# Slow abstraction via priority

A.W. Roscoe and P.J. Hopcroft

Oxford University Department of Computer Science,
and Verum Software Technologies BV

**Abstract.** CSP treats internal $\tau$ actions as *urgent*, so that an infinite sequence of them is the misbehaviour known as *divergence*, and states with them available make no offer that we can rely on. While it has been possible to formulate a number of forms of abstraction in these models where the abstracted actions become $\tau$s, it has sometimes been necessary to be careful about the interpretation of $\tau$s and divergence. In this paper, inspired by an industrial problem, we demonstrate how this range of abstractions can be extended to encompass the offers made by processes during a run of "slow $\tau$s", namely abstractions of interactions with an external agent that does not usually respond urgently to an offer, but always eventually does respond. This extension requires the *prioritise* operator recently introduced into CSP and its refinement checker FDR. We demonstrate its use in the modelling used in Verum's ASD:Suite.

## 1 Introduction

Hoare's CSP [7, 12, 13] treats the actions a process can perform alike, except that while ordinary visible communications in the alphabet $\Sigma$ require the agreement of the external environment to occur, the special action $\tau$ does not.

The CSP hiding operator $P \setminus X$ makes the assumption that the hidden $X$ actions (now $\tau$s) happen as soon as they can. However, hiding is also a means of abstracting part of the external interface of a process, and this has meant that in formulating concepts such as $\mathcal{L}_A(P)$ (lazy abstraction [12]) we have had to allow for abstracted actions not being *urgent*. Lazy abstraction assumes that abstracted events may occur inside $\mathcal{L}_A(P)$ whenever $P$ can do them, or may not, but will never happen so fast as to exclude the rest of $P$'s interface and cause divergence. The abstracted user may even behave like $STOP$ and never do anything. Lazy abstraction has proved a powerful tool for formulating specifications such as security and fault tolerance. It is used in the CSP models of embedded systems created by Verum (see Section 6). Hiding corresponds to *eager* abstraction, as abstracted actions are always on offer to the process.

Some of Verum's modelling needed an abstraction that was somewhere between the two. They needed a model where the owner of $A$ could still delay the process $P$, and still did not prevent other users getting to use $P$, but which would not refuse events in $A$ for ever. So we might characterise these agents as *non-urgent* (as in lazy abstraction) but *ultimately compliant*. One class of actions that fits well into this model are ones (like the *tock* action described in [12, 13]) that represent the regular passage of time.

We characterise *slow* abstraction $\mathcal{S}_A(P)$ by looking at limiting behaviour along infinite execution paths of $P$. While this abstraction proves impossible to express directly using CSP operators, we discover a method using priority for deciding whether $\mathcal{S}_A(P)$ refines a chosen specification. Beginning with a background section on CSP and its models, the rest of this paper develops the above ideas and ends with a case study showing how our methods have been used in Verum's ASD:Suite, a tool for creating correct-by-design embedded software.

We make two simplifying assumptions. Firstly we do not consider abstractions of processes able to terminate ($\checkmark$), avoiding some special cases. Secondly we only consider the case where the alphabet $\Sigma$ is finite, though we will feel free to extend it, for modelling and analytic purposes, to a larger finite set.

## 2 Background

### 2.1 CSP and its semantics

CSP is based on 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 [7, 12, 13, 15] all provide thorough introductions to CSP. The main constructs that we will be using in this paper are set out below.

- The processes $STOP$, $SKIP$ and **div** respectively do nothing, terminate immediately with the signal $\checkmark$ and diverge by repeating the internal action $\tau$. $Run_A$ and $Chaos_A$ can each perform any sequence of events from $A$, but while $Run_A$ always offers the environment every member of $A$, $Chaos_A$ can nondeterministically choose to offer just those members of $A$ it selects, including none at all.
- $a \rightarrow P$ *prefixes* $P$ with the single communication $a$ which belongs to the set $\Sigma$ of normal visible communications. Similarly $?x : A \rightarrow P(x)$ offers the choice $A$ and then behaves accordingly.
- CSP has several *choice* operators. $P \,\Box\, Q$ and $P \sqcap Q$ respectively offer the environment the first visible events of $P$ and $Q$, and make an internal decision via $\tau$ actions whether to behave like $P$ or $Q$.
  The asymmetric choice operator $P \rhd Q$ offers the initial visible choices of $P$ until it performs a $\tau$ action and opts to behave like $Q$. In the cases of $P \,\Box\, Q$ and $P \rhd Q$, the subsequent behaviour depends on what initial action occurs.
- $P \setminus X$ (hiding) behaves like $P$ except that actions in $X$ become $\tau$s.
- $P[\![R]\!]$ (renaming) behaves like $P$ except that when $P$ performs an action $a$, the new process performs 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, including $P \,|||\, Q = P \underset{\emptyset}{\parallel} Q$ in which no synchronisation happens at all.

Other CSP operators such as $P;\ Q$ (sequential composition), $P \triangle Q$ (interrupt) and $P \Theta_a Q$ (throw an exception) do not play a direct role in this paper.

We understand a CSP process in terms of its pattern of externally visible communications. CSP has several styles of semantics that can be shown to be appropriately consistent with one another [12, 13]. The two styles that will concern us 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.

