Noname manuscript No.
(will be inserted by the editor)

Translating between models of concurrency

David Mestel · A.W. Roscoe

the date of receipt and acceptance should be inserted later

Abstract Hoare's Communicating Sequential Processes (CSP) [8] admits a rich universe of semantic models closely related to the van Glabbeek spectrum. In this paper we study finite observational models, of which at least six have been studied for CSP, namely traces, stable failures, revivals, acceptances, refusal testing and finite linear observations [21]. (Others are known.) We show how to use the relatively recently-introduced *priority* operator ([22], ch.20) to transform refinement questions in these models into trace refinement (language inclusion) tests. Furthermore, we are able to generalise this to any (rational) finite observational model. As well as being of theoretical interest, this is of practical significance since the state-of-the-art refinement checking tool FDR4 [5] currently only supports two such models. In particular we study how it is possible to check refinement in a discrete version of the timed failures model that supports Timed CSP.

1 Introduction

In this paper we re-examine part of the Linear-Time spectrum that forms part of the field of study of van Glabbeek in [33,32], specifically the part characterised by *finite linear* observations.

A number of different forms of process calculus have been developed for the modeling of concurrent programs, including Hoare's Communicating Sequential Processes (CSP) [8], Milner's Calculus of Communicating Systems (CCS) [13], and the π-calculus [14]. Unlike the latter two, CSP's semantics are traditionally given in behavioural semantic models coarser than bisimulation, normally ones that depend on linear observations only. Thus, while the immediate range of options possible from a state can be observed, only one of them can be followed in a linear observation and so branching behaviour is not recorded.

David Mestel
Centre for Security, Reliability and Trust, University of Luxembourg
E-mail: david.mestel@uni.lu

A.W. Roscoe
Department of Computer Science, University of Oxford E-mail: Bill.Roscoe@cs.ox.ac.uk

In this paper, we study finite[1] linear-time observational models for CSP; that is, models where all observations considered can be determined in a finite time by an experimenter who can see the visible events a process communicates and the sets of events it can offer in any stable state.[2] While the experimenter can run the process arbitrarily often, he or she can only record the results of individual finite executions. Thus each behaviour recorded can be deduced from a single finite sequence of events and the visible events that link them, together with the sets of events accepted in stable states during and immediately after this *trace*. The representation in the model is determined purely by the set of these linear behaviours that it is possible to observe of the process being examined. We do not introduce new models, but rather introduce a method of embedding them all in the simplest, the traces model.

At least six such models have been actively considered for CSP, but the state-of-the art refinement checking tool, FDR4 [5,6][3], currently only supports two, namely *traces* and *stable failures*. FDR4 also supports the (divergence-strict) failures-divergences model, which is not finite observational.

The question we address in this paper supposes that we have an automated proof tool such as FDR that answers questions about how a process is represented in model A, and asks under what circumstances it is possible to answer questions posed in model B, especially the core property of refinement.

It seems intuitive that if model A records more details than model B, then by looking carefully at how A codes the details recorded in B, the above ought to be possible. We will later see some techniques for achieving this. However it does not intuitively seem likely that we can do the reverse. Surprisingly, however, we find it can be done by the use of process operators for which the coarser model B is not compositional. Sometimes we can use such operators to transform observable features of behaviour that B does not see into ones that it does.

The operator we choose in the world of CSP is the relatively new *priority* operator. While simple to define in operational semantics, this is only compositional over the finest possible finite-linear-observation model of CSP. Priority is not part of "standard" CSP, but is implemented in the model checker FDR4 and greatly extends the expressive power of the notation.

We present first a construction which produces a context $\mathcal{C}$ such that refinement questions in the well-known stable failures model correspond to trace refinement

[1] Models that use a mixture of finite and infinite linear behaviours are also frequently used, the latter involving, *inter alia*, divergences and infinite traces.

[2] The word *stable* here emphasises that the refusal components of failures are only recorded in stable (namely τ-free) states. This distinguishes it both from other models where failures are recorded for other reasons: van Glabbeek (private correspondence) argues that there is in fact an *unstable* failures model (though anything but finite observation), citing [3] as evidence, and several versions have included failures on divergent traces or because of divergence-strictness. We emphasise that all of the models considered in this paper only observe acceptances and refusals in stable states.
This restriction of models to stable refusals and acceptances has been standard in CSP since the earliest days of its theories, and it leads to straightforward models in which one can be comfortable that one is not losing crucial detail by using an LTS as an abstraction of what may well be a concurrent state. We make this assumption here mainly because it is the general practice in the CSP models we are setting out to relate.

[3] See `https://www.cs.ox.ac.uk/projects/fdr/`. At the time of writing there is no major academic paper as a source for FDR4 as opposed to its predecessor FDR3. However the two versions of the tool are the same for the purposes discussed in the present paper.

questions under the application of $\mathcal{C}$. We then generalise this to show (Theorem 1) that a similar construction is possible not only for the six models which have been studied, but also for any sensible finite observational model (where 'sensible' means that the model can be recognised by a finite-memory computer, in a sense which we shall make precise). In fact we can seemingly handle any equivalence determined in a compact way by finitary observations, even though not a congruence.

While at a high level this paper introduces a strategy for translating between differently abstract theories of the same language, in practical terms it is grounded in CSP and Timed CSP, both of which already have substantial literatures, and whose relationships with other ways of studying concurrency have been much studied, not least in van Glabbeek's work as discussed above.

Summary of paper

We first briefly describe the language of CSP, concentrating particularly on the less familiar priority operator and its nuances. We next (Section 3) give an informal description of our construction for the stable failures model. To prove the result in full generality we first (Section 4) give a formal definition of a finite observational model, and of the notion of rationality. We then describe our general constructions (Section 5) before examining the implementation and performance (Section 6) of model shifting on some standard FDR examples: we compare the performance of FDR's native stable failures implementation against the new reduction to traces.

In a major case study (Section 7) we consider $\mathcal{D}$, the discrete version of the timed failures model of Timed CSP, a closely related notation which already depends on priority thanks to its need to enforce the principle of maximal progress. For that we show not only how model shifting can obtain exactly what is needed but also show how timed failures checking can be reduced to its close relative Refusal Testing. For that (Section 8) we use a Timed CSP version of the Sliding Window Protocol as our main example, using it to discuss various specification, performance and optimisation issues.

Finally in the Conclusions we step back and discuss how the framework we have developed depends crucially on *non-compositionality* and might be applicable for notations remote from CSP.

The present paper is a revised and extended version of [12], with the main additions being the study of Timed CSP and the model translation options available there, plus a description of how to include CSP termination $\checkmark$.

2 The CSP language

We provide a brief outline of the language, largely taken from [21]; the reader is encouraged to consult [22] for a more comprehensive treatment.

Throughout, Σ is taken to be a finite nonempty set of communications that are visible and can only happen when the observing environment permits via handshaken communication. The actions of every process are taken from $\Sigma \cup \{\tau\}$, where τ is the invisible internal action that cannot be prevented by the environment. We extend this to $\Sigma \cup \{\tau, \checkmark\}$ if we want the language to allow the successful termination process *SKIP* and sequential compositions as described below. $\checkmark$ is different from other events, because it is observable but not controllable: in that sense it is between a regular Σ event and τ. It only ever appears at the end of traces and from a state which has refusal set Σ and acceptance set $\{\checkmark\}$, although that state is not stable in the usual sense. It thus complicates matters a little, so the reader might prefer to ignore it when first studying this paper. We will later contemplate a second visible event with special semantics: *tock* signifying the passage of time.

The constant processes of our core version of CSP are:

- *STOP* which does nothing—a representation of deadlock.
- **div** which performs (only) an infinite sequence of internal τ actions—a representation of divergence or livelock.
- *CHAOS* which can do anything except diverge, though this absence of divergence is unimportant when studying finite behaviour models for the simple reason that these models do not record divergence.
- *SKIP* which terminates successfully.

