly to its client if the polled data is abnormal. Whenever component A is in a state in which it is waiting for the timeout event to occur, it is also willing to accept client API stimuli, one of which may be an instruction to stop the polling loop. The design of component A has at least 2 used interfaces, one of them being the Timer interface, Timer, as described above and the other being the interface, PolledUC, for the used component whose status is being polled This is summarised in Figure 3.

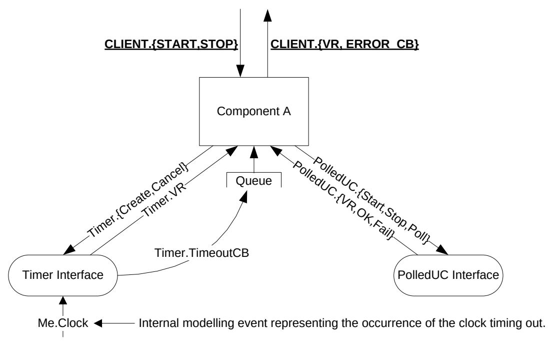


Fig. 3. Component A and its interfaces.

A subset of the behaviour of the design of component A relevant to this discussion can be summarised by the state transition diagram in Figure 4. The events prefixed with CLIENT represent the communication that is shared with the specification on the left-hand side of the refinement and therefore remains visible; all the other events become hidden. The labelled states represent the states of interest for the purposes of describing the divergence in question. All event labels are prefixed with the component name that shares the communication with the design. Events with labels ending in CB are asynchronous notification events that are taken from the design's queue. The divergence occurs in state Y, where the system can perform an infinite cycle of hidden events via state Z, repeating the

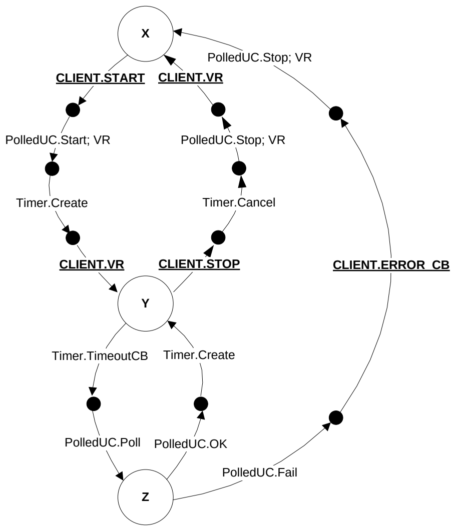


Fig. 4. Subset of Component A's behaviour.

action of timing out, discovering that the polled component is fine and setting the timer again.

In the CSP model and at run-time, A could carry on polling device PolledUC indefinitely. However, at run-time a distinction is made between τ loops where a client API call is available as an alternative and τ loops that offer no alternative and therefore genuinely starve the client. In the former case, the design's client is able to intervene and perform a procedure call that breaks this loop. Provided such a client API stimulus is available then this divergence is not regarded as an error in the design; it will not diverge at run-time because in the real environment time passes between creating a new timer and the corresponding timeout notification event between which the client is able to perform an API call. The design is correct under that assumption which can be safely made due to the implementation of the Timer component. In the example design in the diagram above, the visible event CLIENT.STOP is available in state Y as an option for breaking the diverging cycle of τ events. The assumption at run-time is that the internal clock does not timeout instantaneously, assuming that the create timer procedure call did not set the timer to 0. It is also assumed that it will eventually occur. Therefore a client using the timer process can rely on its occurrence as well as there being some time that passes within which the client may legitimately communicate with components above it in the stack (i.e. the client's client).

In the CSP models, the modelling event Me.Clock cannot be lazily abstracted, since that would imply that it could choose never to happen. On the other hand, simply hiding it means that it is eager and therefore could occur instantaneously. This in turn can cause a divergence in the model exploiting the eager modelling event, which at run-time is not regarded as erroneous behaviour under the assumed circumstances. It is not desirable to place artificial restraints upon the occurrence of such modelling events, as this could restrict the model more than it is able to execute at run-time thereby introducing the risk of missing errors in the design during verification. One cannot simply ignore all divergences in the refinement checks, since they may mask genuine client starvation or other errors normally detected as failures that are present in the design. Abstracting the timing information out is also not an option as there must be a one to one correspondence between the ASD models and the run-time code. Interface models are typically supplied by a third party describing existing hardware or software behaviour and therefore assumed to be fixed.

We therefore need CSP models such that the benign divergences are ignored, without losing any behaviour, including genuine divergences called *malign divergences*, in the design that may result in genuine errors that need to be found. Conventional CSP cannot solve this problem, but a solution is achieved using the priority-based techniques described in Section 8. The set of modelling events M is partitioned into two sets. The first set M_{SE} comprises the slow eager modelling events that are controlled by the external used components and are assumed to occur eventually, but not so fast that their speed starves their client, for example ME.Clock in the timer polling loop example described above. The second set M_L comprises the modelling events that might or might not occur and are therefore accurately modelled by lazy abstraction.

If P is the system model with all these modelling events left visible, a divergence in $P \setminus M_{SE}$ can take three forms

- The infinite sequence of τs may only contain finitely many hidden M_{SE} actions. This clearly represents a form of malign divergence.
- There might be infinitely many hidden M_{SE} actions, only finitely many of which have the alternative of a client API event. This is another form of malign divergence since there is the possibility of client starvation.