An LTS models a process as a set of states that it moves between via actions in $\Sigma^\tau$, where $\tau$ cannot be seen or controlled by the environment. There may be many actions with the same label from a single state, in which case the environment has no control over which is followed. The best known behavioural models of CSP are based on the following. *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 $\tau$. *Divergences* are traces after which the process can perform an infinite uninterrupted sequence of $\tau$ actions, in other words diverge. The models are then

- $\mathcal{T}$ 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*.

Traces, failures and divergences are all observations that can be made of a process in *linear time*. As described in [13], there is a range of other models based on richer forms of linear observation. An example 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 [10]) been recognised as being relevant to priority. However, we show in this paper that (unexpectedly) refusal-testing models are not always sufficient to encapsulate priority, and that 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. The latter are sometimes called *acceptance traces* models.

### 2.2 Lazy abstraction

Lazy abstraction $\mathcal{L}_A(P)$ captures what a process looks like to an observer unable to see the events $A$, assuming that there is another user who can, and can accept or refuse them. See [12] for a full discussion.

The traces of $\mathcal{L}_A(P)$, like any way of abstracting $A$, are those of $P \setminus A$. As the abstracted user can at any time refuse or accept events in $A$, its failures are those of

$$(P \underset{A}{\parallel} Chaos_A) \setminus A$$

whose traces are again correct. We assume that the abstracted user cannot consume so much of $P$'s resources as to make it diverge, so we assert that the abstraction (unlike the above CSP expression) never diverges.

### 2.3 FDR

FDR[11–13] is a refinement checker between finite-state processes defined in CSP. First created in the early 1990's it has been regularly updated since, and indeed a completely new version FDR3 will be released late in 2013.[1]

It uses CSP$_M$, namely CSP extended by Haskell-like functional programming. Thus one can define complex networks and data operations succinctly, and create functions that, given abstract representations of structures or systems, can automatically generate CSP networks to implement and check them. The FDR is at the heart of the verification functionality of ASD:Suite [3, 4]: the tool captures state machine models of proposed embedded systems, and then builds CSP models of how it will implement these so that they can be checked for correctness properties. This is only one of several major uses of FDR in government and industry, almost all of which start by translating some other notation to CSP.

FDR checks refinements of the form $Spec \sqsubseteq_X Impl$, where *Spec* is a process representing a specification in one of the standard CSP models $X$, usually traces, stable failures or failures-divergences. *Impl* is a CSP representation of the system being checked. Typically this sort of check scales better in *Impl* than *Spec*, the latter of which has to be normalised as part of the decision process. FDR supports a number of techniques for attacking the state explosion problem, including hierarchical compression. The algorithms underpinning FDR are set out in [12–14]. A number of recent additions to FDR including priority were summarised in [1].

## 3 A priority operator

There have been a number of proposals, for example [5, 8, 9], for the introduction of priority into CSP. These usually proposed ways in which a process could prefer one action to another, even though both remained available. An approach like that would automatically invalidate not only CSP's existing semantic models, which would have to be redeveloped to accommodate these preferences, but also the use of FDR in anything close to its usual form, since FDR supports transition systems without preferences. However, [13] introduced a priority that does make sense over ordinary labelled transition systems. The one we discuss here is a slightly more expressive version of that.

[1] The initial release of FDR3 will, functionally, be similar to FDR2.94 except that it will support multi-core execution of some functions, will have a new GUI, and will have an integrated type-checker for CSP$_M$. Further functionality is planned for later versions.

Our operator $\mathbf{Pri}_{\leq}(P)$ is parameterised by a partial order $\leq$ on $\Sigma^{\tau}$, the set of action labels. $\tau$ is constrained to be maximal in $\leq$, but not necessarily maximum. (So there may be visible actions less than $\tau$ and ones incomparable to $\tau$, but not ones greater than $\tau$.) Further, we do not permit non-maximal elements of $\Sigma$ to be incomparable to $\tau$. These conditions are both required to preserve the property that CSP treats the processes $P$ and $\tau.P$ (one that can perform a $\tau$ before acting like $P$) as equivalent.

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 [13], because they require negative premises: an action cannot occur unless all actions of higher priority are impossible. The only operational rule needed for the new operator is

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

The fact that $\tau$ is maximal means it is never blocked by this rule.

Since the operational semantics for $\mathbf{Pri}_{\leq}(P)$ fall outside the "CSP-like" class that guarantees a semantics in every CSP model, it is not a surprise that not all such models are congruences for it. We cannot expect it to respect a model that does not tell us which events a process performs happen from stable states, and whether all $\Sigma$-events less than a given event are then refused. The traces model certainly does not tell us 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 give enough information. Consider the pair of processes $(a \rightarrow b \rightarrow STOP) \rhd (a \rightarrow STOP)$ and $(a \rightarrow STOP) \rhd (a \rightarrow b \rightarrow STOP)$. These divergence-free processes have identical failures, but imagine applying a priority operator to them where $a < b$. In each case, the $a \rightarrow \cdot$ that appears to the left of $\rhd$ is prevented because $\tau$ (necessarily, given our assumptions, of higher priority than $a$) 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; we also need to know about stability and the refusal of high-priority events earlier in traces.