The prefixing operator introduces communication:

- $a \to P$ communicates the event a before behaving like P.

There are two main forms of binary choice between a pair of processes:

- $P \sqcap Q$ lets the process decide to behave like P or like Q: this is *nondeterministic* or *internal* choice.
- $P \;\Box\; Q$ offers the environment the choice between the initial Σ-events of P and Q. If the one selected is unambiguous then it continues to behave like the one chosen; if it is an initial event of both then the subsequent behaviour is nondeterministic. The occurence of τ in one of P and Q does *not* resolve $\Box$ (unlike CCS +). This is *external* choice.

A further form of binary choice is the asymmetric $P \rhd Q$, sometimes called *sliding* choice. This offers any initial visible action of P from an unstable (in the combination) state and can (until such an action happens) perform a τ action to Q. It can be re-written in terms of prefix, external choice and hiding. It represents a convenient shorthand way of creating processes in which visible actions happen from an unstable state, so this is not an operator one is likely to use much for building practical systems, rather a tool for analysing how systems can behave. As discussed in [22], to give a full treatment of CSP in any model finer than stable failures, it is necessary to contemplate processes that have visible actions performed from unstable states.

We only have a single parallel operator in our core language since all the usual ones of CSP can be defined in terms of it as discussed in Chapter 2 etc. of [22].

- $P \underset{X}{\parallel} Q$ runs P and Q in parallel, allowing each of them to perform any action in $\Sigma - X$ independently, whereas actions in X must be synchronised between the two.

There are two operators that change the nature of a process's communications.

- $P \setminus X$, for $X \subseteq \Sigma$, *hides* X by turning all P's X-actions into τs.
- $P[\![R]\!]$ applies the *renaming* relation $R \subseteq \Sigma \times \Sigma$ to P: if $(a, b) \in R$ and P can perform a, then $P[\![R]\!]$ can perform b. The domain of R must include all visible events used by P. Renaming by the relation $\{(a, b)\}$ is denoted $[\![{}^{a}/{b}]\!]$.

– Sequential composition $P\,;Q$ allows P to run until it terminates successfully ($\checkmark$). P's $\checkmark$ is turned into τ and then Q is started. So if P and Q respectively have traces $s\hat{}\langle\checkmark\rangle$ and t, then $P\,;Q$ has the trace $s\hat{}t$.

There is another operator that allows one process to follow another:

– $P\,\Theta_{a:A}\,Q$ behaves like P until an event in the set A occurs, at which point P is shut down and Q is started. This is the *throw* operator, and it is important for establishing clean expressivity results.

The final core CSP construct is *recursion*: this can be single or mutual (including mutual recursions over infinite parameter spaces), can be defined by systems of equations or (in the case of single recursion) in-line via the notation $\mu\,p.P$, for a term P that may include the free process identifier p. Recursion can be interpreted operationally as having a τ-action corresponding to a single unwinding. Denotationally, we regard P as a function on the space of denotations, and interpret $\mu\,p.P$ as the least (or sometimes provably unique) fixed point of this function.

We also make use of the *interleaving* operator $|||$, which allows processes to perform actions independently and is equivalent to $\underset{\emptyset}{\|}$, and the process RUN_X, which always offers every element of the set X and is defined by

$$RUN_X = \Box_{x\in X}\, x \to RUN_X$$

This completes our list of operators other than priority. While others, for example $\triangle$ (interrupt) are sometimes used, they are all expressible in terms of the above (see ch.9 of [22]).

2.1 Priority

The priority operator is introduced and discussed in detail in Chapter 20 of [22] as well as [25]. It allows us to specify an ordering on the set of visible events Σ, and prevents lower-priority events from occuring whenever a higher-priority event or τ is available.

The operator is described in [22] and implemented in FDR4 [5]. In these places it has a number of different presentations. [22], which we will chiefly follow, parametrises it by a partial order on the augmented event set $\Sigma^{\tau\checkmark}$ which satisfies the properties that

A τ and (where used) $\checkmark$ have equal priority which is maximal in the order: no action in Σ dominates them and neither dominates the other. Not to follow this restriction would invalidate the principles (i) that in CSP the process which performs τ and becomes P (which we abuse notation and write as $\tau \to P$) is observationally equivalent in all contexts to P, and (ii) any $\checkmark$ action can be replaced harmlessly with the combination of a τ (from whatever state the $\checkmark$ came from) leading to an unconditional $\checkmark$.

B If $b < a$ for $a, b \in \Sigma$ then $b < \tau$ (and therefore $b < \checkmark$). Otherwise $(b \to STOP) \,\Box\, (a \to STOP)$ and $(b \to STOP) \,\Box\, (\tau \to a \to STOP)$ would be observationally different when the given priority was applied, contrary to the same principle discussed above.

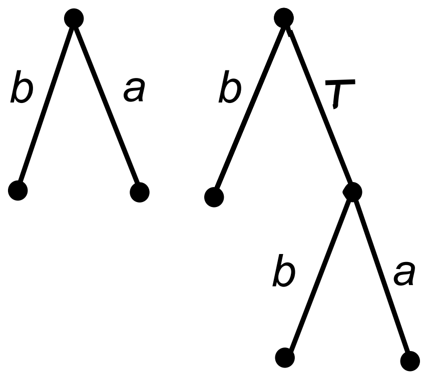

Fig. 1 Two processes that should have the same traces in every context.

The operational semantics of $(b \to STOP) \,\Box\, (a \to STOP)$ and $(b \to STOP) \,\Box\, (\tau \to a \to STOP)$ are illustrated in Fig. 1. We want these to look the same as each other externally whatever priority operator is applied, and this would not be the case if $a, b > \tau$ (for (A)) or $a > b$ (for (B)) and b and τ are incomparable.

We will call such an order on Σ^τ or $\Sigma^{\tau,\checkmark}$ a *priority order*

In implementing priority for FDR our group was conscious of not expecting the user necessarily to have a detailed understanding of τ and preferring to avoid expressing τ and $\checkmark$ in the CSP$_M$ language. (This would be necessary to tabulate partial orders involving them.)

Thus the first (FDR2) implementation of anything like the full priority operator simply allowed the user to give a ranked series of sets of members of Σ with successively lower priority, with the first (which could be empty) being the prioritised events incomparable to $\tau, \checkmark$, and any event not appearing in any of the sets being unaffected by priority. Thus is was written

$$prioritise(P, A_1, \ldots, A_n)$$

so that, for example $prioritise(P, \emptyset, \{c\})$ is the process where c is banned from happening from an unstable state: one which can perform τ or $\checkmark$. This operator continues to be supported in FDR4 except that the operator is now better typed (in the sense that it fits in with the CSP$_M$ type checker introduced in FDR3): it has become $prioritise(P, \langle A_1, \ldots, A_n \rangle)$.

While it is possible, using multiple layers of this operator, to express a general priority order on $\Sigma^{\tau\checkmark}$, it is scarcely intuitive or efficient to do so. Therefore FDR4 implements an operator *prioritisepo* with exactly the same expressive power as the general one by representing a priority order as the combination of a partial order $\leq$ on Σ and a subset X of the maximal elements of Σ under it: the ones incomparable to τ and $\checkmark$. Such a pair $(\leq, X)$ corresponds to the priority order $\leq'$ in which $x <' y$ if and only if either $x < y$ or $x \in \Sigma - X \wedge y \in \{\tau, \checkmark\}$.