– Finally, infinitely many of the M_{SE} events might have a client API event as an alternative. As discussed above, this is a benign divergence.

You can think of there being a distinction between "slow τs" formed by hiding M_{SE} – these give the client time to force an API – and ordinary "fast τs", which do not. This is slightly different from the scenarios in Section 8 because we are only considering unstable offers to be made at the slow τs along a divergence.

Checking the divergence-freedom of

$$\mathbf{Pri}_{\leq}(P[\![^{m,m'}/_{m,m} \mid m \in M_{SE}]\!]) \setminus M_{SE}$$

gives precisely the check for malign divergence that we want: it does not find benign ones. If we needed to check more precisely what API offers were made along sequences of M_{SE} events, we could use the machinery of unstable failures checking discussed earlier in this paper.

After establishing that all divergences are benign, and if necessary make correct offers, the rest of system properties can be checked in the stable failures model of CSP, as is conventional for checks involving lazy abstraction.

10 Conclusions

We have shown how a version of priority, consistent with abstract models of CSP significantly stronger than the usual ones, can be embedded within CSP and implemented in FDR. This extends our ability to model and verify timed systems, to express search strategies more efficiently and to model other languages that themselves have prioritised execution.

A far less obvious benefit is that it allows us to verify that processes' unstable failures are satisfactory, namely that they will offer chosen events infinitely during any divergence, when they are assumed to accept such events (breaking the divergence) if offered.

Our industrial case study was satisfying because this was an example in which a practical problem inspired the creation of a piece of theory (i.e. the connections between priority and model checking acceptance fairness) that would not have been discovered without it. Beyond the scope of the present paper, we have had to bring further fairness considerations into our models to handle further nuances of the ASD models. That will be the subject of a future paper.

We have no doubt that priority will find many other uses in expressing interesting behaviours and specifications in CSP.

References

1. P. Armstrong, M.H. Goldsmith, G. Lowe, J.Ouaknine, H. Palikareva, A.W. Roscoe and J.B. Worrell, *Recent developments in FDR*, To appear in the proceedings of CAV 2012.
2. P. Armstrong, G. Lowe, J.Ouaknine, and A.W. Roscoe, *Model checking Timed CSP*, To appear in Proceedings of HOWARD, Easychair 2012.
3. G. Barrett, *The semantics of priority and fairness in occam*, Mathematical Foundations of Programming Semantics, LCCS 442, 1990.
4. P.J. Hopcroft and G.H. Broadfoot, *Combining the box structure development method and CSP*, Electr. Notes Theor. Comput. Sci., 128(6):127-144, 2005.
5. G H. Broadfoot and P.J. Hopcroft, *A paradigm shift in software development,* Proceedings of Embedded World Conference 2012, Nuremberg. February 29, 2012.
6. E. Brinksma, A. Rensink and W. Vogler, *Fair testing*, Proc CONCUR 1995.
7. C.J. Fidge, *A formal definition of priority in CSP*, ACM Transactions on Programming Languages and Systems, **15**, 4, 1993,
8. T. Gibson-Robinson, *TYGER: A tool for automatically simulating CSP-like languages in CSP*, Oxford University 4th year dissertation, 2010.
9. M.H. Goldsmith, N Moffat, A.W. Roscoe, T. Whitworth and M.I. Zakiuddin, Watchdog transformations for property-oriented model-checking, FME 2003: Formal Methods, LNCS 2805, 2003
10. D. Harel, and M. Politi, *Modeling reactive systems with statecharts: the STATEMATE approach*, McGraw-Hill, Inc. 1998.
11. C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.
12. Inmos Ltd, *Occam programming manual*, Prentice Hall, 1984.
13. D.E. Knuth, *Selected Papers on Fun and Games* CSLI Lecture Notes, no. 192, Stanford, 2010.
14. A.E. Lawrence, *CSPP and event priority*, Communicating Process Architectures, **59**, 2001
15. G. Lowe, *Probabilistic and prioritised models of Timed CSP*, Theoretical Computer Science, **138**, 2, 1995.
16. J. Ouaknine, *Discrete analysis of continuous behaviour in real-time concurrent systems*, Oxford University D.Phil thesis, 2001.
17. J. Ouaknine, *Digitisation and full abstraction for dense-time model checking*, TACAS Springer LNCS, 2002.
18. I. Phillips, *Refusal testing*, Theoretical Computer Science, **50**, 3, 1987.
19. S. J. Prowell and J. H. Poore, *Foundations of sequence-based software specification*, IEEE Trans. on Soft. Eng., **29**(5):417-429, 2003.
20. G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, Theoretical Computer Science **58**, 249-261, 1988.
21. A.W. Roscoe, *Model checking CSP*, in 'A classical mind: essays in honour of C.A.R. Hoare', Prentice Hall, 1994.
22. A.W. Roscoe, *The theory and practice of concurrency* Prentice Hall, 1997.
23. A.W. Roscoe, *Understanding concurrent systems*, Springer, 2010.
24. A.W. Roscoe, *On the expressiveness of CSP*, Submitted for publication.
25. A.W. Roscoe and Zhenzhong Wu, *Verifying Statemate statecharts using CSP and FDR*, Formal Methods and Software Engineering, LNCS 4260, 2006.
26. S.A. Schneider, *Concurrent and real-time systems: the CSP approach*, Wiley, 2000.