The refusal-testing models do distinguish these two processes, because they have different behaviours after the refusal of all events other than $a$, followed by the action $a$ have both been observed (which is written $\langle \Sigma \setminus \{a\}, a \rangle$ in the notation below). 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 [13]. In the simplest of these, the *stable refusal-testing model* $\mathcal{RT}$, the behaviours recorded of a process are all of the form

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

where $n \geq 0$ and each $X_i$ is either a refusal set (subset of $\Sigma$) 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 $\tau$) 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 $\{a \in \Sigma \mid a > a_{i-1}\}$ if not (even if that set is empty so $a_{n-1}$ is less than only $\tau$).

It came as a surprise to us, however (particularly given what one of us wrote in [13]), 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

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

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

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

Now consider $Q1 = DF1(\{a, b\}) \,|||\, CS$ and $Q2 = DF2(\{a, b\}) \,|||\, CS$ where $CS = c \rightarrow CS$. Clearly $Q1$ and $Q2$ are refusal-testing equivalent. Let $\leq$ be 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 $\mathbf{Pri}_{\leq}(Qi)$?

When $i = 1$ the answer is "no", since whenever $Q1$ performs the event $a$ the set of events it offers is precisely $\{a, c\}$. It can also offer $\{b, c\}$, but in neither case can it perform $a$ after the refusal of $\{c\}$. However, $Q2$ 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". Thus we need more information than refusal testing of $Qi$ to calculate the refusal-testing behaviours of $\mathbf{Pri}_{\leq}(Qi)$.

This example tells us that $\mathbf{Pri}_{\leq}(\cdot)$ can only be compositional for refusal testing when the structure of $\leq$ is such that whenever $a$ and $b$ are incomparable events in $\Sigma$ and $c < b$ then also $c < a$. This is because we could reproduce an isomorphic example in any such order. It is, however, possible to give a compositional semantics for refusal testing when there is no such triple. This means that the order has to take one of two forms:

- A list of sets of equally prioritised events, the first of which contains $\tau$.
- A list of sets of equally prioritised events, the first of which is exactly $\{\tau\}$, together with a further set of events that are incomparable to the members of the first two sets in the list and greater than those in the rest.

The priority order used in enforcing maximal progress in timed models does satisfy the above, but the $\leq$s we will use in analysing slow abstraction below do not satisfy it.

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 $\mathcal{FL}$, the semantics of $\mathbf{Pri}_{\leq}(P)$ are the behaviours (the $A_i$ being stable acceptances or $\bullet$):

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

where for each $i$ 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 $A_i = \{a \in Z_i \mid \neg\, \exists\, b \in Z_i . b > a\}$,
- and in each case where $A_{i-1} \neq \bullet$, $a_i \in A_{i-1}$.

### 3.1 FDR implementation of priority

The nature of the operational semantics of $\mathbf{Pri}_{\leq}(\cdot)$, in particular its use of negative premises, means that this operator cannot be folded into the *supercombinator* structures (see [13]) that lie at the heart of FDR's state machine implementation. It has therefore been implemented as a stand-alone operator that both inputs and outputs an LTS.

We decided that the practical version would have an easier-to-use input format rather than making all users construct a representation of a partial order with the constraints stated earlier. The implemented version therefore restricts the orders to ones that can be represented as a list of sets of visible events, where the first (the events incomparable to $\tau$) may be empty:

```
prioritise(P,As)
```

where `As = <A0,A1...,An>` is a list of subsets of $\Sigma$.

These sets of events have lower priority as the index increases, so `An` are the ones of lowest priority. Importantly, there is no need for the `Ai` to cover all the visible events of `P`: those not in one of the `Ai` are incomparable to all other events including $\tau$ and neither exclude nor are excluded by any other.

### 3.2 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 may be included as explicit annotations to nodes rather than being deduced only from transitions.

With the exception of strong bisimulation, none of the compressions described in [14] is guaranteed to preserve the refusal-testing and acceptance traces models of CSP. In consequence, they cannot be reliably used inside a `prioritise` operator.

In part as a remedy for this problem, we have recently implemented the compression *divergence-respecting weak bisimulation* as defined in [13]. (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, it turns an LTS into another LTS rather than a GLTS. We will report separately on this implementation and weak bisimulation's place in the family of CSP compression functions.

## 4 Slow abstraction

In the introduction we set out, informally, the problem of formulating the correct abstraction of a process $P$ relative to an unseen user who is assumed to be lazy but eventually compliant with requests from the process, and who controls some subset $A$ of $P$'s events. We do not want these events to be visible to the external observer, as represented by our specification, which is expressed in the failures model $\mathcal{F}$ – a choice we will justify shortly. We will assume that $P$ itself is free from divergence.

The abstracted events do not happen quickly enough for them to exclude offers of other events being accepted by the process indefinitely. Our model, therefore, is that the abstracted process has the *stable* failure $(s, X)$ if and only if $(s, X \cup A)$ is a failure of $P$ for some $s'$ with $s' \setminus A = s$. We say it has the *unstable* failure $(s, X)$ if and only if $P$ has an infinite behaviour

$$P = P_0 \xrightarrow{x_0} P_1 \xrightarrow{x_1} P_2 \ldots$$

where, if $u$ is the necessarily infinite trace consisting of all the non-$\tau$ $x_i$, $u \setminus A = s$ and there is some $k$ such that (i) $x_i \in A \cup \{\tau\}$ for $i > k$, (ii) sufficient of the $P_i$ are stable, an issue we will discuss below, and (iii) all but finitely many of these stable $P_i$ refuse $X$. In other words it ultimately performs an infinite number of $A$ events from stable states, all of which refuse $X$. We characterise the second sort as unstable failures because the abstraction is turning $A$ actions into a sort of slow internal action, meaning that the refusals are occurring over a series of states linked by these actions.