It should be clear that this correspondence between the permitted pairs $(\leq, X)$ and the orders $\leq'$ on $\Sigma^{\tau\checkmark}$ satisfying (A) and (B) is bijective. In the rest of this paper we will most often use the **prioritise**$(P, \leq, X)$ format since it relates most closely to the FDR4 syntax, though in defining semantics it is often useful to have the corresponding $\leq'$ around also, as in the following basic definition of the operational semantics.

$$\frac{P \xrightarrow{x} P'}{\mathbf{prioritise}(P, \leq, X) \xrightarrow{x} \mathbf{prioritise}(P', \leq, X)}(x \in \{\tau, \checkmark\}$$

$$\frac{P \xrightarrow{a} P' \wedge \forall\, x \neq a.a \leq' x \Rightarrow x \notin \mathit{initials}(P)}{\mathbf{prioritise}(P, \leq, X) \xrightarrow{a} \mathbf{prioritise}(P', \leq, X)}$$

In other words, priority never blocks τ or $\checkmark$ actions, and if a is any other action it is blocked when P can perform a higher (as judged by $\leq'$) priority action.

prioritise makes enormous contributions to the expressive power of CSP as explained in [25], meaning that CSP+**prioritise** can be considered a universal language for a much wider class of operational semantics than the *CSP-like* class described in [23,22].

It should not therefore be surprising that **prioritise** is not compositional over denotational finite observation models other than the most precise model, as we will discuss below. So we think of it as an optional addition to CSP rather than an integral part of it; when we refer below to particular types of observation as giving rise to valid models for CSP, we will mean CSP without priority.

3 Example: the stable failures model

We introduce our model shifting construction using the *stable failures* model: we will produce a context $\mathcal{C}$ such that for any processes P, Q, we have that Q refines P in the stable failures model if and only $\mathcal{C}[Q]$ refines $\mathcal{C}[P]$ in the traces model.

3.1 The traces and failures models

The *traces* model $\mathcal{T}$ is familiar from both process algebra and automata theory, and represents a process by the set of (finite) strings of events it is able to accept. Thus, for the time being ignoring $\checkmark$, each process is associated (for fixed alphabet Σ) to a subset of Σ^* the set of finite words over Σ (plus words of the form $w\langle\checkmark\rangle$ if we allow SKIP and sequential composition). The *stable failures* model $\mathcal{F}$ also records sets X of events that the process is able to stably refuse after a trace s (that is, the process is able after trace s to be in a state where no τ events are possible, and where the set of initial events is disjoint from X). Thus a process is associated to a subset of $\Sigma^* \times (\mathcal{P}(\Sigma) \cup \{\bullet\})$, where $\bullet$ represents the absence of a recorded refusal set.[4] We would add the symbol $\checkmark$ to this set of possible second components when including termination. Note that recording $\bullet$ does not imply that there is no refusal to observe, simply that we have not observed stability. The

[4] This is equivalent to the standard presentation in which a process is represented by a subset of Σ^* and one of $\Sigma^* \times \mathcal{P}(\Sigma)$: the trace component is just $\{s : (s, \bullet) \in \mathcal{F}(P)\}$.

observation of the refusal $\emptyset$ implies that the process can be stable after the present trace, whereas observing $\bullet$ does not.

In any model $\mathcal{M}$, we say that Q *$\mathcal{M}$-refines* P, and write $P \sqsubseteq_M Q$, if the set associated to Q is a subset of that corresponding to P.

Because $\checkmark$ can be seen, but happens automatically, we need to distinguish a process like *SKIP* which must terminate from one that can but may not like $STOP \sqcap SKIP$. After all if these are substituted for P in $P\,;Q$ we get processes equivalent to Q and $STOP \sqcap Q$. However the state that accepts $\checkmark$ can be thought of as being able to refuse the rest of the visible events Σ, since it can terminate all by itself.

3.2 Model shifting for the stable failures model

We first consider this without $\checkmark$. The construction is as follows:

Lemma 1 *For each finite alphabet Σ there exists a context $\mathcal{C}$ (over an expanded alphabet) such that for any processes P and Q we have that $P \sqsubseteq_F Q$ if and only if $\mathcal{C}[P] \sqsubseteq_T \mathcal{C}[Q]$.*

Proof
Step 1: We use priority to produce a process (over an expanded alphabet) that can communicate an event x' if and only if the original process P is able to stably refuse x.

This is done by expanding the alphabet Σ to $\Sigma \cup \Sigma'$ (where Σ' contains a corresponding primed event x' for every event $x \in \Sigma$), and prioritising with respect to the partial order which prioritises each x over the corresponding x' and makes τ incomparable to x and greater than x'.

We must also introduce an event *stab* to signify the observation of stability (i.e. no τ is possible in this state) without requiring any refusals to be possible. This is necessary in order to be able to record an empty refusal set. The priority order $\leq_1$ is then the above (i.e. $x' < x$ for all $x \in \Sigma$) extended by making *stab* less than only τ and independent of all x and x'.

We can now fire up these new events as follows:

$$\mathcal{C}_1[P] = \mathbf{prioritise}(P \mid\mid\mid RUN_{\Sigma' \cup \{stab\}}, \leq_1, \Sigma).$$

This process has a state ξ' for each state ξ of P, where ξ' has the same unprimed events (and corresponding transitions) as ξ. Furthermore ξ' can communicate x' just when ξ is stable and can refuse x, and *stab* just when ξ is stable.

Step 2: We now recall that the definition of the stable failures model only allows a refusal set to be recorded at the *end* of a trace, and is not interested in (so does not record) what happens after the refusal set.

We gain this effect by using a regulator process to prevent a primed event (or *stab*) from being followed by an unprimed event. Let

$$\begin{aligned} UNSTABLE &= \square_{x \in \Sigma}\, x \to UNSTABLE \\ &\quad \square\ \square_{x \in \Sigma' \cup \{stab\}}\, x \to STABLE \\ STABLE &= \square_{x \in \Sigma' \cup \{stab\}}\, x \to STABLE, \end{aligned}$$

and define $\mathcal{C}$ by

$$\mathcal{C}[P] = \mathcal{C}_1[P] \underset{\Sigma\cup\Sigma'\cup\{stab\}}{\parallel} \mathit{UNSTABLE}.$$

A trace of $\mathcal{C}[P]$ consists of: firstly, a trace s of P; followed by, if P can after s be in a stable state, then for some such state σ_0 any string formed from the events that can be refused in σ_0, together with *stab*. The lemma clearly follows. ∎

As discussed in Step 1 above, the process $\mathcal{C}_1[P]$ has the same number of states as P. The process *UNSTABLE* has two states (*STABLE* and *UNSTABLE*; note that that external choice operator does not introduce new states). Hence the process $\mathcal{C}[P]$ has at most twice the number of states of P.[5]

It is clear that any such context must involve an operator that is not compositional over traces, for otherwise we would have $P \sqsubseteq_T Q$ implies $\mathcal{C}[P] \sqsubseteq_T \mathcal{C}[Q]$, which is equivalent to $P \sqsubseteq_F Q$, and this is not true for general P and Q (consider for instance $P = a \to \mathit{STOP}$, $Q = (a \to \mathit{STOP}) \sqcap \mathit{STOP}$). It follows that only contexts which like ours involve priority or some operator with similar status can achieve this.

Adding ✓ to the model causes a few issues with the above. For one thing it creates a refusal (namely of everything except ✓) from what could be an unstable state, namely a state that can perform ✓ and perhaps also a τ. And secondly we need to find an effective way of making processes show their refusal of ✓, and their refusal of all events other than ✓, when respectively appropriate. (The syntax ✓ is not part of the CSP language: ✓ only appears through *SKIP*.) One way of doing these things is to add to the state space so that termination goes through multiple stages. Create a new event *term* and consider $P\,;\mathit{term} \to \mathit{SKIP}$. This performs any behaviour of P except that all ✓s of P become τs and lead to the state $\mathit{term} \to \mathit{SKIP}$. That of course is a stable state. If we now (treating *term* as a member of Σ) apply $\mathcal{C}$ as defined above, this will be able to perform term' in any stable state that cannot terminate, and will perform every a' event other than term' every time it reaches the state $\mathit{term} \to \mathit{SKIP}$. Thus if we define

$$\mathcal{C}^{\checkmark}(P) = \mathcal{C}(P\,;\mathit{term} \to \mathit{SKIP}) \setminus \{\mathit{term}\}$$

we get exactly the decorated traces we might have expected from the stable failures representation of P except that instead of having an event $\checkmark'$ we have term'.