The stable failure case, of course, corresponds to the regular definition of $P \setminus A$ over $\mathcal{F}$. The unstable case comes when the trace $s$ in $P \setminus A$ is followed by an infinite sequence of events in $\{\tau\} \cup A$. Infinitely many of these must be in $A$ because $P$ is divergence-free.

Our idea that the abstracted user is lazy has to be made a little more specific here. It should not be too hard to see that this is closely related to the question of how many of the $A$ actions in the trace above happen from stable states. Since $P$ is divergence-free, it will always reach a stable state if left alone and so it is reasonable (though perhaps debatable if long finite sequences of $\tau$s can occur) to describe an $A$ action that occurs from an unstable state as eager: our imaginary user has performed it before $P$'s available $\tau$s had completed.

If all of these stable states, beyond a certain point, refuse $X$ then we want $(s, X)$ to be a failure of our abstraction.

- If all but finitely many of the $A$s were eager, then it would neither make sense to describe our user as lazy, nor to detect any infinite pattern of refusals from the sequence of states: refusals will only happen from stable $P_i$. We assume that our user is too lazy for this to happen.
- If we insisted that all $A$s happened in stable states, then this is quite a strong assumption about the user. States of $P$ reachable only using eager occurrences of $A$ would not be reached at all.
- The remaining two possibilities are that we insist (i) that only finitely many $A$s are eager or (ii) less restrictively, that an infinite number of them are not eager. Each of these is a reasonable view, but we adopt (i) in part because we have a solution to the problem of automating it in FDR, but not (ii). So what we are saying is that the abstracted $A$-user is eventually sufficiently lazy not to take an event from an unstable state. The difference between (i) and (ii) shows up in the process

  $$Q = (a \rightarrow a \rightarrow Q) \rhd ((a \rightarrow a \rightarrow Q) \Box b \rightarrow STOP)$$

  If we abstract $\{a\}$ under assumption (i) then the result cannot refuse $\{b\}$ on $\langle\rangle$. However, under (ii) it can, because it would consider the behaviour in which every odd-numbered $a$ occurs from $Q$'s unstable initial state.

If an infinite, non-divergent behaviour of $P$ has only finitely many visible actions outside $A$, and satisfies (i) we will call it $*A$-stable. We will spend much of this paper analysing such behaviours. For the process $Q$ defined above, only finitely many $a$s can occur from the initial state in a $*A$-stable behaviour.

An unstable failure only manifests itself over an infinite behaviour of $P$, which means that it would not make sense to ask what the process did *after* it. It would not, therefore, make sense to try to work out how our abstraction looks in a refusal-testing model. That is why in considering this type of abstraction we consider only failures specifications, given that for traces specifications we can just use $Spec \sqsubseteq_T P \setminus A$.

The value of $\mathcal{S}_A(P)$ is then the union of both its stable and unstable failures, paired with the traces of $P \setminus A$. This a member of $\mathcal{F}$, the "stable" failures model.

We have thus characterised what the behaviours of our new abstraction are, and so know what it means for it to refine some failures specification. This, however, gives us no clue about how to check properties of them in FDR.

It is interesting to compare $\mathcal{L}_A(P)$ and $\mathcal{S}_A(P)$. There is one direct relation that holds.

**Lemma 1.** *If $P$ is a divergence-free process then $\mathcal{L}_A(P) \sqsubseteq_F \mathcal{S}_A(P)$.*

*Proof.* These two processes have the same set of traces by definition, so all we must do is show that every failure (stable or unstable) of $\mathcal{S}_A(P)$ belongs to $\mathcal{L}_A(P)$. This is trivially true for the stable ones, which are just those of $P \setminus A = (P \underset{A}{\parallel} Run_A) \setminus A$ and $Run_A \sqsupseteq Chaos_A$.

If $(s, X)$ is an unstable failure of $\mathcal{S}_A(P)$ then, from definitions above it follows that there is an infinite series of traces $t_0 < t_1 < t_2 < \ldots$ where $P$ has the failure $(t_i, X)$ for each of them, and $t_i \setminus A = s$ for all $i$. Any one of them, combined with the failure $(t_i \restriction A, A)$ of $Chaos_A$, yields the (stable) failure $(s, X)$ in $(P \underset{A}{\parallel} Chaos_A) \setminus A$. ∎

Consider the process $P = a \rightarrow P \,\Box\, b \rightarrow a \rightarrow P$. Because $P \setminus \{a\}$ can diverge, eager abstraction $P \setminus \{a\}$ does not make sense. $\mathcal{L}_{\{a\}}(P) = b \rightarrow Chaos_{\{b\}}$ since if the abstracted user stops performing $a$s at any stage then $b$s are forced to stop also. $\mathcal{S}_{\{a\}}(P)$, on the other hand, is just $Run_{\{b\}}$ as the abstracted user will always eventually perform $a$, enabling another $b$ if that is necessary.

## 5 Unstable failures checking via priority

We want to find a way of checking whether $Spec \sqsubseteq_F \mathcal{S}_A(P)$. That this holds for the traces and stable failures of the right-hand side can be established by checking $Spec \sqsubseteq_F P \setminus A$. We will assume this has been done, meaning that we want to check that the unstable failures of $\mathcal{S}_A(P)$ also satisfy *Spec*. Doing this on FDR will mean that any counterexample will manifest itself as a divergence, since this is the only sort of infinite counterexample that FDR can produce.

This tells us immediately that this type of behaviour cannot be checked in the usual CSP language without priority, since in that language the divergences of any context $F(P)$ depend only on the traces and divergences of $P$, not on its failures.

It also tells us we cannot find a context $G_A[\cdot]$ such that $G_A[P]$ has the same failures in $\mathcal{F}$ (the stable failures model) as $\mathcal{S}_A(P)$. It will therefore not be possible to test that this abstraction refines a failures specification *Spec* by checking a refinement with *Spec* itself on the left-hand side.

We can conclude that checking the unstable failures aspect of $Spec \sqsubseteq_F \mathcal{S}_A(P)$, at least without extending the functionality of FDR, must take the form

$$LHS(Spec, P, A) \sqsubseteq_M RHS(Spec, P, A)$$

in which $M$ represents a model sensitive to divergence, and where an operator such as $\mathbf{Pri}_{\leq}(\cdot)$ that falls outside traditional CSP is used. We will see later that we can take *LHS* (independent of *Spec*, $P$ and $A$), to be $Chaos_\Sigma$, the most nondeterministic divergence-free process, and $M$ to be failures-divergences.

Before handling general *Spec* we will first show how to deal with the case that *Spec* is the failures specification of deadlock freedom:

$$DF = \bigsqcap \{a \rightarrow DF \mid a \in \Sigma\}$$

$\mathcal{S}_A(P)$ meets this specification provided both the following hold:

- $P$ is deadlock free.
- There is no $*A$-stable behaviour of $P$ such that eventually no action outside $A$ is ever offered from a stable state.

$\mathcal{S}_A(P)$ will satisfy this provided (i) $P$ is deadlock free in the usual sense and (ii) $P$ has no state from which there is an infinite sequence of events in $A \cup \{\tau\}$ where all the $A$'s are from states offering only subsets of $A$.

Priority can tell us if there is such a sequence starting from $P$'s initial state. Prioritise all events outside $A$ over those in $A$. Then $\mathbf{Pri}_{\leq}(Q)$ can perform an infinite sequence of $A$ events if and only if none of them is offered from the same state as a higher priority, non-$A$ event in $\Sigma \cup \{\tau\}$. Thus $\mathbf{Pri}_{\leq}(P) \setminus A$ has divergence $\langle\rangle$ if and only if $P$ can perform an infinite trace of $A$ events where none of them is from a state where a non-$A$ event is offered.

Proving divergence freedom of this process does not, however, prove that $\mathcal{S}_A(P)$ can never unstably refuse the whole alphabet. If $a \in A$ and $b \notin A$ then, for any $n$, the process $\mathcal{S}_A(NB(n))$ can unstably refuse $\Sigma$, where

$$NB(0) = a \rightarrow NB(0)$$
$$NB(n) = (b \rightarrow STOP) \,\Box\, (a \rightarrow NB(n-1)) \quad (n > 0)$$

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

We can solve this problem and find the unstable refusal in $\mathcal{S}_A(NB(n))$ if we introduce a second copy of $a$ by renaming it, say to $a'$, and make it incomparable with both $a$ and $b$ in the priority order. So in particular it can happen even when $a$ is prevented by priority.

$\mathbf{Pri}_{\leq}(NB(n)[\![a, a'/a, a]\!])$ can now perform any number of $a'$ events whatever the value of $n$ After a trace of $n$ or more $a'$ events, this prioritised process will also be able to perform $a$, which is excluded in the initial states. Therefore $\mathbf{Pri}_{\leq}(NB(n)[\![a, a'/a, a]\!]) \setminus \{a\}$ can diverge after sufficiently long traces of $a'$s.

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

The construction $\mathbf{Pri}_{\leq}(NB(n)[\![a, a'/a, a]\!]) \setminus \{a\}$ does not in itself represent the abstraction $\mathcal{S}_A(NB(n))$, both because it has the ability to perform $a'$ actions, and because deadlocks in the abstraction have become divergences. It does, however, show us how to use FDR to check for the abstraction being deadlock free. This method generalises as follows.

**Theorem 1.** *For the divergence-free process $P$, the abstraction $\mathcal{S}_A(P)$ contains no unstable failure of the form $(s, \Sigma)$ if and only if $((P[\![a, a'/a, a \mid a \in A]\!]) \setminus A$ is divergence free. Here it is immaterial whether $a'$ is a single event disjoint from[2] $\alpha P \cup A$ or a separate such event for every member $a$ of $A$.*

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

We now seek to generalise the above to a method for deciding whether $Spec \sqsubseteq_F \mathcal{S}_A(P)$ for arbitrary divergence-free $P$ and failures specification $Spec$.

[2] $\alpha P$ is the set of events used by $P$.

Suppose *Spec* is a specification process that $P \setminus A$ trace refines. We are trying to decide if $S \sqsubseteq_F \mathcal{S}_A(P)$. We can assume that no event of $\alpha Spec$ belongs to $A$, because if there were such events we could rename $A$ to achieve this.