[5] Note, however, that to perform refinement checks, the FDR tool constructs the 'normalisation' (determinisation) of the specification process, and it is sometimes possible for the *normalised* form of $\mathcal{C}[P]$ to have up to a factor of $2^{|\Sigma|}$ more states than the normalised form of P (both of which may be exponentially larger than P itself, though the normal form of $\mathcal{C}[P]$ is bounded in size by $4^{|P|}$). This is closely related to the famous result of Kannellakis and Smolka [10] that checking equivalence in a CSP-style model is *PSPACE*-hard, and is discussed and illustrated in [19]. In practice since the specification process is usually far smaller than the implementation process this has not often been a constraint for FDR users. Now consider the normal forms of P and $\mathcal{C}_1[P]$. The latter will have a state corresponding to each state of the former, but in addition will have states only reachable via events a' signifying the observation of refusals. Consider the cases where a single normal form state S of P has M maximal refusals X_i (the states of the $\mathcal{F}$ normal form are marked with perhaps multiple such sets). It is clear that there will be a normal form state of $\mathcal{C}_1[P]$ corresponding to S itself and to each of the X_i. But there is also one for all intersections of X_i. Thus it was always inevitable that the number of trace normal form states of $\mathcal{C}_1[P]$ would sometimes be larger than the number of failures normal form states of P, but in fact this difference can be problematic. We will discuss this further in Section 6.2.

4 Semantic models

In order to generalise this construction to arbitrary finite observational semantic models, we must give formal definitions not only of particular models but of the very notion of a finite observational model.

4.1 Finite observations

We consider only models arising from *finite linear observations.* Intuitively, we postulate that we are able to observe the process performing a finite number of visible actions, and that, where the process was stable (unable to perform a τ) immediately before an action, we are able to observe the precise *acceptance set* of actions it was willing to perform.

Note that there cannot be two separate stable states before visible event b without another visible event c between them, even though it is possible to have many visible events between stable states. Thus it makes no sense to record two separate refusals or acceptance sets between consecutive visible events. Similarly it does not make sense to record both an acceptance and a refusal, since observing an acceptance set means that recording a refusal conveys no extra information: if acceptance A is observed then no other is seen before the next visible event, and observable refusals are exactly those disjoint from A.

The main difference between an acceptance set and a refusal set, beyond the obvious one that the first records a set of events the process can do and the second a set it cannot do, is that refusal sets are subset closed and acceptance sets are not superset closed. Thus refusals are not just complements of acceptances. An acceptance represents the exact set of events offered by a stable state, whereas a refusal is any set disjoint from an acceptance. Thus, as is well known to those who have studied refusal-style models, the set of refusals after a given trace is always subset closed. However acceptances are not closed under either superset or subset: the process $STOP \sqcap (a \to STOP \,\square\, b \to STOP)$ has initial acceptances $\{\emptyset, \{a, b\}\}$.

We are unable to finitely observe *instability*: the most we are able to record from an action in an unstable state is that we did not *observe* stability. Thus in any context where we can observe stability we can also fail to observe it by simply not looking.

We take models to be defined over finite alphabets Σ, and take an arbitrary linear ordering on each finite Σ which we refer to as *alphabetical.*

The most precise finite observational model is that considering all finite linear observations, and is denoted $\mathcal{FL}$:

Definition 1 The set of *finite linear observations* over an alphabet Σ is

$$\mathcal{FL}^{\Sigma} := \{\langle A_0, a_1, A_1, \ldots, A_{n-1}, a_n, A_n\rangle : n \in \mathbb{N}, a_i \in \Sigma, A_i \subseteq \Sigma \text{ or } A_i = \bullet\},$$

where the a_i are interpreted as a sequence of communicated events, and the A_i denote stable acceptance sets, or in the case of $\bullet$ failure to observe stability. Let the set of such observations corresponding to a process P be denoted $\mathcal{FL}_{\Sigma}(P)$. This needs to be extended to encompass final $\checkmark$s if we want to include termination.

(Sometimes we will drop the Σ and just write $\mathcal{FL}(P)$).

More formally, $\mathcal{FL}(P)$ can be defined inductively; for instance

$$\mathcal{FL}(P \,\Box\, Q) := \{\langle A \cup B\rangle\hat{}\alpha, \langle A \cup B\rangle\hat{}\beta : \langle A\rangle\hat{}\alpha \in \mathcal{FL}(P), \langle B\rangle\hat{}\beta \in \mathcal{FL}(Q)\}$$

(where $X \cup \bullet := \bullet$ for any set X). See Section 11.1.1 of [22] for further details.

For observations $s, t \in \mathcal{FL}^{\Sigma}$, we say that s is *extended* by t, and write $s \leq t$, if any process which can produce the observation t can also produce s: that is, if s is a prefix of t, perhaps with some sets A_i replaced by $\bullet$. This is a partial order on $\mathcal{FL}^{\Sigma}$, with respect to which we have that $\mathcal{FL}_{\Sigma}(P)$ is downwards-closed for any process P.

The definition of priority over $\mathcal{FL}$ (accommodating final $\checkmark$s) is as follows. **prioritise**$(P, \leq, X)$ is, with $\leq$ extended as before to $\leq'$ on the whole of $\Sigma \cup \{\tau\}$ by making all elements not in X incomparable to all others

$$\begin{array}{c}\{\langle A_0, b_1, A_1, \ldots, A_{n-1}, b_n, A_n\rangle \mid \langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, Z_n\rangle \in P\}\\ \cup \\ \{\langle A_0, b_1, A_1, \ldots, A_{n-1}, b_n, \bullet, \checkmark\rangle \mid \langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, \bullet, \checkmark\rangle \in P\}\end{array}$$

where for each i one of the following holds:

- b_i is maximal under $\leq'$ and $A_i = \bullet$ (so there is no condition on Z_i except that it exists).
- b_i is not maximal under $\leq'$ and $A_{i-1} = \bullet$ and Z_{i-1} is not $\bullet$ and neither does Z_{i-1} contain any $c > b_i$.
- 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 $A_{i-1} \neq \bullet$, $a_i \in A_{i-1}$.

Note that for a given b_i, all of A_{i-1}, A_i, Z_{i-1} and Z_i are directly relevant to it: the Z's are what are seen to be accepted before and after b_i in P, and the As in **prioritise**$(P, \leq, X)$.

We can interpret the above clauses as follows. We are looking at when a typical behaviour $\langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, Z_n\rangle$ of P or respectively one with $\checkmark$ at the end proves that $\langle A_0, b_1, A_1, \ldots, A_{n-1}, b_n, A_n\rangle$ belongs to **prioritise**$(P, \leq, X)$. Recall that τs are not amongst the events recorded here as b_i or members of A_i, Z_i, but that the presence of a non-$\bullet$ acceptance implies that no τ is possible from the state that witnessed it.

- If b_i is maximal and not dominated by τ, then the prioritised process can certainly perform it from any reachable state **prioritise**$(S, \leq, X)$ where S can perform b_i, so we need no condition on Z_{i-1}.
- However if b_i is not maximal under $\leq'$, then firstly it can only happen when P was known to be in a stable state (i.e. Z_{i-1} is not $\bullet$) because τ or $\checkmark$ would pre-empt it, and secondly P was not accepting (in Z_{i-1}) any event of Σ that is of higher priority than b_i.
- In cases where b_i is possible and followed by $\bullet$ in P, then b_i is followed by $\bullet$ in the result.
- Where b_i is possible and followed by Z_i (not $\bullet$) in P, the prioritised process accepts precisely those events of Z_i which are not dominated in the priority order by other members of Z_i. Namely, $A_i = \{a \in Z_i \mid \neg\exists\, b \in Z_i.b > a\}$,

Thus for $\langle Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, Z_n\rangle$ to give rise to any behaviour of **prioritise**$(P, \leq, X)$, the b_i that are not maximal require proof of stability before them in Z_{i-1}, and an inability to perform any event that would pre-empt b_i under priority.