If $S$ is any process such that $\alpha S \subseteq \alpha Spec$ and $(\langle\rangle, \Sigma) \notin failures(S)$, we can define $NR(S)$ to be the set of those $X$ that are subset minimal with respect to $(\langle\rangle, X) \notin failures(S)$. $NR(S)$ is nonempty because $\Sigma$ is finite and $(\langle\rangle, \Sigma) \notin S$.

If $S$ is a process that can deadlock immediately, let $NR(S) = \emptyset$.

Choose a new $d$ that is outside $\alpha Spec \cup A$. (Note that $\alpha P \subseteq \alpha Spec \cup A$ because we are assuming that $Spec \sqsubseteq_T P \setminus A$.) For a set of refusals $R \neq \emptyset$, let

$$T(R) = \Box_{X \in R}\, d \rightarrow (?x : X \rightarrow DS)$$

and $T(\emptyset) = DS$, where $DS = d \rightarrow DS$. Note that $T(R) \underset{\alpha Spec}{\parallel} Q$, for $Q$ a process such that $S \sqsubseteq_T Q$, can deadlock if and only if, when one of the sets $X \in R$ is offered to $P$ when it has performed $\langle\rangle$, $P$ refuses it. This parallel composition is therefore deadlock free if no member of $R$ is an initial (stable) refusal of $P$. Now let

$$Test(S) = (?x : S^0 \rightarrow Test(S/\langle x\rangle)) \ \Box\ T(NR(S))$$

For $Q$ such that $Spec \sqsubseteq_T Q$, the parallel composition $Test(Spec) \underset{\alpha Spec}{\parallel} Q$ is then deadlock free if and only if $Spec \sqsubseteq_F Q$, 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. In understanding this it is crucial to note that the $d$s of $T(NR(S))$, including the one in the initial state of $Test(S)$, can occur unfettered in the parallel composition because they are not synchronised with $Q$. The first $d$ that occurs fixes the present trace as the one after which $Test(Spec)$ checks to see that a disallowed refusal set (if any) does not appear in $Q$.

Our construction turns any case where $Q$ fails *Spec* into a deadlock. It is very similar to the "watchdog" transformation for the usual failures model set out in [6]. The main difference is that ours is constructed with no $\tau$ actions: the visible action $d$ replaces $\tau$.

Consider the case where $Q$ is replaced by $P$. This has the additional events $A$ which are not synchronised with $Test(Spec)$, so the combination $Test(Spec) \underset{\alpha Spec}{\parallel} P$ can only deadlock in states where $P \setminus A$ has a stable failure illegal for *Spec*.

We would similarly like unstable failures of $\mathcal{S}_A(P)$ to turn into unstable "deadlocks", namely unstable refusals of $\Sigma$, in $\mathcal{S}_A(Test(Spec) \underset{\alpha Spec}{\parallel} P)$. This is confirmed by the following result.

**Theorem 2.** *Under the assumptions above, including the one that $P$ is divergence-free and $Spec \sqsubseteq_F P \setminus A$, $\mathcal{S}_A(P)$ has an unstable failure that violates Spec if and only if $\mathcal{S}_A(Test(Spec) \underset{\alpha Spec}{\parallel} P)$ has an unstable failure of the form $(s, \Sigma)$. Furthermore $\mathcal{S}_A(P) \sqsupseteq_F Spec$ if and only if*

$$(\mathbf{Pri}_{\leq}((Test(Spec) \underset{\alpha Spec}{\parallel} (P[\![a, a' / a, a \mid a \in A]\!])))) \setminus A$$

*is divergence-free.*

*Proof.* The second equivalence follows from what we know once we observe that, since *Test* never performs any member of $A$ and $\alpha Spec \cap (A \cup \{a'\}) = \emptyset$,

$$Test(Spec) \underset{\alpha Spec}{\parallel} (P[\![a, a'/a, a \mid a \in A]\!]) = (Test(Spec) \underset{\alpha Spec}{\parallel} P)[\![a, a'/a, a \mid a \in A]\!]$$

So we need just to establish the first equivalence. Any unstable failure of the form $(s, \Sigma)$ in $Test(Spec) \underset{\alpha Spec}{\parallel} P$ arises from a $*A$-stable behaviour of this combination such that eventually no action outside $A$ is ever offered, and necessarily where eventually no event, other than members of $A$ and $\tau$, occurs. Since $Test(Spec)$ has no $\tau$ or $A$ actions, there is a point in the infinite behaviour beyond which this process performs no action, so all the subsequent actions of the parallel composition are performed by $P$ alone, with $Test(Spec)$ left in some "terminal" state. Since the resulting unstable refusal is $\Sigma$, this terminal state must be one refusing the unsynchronised $d$. Therefore the state is one offering some $X$ such that $(s', X) \notin failures(Spec)$, where $s$ is the trace up to any point in the infinite behaviour beyond the one at which $Test(Spec)$ first reaches its terminal state and $s' = s \setminus (A \cup \{d\})$. It should be clear that from this point extra events may add to $s$ but will not change $s'$.

Since the infinite behaviour witnesses the unstable refusal of $\Sigma$ in the parallel combination, we can assume that the infinite tail of states in which all the stable ones refuse $\Sigma \setminus A$ starts beyond the point where $Test(Spec)$ reaches its terminal state. The sequence of corresponding states in $P$ must all refuse the $X$ that this terminal state is offering. $P \setminus A$ has by that point performed $s'$, recalling that $(s', X) \notin Spec$. $P$'s behaviour thus witnesses the unstable failure $(s,' X)$, meaning that $\mathcal{S}_A(P)$ does not satisfy *Spec*.