This is not possible for the other studied finite behaviour models of CSP: the statement that it is for refusal testing $\mathcal{RT}$ in [22] is not true, though it is possible for some priority orders $\leq'$ including those needed for maximal progress in timed modelling of the sort we will see later. For more discussion on this point see [25].

4.2 Finite observational models

We consider precisely the models which are derivable from the observations of $\mathcal{FL}$, which are well-defined in the sense that they are compositional over CSP syntax (other than priority), and which respect extension of the alphabet Σ.

Definition 2 A finite observational *pre-model* $\mathcal{M}$ consists for each (finite) alphabet Σ of a set of *observations*, $\mathrm{obs}_\Sigma(\mathcal{M})$, together with a relation $\mathcal{M}_\Sigma \subseteq \mathcal{FL}_\Sigma \times \mathrm{obs}_\Sigma(\mathcal{M})$. The representation of a process P in $\mathcal{M}_\Sigma$ is denoted $\mathcal{M}_\Sigma(P)$, and is given by

$$\mathcal{M}_\Sigma(P) := \mathcal{M}_\Sigma(\mathcal{FL}_\Sigma(P)) = \{y \in \mathrm{obs}_\Sigma(\mathcal{M}) : \exists\, x \in \mathcal{FL}_\Sigma(P).(x, y) \in \mathcal{M}_\Sigma\}.$$

For processes P and Q over alphabet Σ, if we have $\mathcal{M}_\Sigma(Q) \subseteq \mathcal{M}_\Sigma(P)$ then we say Q *$\mathcal{M}$-refines* P, and write $P \sqsubseteq_M Q$.

(As before we will sometimes drop the Σ).

Note that this definition is less general than if we had defined a pre-model to be any equivalence relation on $\mathcal{P}(\mathcal{FL}_\Sigma)$. For example, the equivalence relating sets of the same cardinality has no corresponding pre-model. Definition 2 agrees with that sketched in [22].

To illustrate this and the following definitions, we will take as a running example the *traces* model. This is the coarsest non-trivial model, and its observations correspond to the language of the process viewed as a nondeterministic finite automaton (NFA). We thus have $\mathrm{obs}_\Sigma(\mathcal{T}) = \Sigma^*$, the set of finite words over Σ, and $\mathcal{T}_\Sigma$ is the relation which relates the observation $\langle A_0, a_1, A_1, \ldots, a_n, A_n\rangle$ to the string $a_1 \ldots a_n$.

Without loss of generality, $\mathcal{M}_\Sigma$ does not identify any elements of $\mathrm{obs}_\Sigma(\mathcal{M})$; that is, we have $\mathcal{M}_\Sigma^{-1}(x) = \mathcal{M}_\Sigma^{-1}(y)$ only if $x = y$ (otherwise quotient by this equivalence relation). Subject to this assumption, $\mathcal{M}_\Sigma$ induces a partial order on $\mathrm{obs}_\Sigma(\mathcal{M})$, inherited from the extension order on $\mathcal{FL}^\Sigma$:

Definition 3 The partial order *induced by $\mathcal{M}_\Sigma$ on* $\mathrm{obs}_\Sigma(\mathcal{M})$ is given by: $x \leq y$ if and only if for all $b \in \mathcal{M}_\Sigma^{-1}(y)$ there exists $a \in \mathcal{M}_\Sigma^{-1}(x)$ with $a \leq b$.

Observe that for any process P it follows from this definition that $\mathcal{M}(P)$ is downwards-closed with respect to this partial order (since $\mathcal{FL}(P)$ is downwards-closed).

For the traces model, we have that the partial order induced by $\mathcal{T}_\Sigma$ on $\mathrm{obs}_\Sigma(\mathcal{T}) = \Sigma^*$ is just the prefix order.

Definition 4 A pre-model $\mathcal{M}$ is *compositional* if for all CSP operators $\bigoplus$, say of arity k, and for all processes $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_k$ such that $\mathcal{M}(P_i) = \mathcal{M}(Q_i)$ for all i, we have

$$\mathcal{M}\left(\bigoplus(P_i)_{i=1\ldots k}\right) = \mathcal{M}\left(\bigoplus(Q_i)_{i=1\ldots k}\right).$$

This means that the operator defined on processes in obs($\mathcal{M}$) by taking the pushforward of $\bigoplus$ along $\mathcal{M}$ is well-defined: for any sets $X_1, \ldots, X_k \subseteq \text{obs}(\mathcal{M})$ which correspond to the images of CSP processes, take processes $P_1, \ldots, P_k$ such that $X_i = \mathcal{M}(P_i)$, and let

$$\bigoplus(X_i)_{i=1\ldots k} = \mathcal{M}\left(\bigoplus(P_i)_{i=1\ldots k}\right).$$

Definition 4 says that the result of this does not depend on the choice of the P_i.

Note that it is not necessary to require the equivalent of Definition 4 for recursion in the definition of a model, because of the following lemma which shows that least fixed point recursion is automatically well-defined (and formalises some arguments given in [22]):

Lemma 2 *Let $\mathcal{M}$ be a compositional pre-model. Let $\mathcal{C}_1, \mathcal{C}_2$ be CSP contexts, such that for any process P we have $\mathcal{M}(\mathcal{C}_1[P]) = \mathcal{M}(\mathcal{C}_2[P])$. Let the least fixed points of $\mathcal{C}_1$ and $\mathcal{C}_2$ (viewed as functions on $\mathcal{P}(\mathcal{FL})$ under the subset order) be P_1 and P_2 respectively. Then $\mathcal{M}(P_1) = \mathcal{M}(P_2)$.*

Proof Using the fact that CSP contexts induce Scott-continuous functions on $\mathcal{P}(\mathcal{FL})$ (see [8], Section 2.8.2), the Kleene fixed point theorem gives that $P_i = \bigcup_{n=0}^{\infty} \mathcal{C}_i^n(\perp)$. Now any $x \in \mathcal{M}(P_1)$ is in the union taken up to some finite N, and since finite unions correspond to internal choice, and $\perp$ to the process **div**, we have that the unions up to N of $\mathcal{C}_1$ and $\mathcal{C}_2$ agree under $\mathcal{M}$ by compositionality. Hence $x \in \mathcal{M}(P_2)$, so $\mathcal{M}(P_1) \subseteq \mathcal{M}(P_2)$. Similarly $\mathcal{M}(P_2) \subseteq \mathcal{M}(P_1)$.

Definition 5 A pre-model $\mathcal{M}$ is *extensional* if for all alphabets $\Sigma_1 \subseteq \Sigma_2$ we have that $\text{obs}_{\Sigma_1}(\mathcal{M}) \subseteq \text{obs}_{\Sigma_2}(\mathcal{M})$, and $\mathcal{M}_{\Sigma_2}$ agrees with $\mathcal{M}_{\Sigma_1}$ on $\mathcal{FL}(\Sigma_1) \times \text{obs}_{\Sigma_1}(\mathcal{M})$.

Definition 6 A pre-model is a *model* if it is compositional and extensional.

In this setting, we now describe the five main finite observational models coarser than $\mathcal{FL}$: traces, stable failures, revivals, acceptances and refusal testing.

4.2.1 The traces model

Recording the remarks from the previous section, we have the definition of the traces model.

Definition 7 The *traces* model, $\mathcal{T}$, is given by

$$\text{obs}_\Sigma(\mathcal{T}) = \Sigma^*,\ \mathcal{T}_\Sigma = \mathit{trace}_\Sigma$$

where *trace* is the relation which relates the observation $\langle A_0, a_1, A_1, \ldots, a_n, A_n \rangle$ to the string $a_1 \ldots a_n$.

4.2.2 Failures

The traces model gives us information about what a process is *allowed* to do, but it in some sense tells us nothing about what it is *required* to do. In particular, the process *STOP* trace-refines any other process.

In order to specify liveness properties, we can incorporate some information about the events the process is allowed to refuse, beginning with the *stable failures* model. Intuitively, this captures traces s, together with the sets of events the process is allowed to stably refuse after s.

Definition 8 The *stable failures* model, $\mathcal{F}$, is given by

$$\mathrm{obs}_\Sigma(\mathcal{F}) = \Sigma^* \times (\mathcal{P}(\Sigma) \cup \{\bullet\}),\ \mathcal{F}_\Sigma = \mathit{fail}_\Sigma,$$

where fail_Σ relates the observation $\langle A_0, \ldots, a_n, A_n\rangle$ to all pairs $(a_1 \ldots a_n, X)$, for all $X \subseteq \Sigma - A_n$ if $A_n \neq \bullet$, and for $X = \bullet$ otherwise.

4.2.3 Revivals

The next coarsest model, first introduced in [21], is the *revivals* model. Intuitively this captures traces s, together with sets X that can be stably refused after s, and events a (if any) that can then be accepted.

Definition 9 The *revivals* model, $\mathcal{R}$, is given by

$$\mathrm{obs}_\Sigma(\mathcal{R}) = \Sigma^* \times (\mathcal{P}(\Sigma) \cup \{\bullet\}) \times (\Sigma \cup \{\bullet\}),\ \mathcal{R}_\Sigma = \mathit{rev}_\Sigma),$$

where rev_Σ relates the observation $\langle A_0, a_1, \ldots, a_{n-1}, A_{n-1}, a_n, A_n\rangle$ to

(i) the triples $(a_1 \ldots a_{n-1}, X, a_n)$, for all $X \subseteq \Sigma - A_{n-1}$ if $A_{n-1} \neq \bullet$ and for $X = \bullet$ otherwise, and
(ii) the triples $(a_1 \ldots a_n, X, \bullet)$, for all $X \subseteq \Sigma - A_n$ if $A_n \neq \bullet$ and for $X = \bullet$ otherwise.

A finite linear observation is related to all triples consisting of: its initial trace; a stable refusal that could have been observed, or $\bullet$ if the original observation did not observe stability; and optionally (part (i) above) a single further event that can be accepted.

4.2.4 Acceptances

All the models considered up to now refer only to sets of refusals, which in particular are closed under subsets. The next model, *acceptances* (also known as 'ready sets'), refines the previous three and also considers the precise sets of events that can be stably accepted at the ends of traces.

Definition 10 The *acceptances* model, $\mathcal{A}$, is given by

$$\mathrm{obs}_\Sigma(\mathcal{A}) = \Sigma^* \times (\mathcal{P}(\Sigma) \cup \{\bullet\}),\ \mathcal{A}_\Sigma = \mathit{acc}_\Sigma,$$

where acc_Σ relates the observation $\langle A_0, a_1, \ldots, a_n, A_n\rangle$ to the pair $(a_1 \ldots a_n, A_n)$.

It is convenient to note here that, just as we were able to use a' as a cipher for the refusal of a when model shifting, we can introduce a second one a'' as a cipher for stable acceptance of a: it is performed (without changing the state) just when a' is stably refused. We will apply this idea and discuss it further below.

4.2.5 Refusal testing

The final model we consider is that of *refusal testing*, first introduced in [17]. This refines $\mathcal{F}$ and $\mathcal{R}$ by considering an entire history of events and stable refusal sets. It is incomparable to $\mathcal{A}$, because it does not capture precise acceptance sets.

Definition 11 The *refusal testing* model, $\mathcal{RT}$, is given by

$$\mathrm{obs}_{\Sigma}(\mathcal{RT}) = \{\langle X_0, a_1, X_1, \ldots, a_n, X_n\rangle : n \in \mathbb{N}, a_i \in \Sigma, X_i \subseteq \Sigma \mathtt{or} X_i = \bullet\}$$

$$\mathcal{RT}_{\Sigma} = rt_{\Sigma},$$

where rt_{Σ} relates the observation $\langle A_0, \ldots, a_n, A_n\rangle$ to $\langle X_0, \ldots, a_n, X_n\rangle$, for all $X_i \subseteq \Sigma - A_i$ if $A_i \neq \bullet$, and for $X_i = \bullet$ otherwise.

The correct way to handle $\checkmark$, if needed, in any of these models is to add to the respective transformation in exactly the same way we did for stable failures. This is to be expected because $\checkmark$ only ever happens at the end of traces. Clearly we will need to use $term''$ as a cipher for $\checkmark''$ in appropriate cases.

4.3 Rational models

We will later on wish to consider only models $\mathcal{M}$ for which the correspondence between $\mathcal{FL}$-observations and $\mathcal{M}$ observations is decidable by a finite memory computer. We will interpret this notion as saying that the relation $\mathcal{M}_{\Sigma}$ corresponds to the language accepted by some finite state automaton. In order to do this, we must first decide how to convert elements of $\mathcal{FL}_{\Sigma}$ to words in a language. We do this in the obvious way (the reasons for using fresh variables to represent the A_i will become apparent in Section 5).

Definition 12 The *canonical encoding* of $\mathcal{FL}_{\Sigma}$ is over the alphabet $\Xi := \Sigma \cup \Sigma'' \cup Sym$, where $\Sigma'' := \{a'' : a \in \Sigma\}$ and $Sym = \{\langle, \rangle, `,', \bullet\}$.[6] It is given by writing out the representation given in Definition 1, where sets A_i are expressed by listing the elements of Σ'' corresponding to the members of A_i in alphabetical order (so an example of an encoded element would be $\langle ab, a, \bullet, c, bc, b, abc\rangle$). We denote this encoding by $\phi_{\Sigma} : \mathcal{FL}_{\Sigma} \to \Xi^*$.

We now define a model to be *rational* (borrowing a term from automata theory) if its defining relation can be recognised (when suitably encoded) by some nondeterministic finite automaton.

Definition 13 A model $\mathcal{M}$ is *rational* if for every alphabet Σ, there is some finite alphabet Θ, a map $\psi_{\Sigma} : \mathrm{obs}_{\Sigma}(\mathcal{M}) \to \Theta^*$ and a (nondeterministic) finite automaton $\mathcal{A}$ such that

(i) $\mathcal{A}$ recognises $\{(\phi_{\Sigma}(x), \psi_{\Sigma}(y)) : (x, y) \in \mathcal{M}_{\Sigma}\}$, in the sense described below, and
(ii) ψ_{Σ} is *order-reflecting* (that is, $\psi_{\Sigma}(x) \leq \psi_{\Sigma}(y)$ only if $x \leq y$),

[6] Note that this somewhat unsatisfactory notation denotes a set of four elements: the angle brackets $\langle$ and $\rangle$, the comma , and the symbol $\bullet$.

where Θ^* is given the prefix partial order, and $\mathrm{obs}_\Sigma(\mathcal{M})$ is given the partial order induced by $\mathcal{M}_\Sigma$ (in the sense of Definition 3).

What does it mean for an automaton to 'recognise' a relation?

Definition 14 For alphabets Σ and T, a relation $\mathcal{R} \subseteq \Sigma^* \times T^*$ is *recognised* by an automaton $\mathcal{A}$ just when:

(i) The event-set of $\mathcal{A}$ is $\mathrm{left}.\Sigma \cup \mathrm{right}.T$, and
(ii) For any $s \in \Sigma^*, t \in T^*$, we have $s\mathcal{R}t$ if and only if there is some interleaving of $\mathrm{left}.s$ and $\mathrm{right}.t$ accepted by $\mathcal{A}$.