It should not be difficult to see that the reverse also holds: if there is a $*A$-stable behaviour of $P$ witnessing an unstable failure $(s, X)$ of $\mathcal{S}_A(P)$ that violates *Spec*, we can assume that $X$ is $\subseteq$-minimal with respect to this. The trace $s^\frown\langle d\rangle$ therefore leads $Test(Spec)$ to a state that offers exactly $X$. The combination $Test(Spec) \underset{\alpha Spec}{\parallel} P$ then has a $*A$-stable behaviour in which, after a finite trace $s'$ such that $s' \setminus (A \cup \{d\}) = s$, $Test(Spec)$ permanently offers $X$ and all stable states of $P$ refuse it, linked only by $\tau$ and $A$ actions. This behaviour clearly witnesses an unstable failure with refusal $\Sigma$. ∎

We therefore have a general technique for deciding whether, for divergence-free $P$, $\mathcal{S}_A(P)$ meets an arbitrary failures specification with respect to unstable failures.

## 6 Availability checking in Verum's ASD:Suite

This example inspired the formulation of slow abstraction and the creation of the decision procedure in terms of priority.

### 6.1 Background on ASD:Suite

Analytical Software Design (ASD) [3] is a software design automation platform developed by Verum[3] that provides software developers with fully automated formal verification tools that can be applied to industrial scale designs without requiring specialised formal methods knowledge of 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 using ASD, such as the development of a digital pathology scanner, can be found in [4].

ASD is a component-based technology: systems contain both 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. While design models are complete and deterministic, interface models are abstract and frequently nondeterministic.

Figure 1 gives an overview of the standard ASD architecture which is based

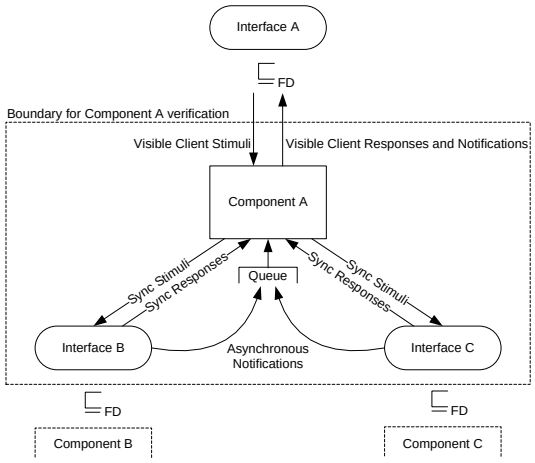


**Fig. 1.** ASD architecture.

on the client-server model. Within an ASD model, system behaviour is specified

[3] `www.verum.com`

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 $C$.

The CSP model generated by ASD not only captures the behaviour in the models specified by the user, but also reflects the properties of the ASD runtime environment in which the generated code will be executed. This includes the externally visible behaviour of the foreign components and ASD components that form the environment in which the ASD design runs. Clients can initiate synchronous procedure calls to servers, with servers communicating in the other direction by return events and non-blocking queues.

The CSP models are verified for errors such as deadlocks, livelocks, interface non-compliance, illegal behaviour, illegal nondeterminism, data range violations, and refinement of the design and its interfaces with respect to a given specification. In Figure 1, the implemented interface is that Component $A$ must satisfy is Interface $A$. As an example, a simplified version of the standard ASD timer interface specification is defined in Figure 2.

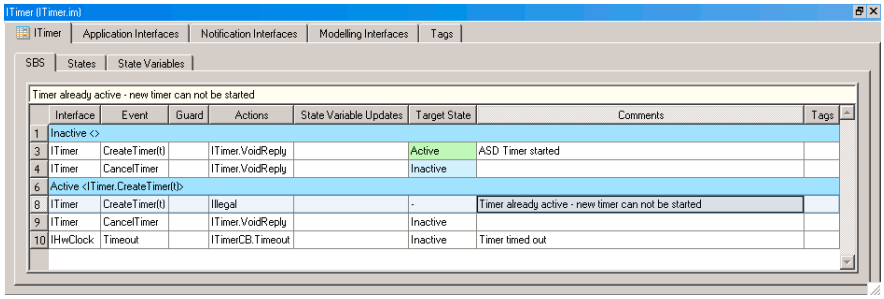


**Fig. 2.** ASD interface model.

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 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 earlier, 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*, which 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 in this design. One of the properties that the complete implementation must satisfy is livelock freedom. For example, if a design can invoke an infinite cycle of communication with one or more of its used interfaces without any visible communication being offered to its client, we say the client is starved: this erroneous behaviour must be flagged and corrected. Within CSP such behaviour is captured as divergence.

### 6.2 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 described above, and the other being the interface, PolledUC, for the used component whose status is being polled. This is summarised in Figure 3.