Note that recognisability in the sense of Definition 14 is easily shown to be equivalent to the common notion of recognisability by a *finite state transducer* given for instance in [31], but the above definition is more convenient for our purposes. Note also that $\mathcal{FL}$ itself (viewing $\mathcal{FL}_\Sigma$ as the diagonal relation) is trivially rational.

Lemma 3 *The models $\mathcal{T}, \mathcal{F}, \mathcal{R}, \mathcal{A}$ and $\mathcal{RT}$ are rational.*

Proof
By inspection of Definitions 7–11. We take $\Theta = \Sigma \cup \Sigma' \cup \Sigma'' \cup Sym$, with Σ'' and the expression of acceptance sets as in the canonical encoding of $\mathcal{FL}$, and refusal sets expressed in the corresponding way over $\Sigma' := \{a' : a \in \Sigma\}$. It is easy to see that all the resulting relations are rational; for instance the transducer for the traces model echoes elements of Σ and ignores all other inputs. ▮

Note that not all relations are rational. For instance, the 'counting relation' mapping each finite linear observation to its length is clearly not rational. We do not know whether the additional constraint of being a finite observational model necessarily implies rationality; however, no irrational models are known. We therefore tentatively conjecture: that *every finite observational model is rational.*

5 Model shifting

We now come to the main substance of this paper: we prove results on 'model shifting', showing that there exist contexts allowing us to pass between different semantic models and the basic traces model. The main result is Theorem 1, which shows that this is possible for any rational model.

5.1 Model shifting for $\mathcal{FL}$

We begin by proving the result for the finest model, $\mathcal{FL}$. We show that there exists a context $\mathcal{C}_{\mathcal{FL}}$ such that for any process P, the finite linear observations of P correspond to the traces of $\mathcal{C}_{\mathcal{FL}}(P)$.

Lemma 4 (Model shifting for $\mathcal{FL}$) *For every alphabet Σ, there exists a context $\mathcal{C}_{\mathcal{FL}}$ over alphabet $T := \Sigma \cup \Sigma' \cup \Sigma'' \cup \{done\}$, and an order-reflecting map $\pi : \mathcal{FL}_\Sigma \to T^*$ (with respect to the extension partial order on $\mathcal{FL}_\Sigma$ and the prefix partial order on T^*) such that for any process P over Σ we have $\mathcal{T}(\mathcal{C}_{\mathcal{FL}}[P]) = \mathrm{pref}(\pi(\mathcal{FL}(P)))$ (where $\mathrm{pref}(X)$ is the prefix-closure of the set X).*

Proof
We will use the unprimed alphabet Σ to denote communicated events from the original trace, and the double-primed alphabet Σ'' to denote (members of) stable acceptances. Σ' will be used in an intermediate step to denote refusals, and *done* will be used to distinguish $\emptyset$ (representing an empty acceptance set) from $\bullet$ (representing a failure to observe anything).

Step 1: We first produce a process which is able to communicate events x_i', just when the original process can stably refuse the corresponding x_i. Define the partial order $\leq_1 = \langle x' <_1 x : x \in \Sigma \rangle$, which prevents refusal events when the corresponding event can occur.

Let the context $\mathcal{C}_1$ be given by

$$\mathcal{C}_1[X] = \textbf{prioritise}(X \;|||\; RUN_{\Sigma'}, \leq_1, \Sigma).$$

Note that the third argument prevents primed events from occurring in unstable states.

Step 2: We now similarly introduce acceptance events, which can happen in stable states when the corresponding refusal can't. The crucial difference between a and a'' is that a usually changes the underlying process state, whereas a'' leaves it alone. a'' means that P can perform a from its present stable state, but does not explore what happens when it does.

Similarly define the partial order $\leq_2 = \langle x'' <_2 x' : x \in \Sigma \rangle$, which prevents acceptance events when the corresponding refusal is possible. Let the context $\mathcal{C}_2$ be defined by

$$\mathcal{C}_2[X] = \textbf{prioritise}(\mathcal{C}_1[X] \;|||\; RUN_{\Sigma''}, \leq_2, \Sigma).$$

Step 3: We now ensure that an acceptance set inferred from a trace is a complete set accepted by the process under examination. This is most straightforwardly done by employing a regulator process, which can either accept an unprimed event or accept the alphabetically first refusal or acceptance event, followed by a refusal or acceptance for each event in turn. In the latter case it then communicates a *done* event, and returns to its original state. It has thus recorded the complete set of events accepted by P's present state.

The *done* event is necessary in order to distinguish between a terminal $\emptyset$, which can have a *done* after the last event, and a terminal $\bullet$, which cannot (observe that a $\emptyset$ cannot occur other than at the end). Along the way, we hide the refusal events.

Let a and z denote the alphabetically (by which we mean in a fixed but arbitrary linear order on Σ) first and last events respectively, and let $\mathtt{succ}x$ denote the alphabetical successor of x. Define the processes

$$\begin{aligned}
\mathit{UNSTABLE} &= \Box_{x \in \Sigma}\, x \to \mathit{UNSTABLE} \\
&\qquad \Box\; a' \to \mathit{STABLE}(a) \;\Box\; a'' \to \mathit{STABLE}(a) \\
\mathit{STABLE}(x) &= (\mathtt{succ}\, x)' \to \mathit{STABLE}(\mathtt{succ}\, x) \\
&\qquad \Box\; (\mathtt{succ}\, x)'' \to \mathit{STABLE}(\mathtt{succ}\, x) \qquad\qquad (x \neq z) \\
\mathit{STABLE}(z) &= \mathit{done} \to \Box_{x \in \Sigma}\, x \to \mathit{UNSTABLE},
\end{aligned}$$

and let

$$\mathcal{C}_{\mathcal{FL}}[X] = \left(\mathcal{C}_2[X] \underset{\Sigma\cup\Sigma'\cup\Sigma''}{\parallel} \mathit{UNSTABLE}\right) \setminus \Sigma'.$$

Note that $\mathcal{C}_1[P]$ and $\mathcal{C}_2[P]$ have the same number of states as P, and *UNSTABLE* has $|\Sigma| + 1$ states. Hence the state space of $\mathcal{C}_{\mathcal{FL}}[P]$ exceeds that of P by a factor of at most $|\Sigma| + 1$.

Step 4: We now complete the proof by defining the function π inductively as follows:

$$\begin{aligned}\pi(s\hat{}\langle\bullet\rangle) &= \pi(s)\\ \pi(s\hat{}\langle x\rangle) &= \pi(s)\hat{}\langle x\rangle\\ \pi(s\hat{}\langle A = \{x_1,\ldots,x_k\}\rangle) &= \pi(s)\hat{}\langle x_1''\ldots x_k''\mathit{done}\rangle,\end{aligned}$$

where without loss of generality the x_i are listed in alphabetical order.

It is clear that this is order-reflecting, and by the construction above satisfies $\mathcal{T}(\mathcal{C}_{\mathcal{FL}}[P]) = \mathrm{pref}(\pi(\mathcal{FL}(P)))$. ∎

This result allows us to translate questions of $\mathcal{FL}$-refinement into questions of trace refinement under $\mathcal{C}_{\mathcal{FL}}$, as follows:

Corollary 1 *For $\mathcal{C}_{\mathcal{FL}}$ as in Lemma 4, and for any processes P and Q, we have $P \sqsubseteq_{\mathrm{FL}} Q$ if and only if $\mathcal{C}_{\mathcal{FL}}[P] \sqsubseteq_{\mathrm{T}} \mathcal{C}_{\mathcal{FL}}[Q]$.*

Proof
Certainly if $\mathcal{FL}(Q) \subseteq \mathcal{FL}(P)$ then $\mathcal{T}(\mathcal{C}_{\mathcal{FL}}[Q]) = \mathrm{pref}(\pi(\mathcal{FL}(Q))) \subseteq \mathrm{pref}(\pi(\mathcal{FL}(P))) = \mathcal{T}(\mathcal{C}_{\mathcal{FL}}[P])$ and so $\mathcal{C}_{\mathcal{FL}}[P] \sqsubseteq_{\mathrm{T}} \mathcal{C}_{\mathcal{FL}}[Q]$.