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

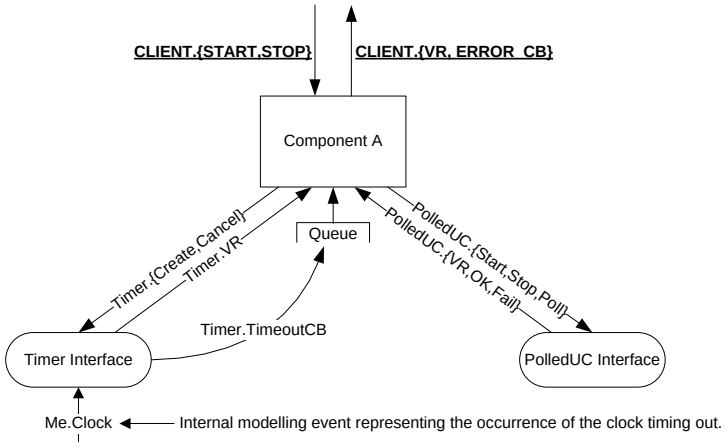


**Fig. 3.** Component $A$ and its interfaces.

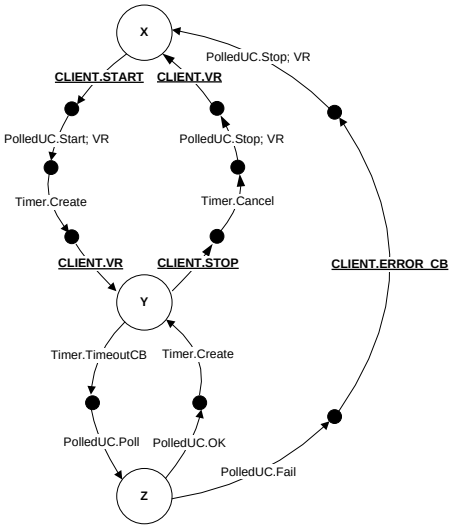


**Fig. 4.** Subset of component $A$'s behaviour.

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 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 $\tau$ loops where a client API call is available as an alternative and $\tau$ 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, 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, during 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 $\tau$ 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).

For this we use our new *slow* abstraction for the modelling event Me.Clock rather than eager or lazy, which would not be correct. Prior to the discover of this technique, the only option was to place artificial assumptions in for form of restraints upon the occurrence of such modelling events. This both increased the state space and carried the risk of missing erroneous behaviours during verification.

One can describe divergences composed of abstracted events, during which the slow abstraction makes an offer, as being *benign*, whereas ones where the offers end, or which are composed of other hidden events, as being *malign* and genuinely erroneous. Our new methods allow this distinction to be made.

The analysis in ASD uses the priority-based techniques described in Section 5. 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 $\tau$s 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 $\tau$s" formed by hiding $M_{SE}$ – these give the client time to force an API – and ordinary "fast $\tau$s", which do not. This is just the view formed by the slow abstraction of the events mapping to the slow $\tau$s after conventional hiding (eager abstraction) of the ones mapping to the fast ones.

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 the system properties can be checked in the stable failures model of CSP, as is conventional for checks involving lazy abstraction. The ASD use of slow abstraction described here corresponds exactly to the case of checking for deadlock freedom which was the core case earlier.

## 7 Conclusions

We have explained the CSP priority operator in terms of operational semantics and shown that, depending on the form of the partial order used, it requires either CSP's refusal-testing or acceptance traces model for compositionality.

We have also studied the problem of abstracting an interface that is neither eager nor is allowed to be completely idle, capturing refusal information from infinite traces. The reason for studying this alongside priority is that priority was the key to automating checks of the *slow* abstraction we developed against failures specifications.

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. slow abstraction and the priority technique for checking properties of it) 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.

The techniques developed in this paper are applicable wherever one models a system which has events that, either because of their assumed internal control or the way they are assumed to be controlled by an unseen external agent, progress in a measured rather than eager manner. It would be interesting to investigate the relationship with Schneider's theory of *timewise refinement* [15], which shows how Timed CSP processes can be seen to satisfy untimed specifications: at least in discretely timed versions of CSP, this appears to be closely related to the slow abstraction of time. Clock and time signals, in general, appear to be excellent candidates for this form of abstraction.

## 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.
3. 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.
4. G H. Broadfoot and P.J. Hopcroft, *A paradigm shift in software development,* Proceedings of Embedded World Conference 2012, Nuremberg. February 29, 2012.
5. C.J. Fidge, *A formal definition of priority in CSP*, ACM Transactions on Programming Languages and Systems, **15**, 4, 1993.
6. 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.
7. C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.
8. A.E. Lawrence, *CSPP and event priority*, Communicating Process Architectures, **59**, 2001.
9. G. Lowe, *Probabilistic and prioritised models of Timed CSP*, Theoretical Computer Science, **138**, 2, 1995.
10. I. Phillips, *Refusal testing*, Theoretical Computer Science, **50**, 3, 1987.
11. A.W. Roscoe, *Model checking CSP*, in 'A classical mind: essays in honour of C.A.R. Hoare', Prentice Hall, 1994.
12. A.W. Roscoe, *The theory and practice of concurrency* Prentice Hall, 1997.
13. A.W. Roscoe, *Understanding concurrent systems*, Springer, 2010.
14. A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson and J.B. Scattergood, *Hierarchical compression for model-checking CSP* or *how to check* $10^{20}$ *dining philosophers for deadlock*, Proceedings of the 1st TACAS, 1995. Springer LNCS 1019.
15. S.A. Schneider, *Concurrent and real-time systems: the CSP approach*, Wiley, 2000.