Conversely, suppose there exists $x \in \mathcal{FL}(Q) - \mathcal{FL}(P)$. Then since $\mathcal{FL}(P)$ is downwards-closed, we have $x \not\leq y$ for all $y \in \mathcal{FL}(P)$. Since π is order-reflecting, we have correspondingly $\pi(x) \not\leq \pi(y)$ for all $y \in \mathcal{FL}(P)$. Hence $\pi(x) \notin \mathrm{pref}(\pi(\mathcal{FL}(P)))$, so $\mathrm{pref}(\pi(\mathcal{FL}(Q))) \not\subseteq \mathrm{pref}(\pi(\mathcal{FL}(P)))$. ∎

5.2 Model shifting for rational observational models

We now have essentially all we need to prove the main theorem. We formally record a well known fact, that any Nondeterministic Finite Austomaton (NFA) can be implemented as a CSP process (up to prefix-closure, since trace-sets are prefix-closed but regular languages are not):

Lemma 5 (Implementation for NFA) *Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be a (nondeterministic) finite automaton. Then there exists a finite-state CSP process $P_{\mathcal{A}}$ such that $\mathrm{pref}(L(\mathcal{A})) = \mathrm{pref}(\mathcal{T}(P_{\mathcal{A}}))$.*

See Chapter 7 of [19] for the proof.

Theorem 1 (Model shifting for rational models) *For every rational model $\mathcal{M}$, there exists a context $\mathcal{C}_{\mathcal{M}}$ such that for any process P we have $\mathcal{T}(\mathcal{C}_{\mathcal{M}}[P]) = \mathrm{pref}(\psi(\mathcal{M}(P)))$.*

Proof
Let $\mathcal{A}$ be the automaton recognising $(\phi \times \psi)(\mathcal{M})$ (as from Definition 13), and let $P_{\mathcal{A}}$ be the corresponding process from Lemma 5.

We first apply Lemma 4 to produce a process whose traces correspond to the finite linear observations of the original process, prefixed with left: let $\mathcal{C}_{\mathcal{FL}}$ be the context from Lemma 4, and let the context $\mathcal{C}_1$ be defined by

$$\mathcal{C}_1[X] = \mathcal{C}_{\mathcal{FL}}[X][\![^{\text{left}.x}/x]\!].$$

We now compose in parallel with $P_{\mathcal{A}}$, to produde a process whose traces correspond to the $\mathcal{M}$-observations of the original process. Let $\mathcal{C}_2$ be defined by

$$\mathcal{C}_2[X] = \left(\left(\mathcal{C}_1[X] \underset{\{|\text{left}|\}}{\|} P_{\mathcal{A}}\right) \setminus \{|\text{left}|\}\right) [\![^{x}/\text{right}.x]\!].$$

Then the traces of $\mathcal{C}_2[X]$ are precisely the prefixes of the images under ψ of the observations corresponding to X, as required. ∎

By the same argument as for Corollary 1, we have

Corollary 2 *For any rational model $\mathcal{M}$, let $\mathcal{C}_{\mathcal{M}}$ be as in Theorem 1. Then for any processes P and Q, we have $P \sqsubseteq_M Q$ if and only if $\mathcal{C}_{\mathcal{M}}[P] \sqsubseteq_{\mathrm{T}} \mathcal{C}_{\mathcal{M}}[Q]$.*

6 Implementation

We demonstrate the technique by implementing contexts with the property of Corollary 2; source code may be found at [1].

For the sake of efficiency we produce contexts by hand for each model, rather than going via a context for $\mathcal{FL}$ as in the general result Theorem 1 (which in particular would involve introducing many new events only to hide them).

We have illustrated this effect in Section 7 where we have implemented both the Theorem 1 construction and a custom one, and explained the reasons for this phenomenon.

The starting point for building a custom context for the stable failures model is the context `CR`, which is the same as C_1 used in the first stage of constructing $\mathcal{C}_{\mathcal{FL}}$ in the proof of Lemma 4, except that it also has the event *stab* (of lower priority than τ only) which is used to distinguish between an unstable state and a stable one that accepts everything. `CR` is an efficient direct model shifting transformation for the refusal testing model. The context `CF[P]` is `CR[P]` put in parallel with a process that prevents real (namely unprimed) events after either primed events or *stab*: thus refusals and stability are only recorded at the end of a trace. It characterises the stable failures model. The code for all the contexts we introduce can be found in the accompanying example files [1].

The revivals model can also be produced from `CR` by specifying that no more than one unprimed event (itself followed by nothing else) comes after one or more primed events and *stab*.

For models based on acceptance rather than refusal sets, we introduce acceptance events which can occur only when the corresponding refusal events cannot and introduce an appropriate regulator to allow only precise acceptance sets, in a similar manner to the construction in the proof of Theorem 1.

These contexts for the 'refusal sets' models ($\mathcal{F}$, $\mathcal{R}$ and $\mathcal{RT}$) are however suboptimal over large alphabets, in the typical situation where most events are refused most of the time. FDR's inbuilt failures refinement checking codes refusal in terms of *minimal acceptance* sets (checking that each such acceptance of the specification is a superset of one of the implementation). Minimal acceptances are typically smaller than maximal refusal sets, though this is only in typical rather than worst case examples.

If we too wish to work with acceptances then we face the following problem: how to check that the acceptances of the *specification* are a subset of those of the *implementation*, despite the fact that trace refinement checks for inclusion the other way?

The answer is to treat the specification and the implementation slightly differently. We allow the implementation to produce only the precise acceptance sets of *Impl*, but the specification to produce all *supersets* of the acceptance sets of *Spec*. In practice this means that to check (for instance) the failures model, we take the implementation to be $\mathcal{C}_{\mathcal{A}}[\mathit{Impl}]$, whereas for the specification we additionally interleave with $\mathit{RUN}_{\Sigma''}$ (before application of the regulator which prevents acceptances except at the end of traces).

6.1 Testing

We test this implementation by constructing processes which are first distinguished by the stable failures, revivals, refusal testing and acceptance models respectively (the latter two being also distinguished by the finite linear observations model). The processes, and the models which do and do not distinguish them, are shown in Table 1 (recall the precision hierarchy of models: $\mathcal{T} \leq \mathcal{F} \leq \mathcal{R} \leq \{\mathcal{A}, \mathcal{RT}\} \leq \mathcal{FL}$). The correct results are obtained when these checks are run in FDR4 with the implementation described above.

Specification	Implementation	Passes	Fails
$a \to \mathbf{div}$	$a \to \mathit{STOP}$	$\mathcal{T}$	$\mathcal{F}$
$((a \to \mathbf{div}) \,\Box\, \mathbf{div}) \sqcap \mathit{STOP}$	$a \to \mathbf{div}$	$\mathcal{F}$	$\mathcal{R}$
$(a \to \mathbf{div}) \sqcap (\mathbf{div} \triangle (a \to \mathit{STOP}))$	$a \to \mathit{STOP}$	$\mathcal{R}, \mathcal{A}$	$\mathcal{RT}, \mathcal{FL}$
$(a \to \mathit{STOP}) \sqcap (b \to \mathit{STOP})$	$(a \to \mathit{STOP}) \,\Box\, (b \to \mathit{STOP})$	$\mathcal{R}, \mathcal{RT}$	$\mathcal{A}, \mathcal{FL}$

Table 1 Tests distinguishing levels of the model precision hierarchy. $\triangle$ is the *interrupt* operator; see [22] for details.

6.2 Performance

We assess the performance of our simulation by running those examples from Table 1 of [7] which involve refinement checks (as opposed to deadlock- or divergence-freedom assertions), and comparing the timings for our construction against the time taken by FDR4's inbuilt failures refinement check (since $\mathcal{F}$ is the only model for which we have a point of comparison between a direct implementation and the methods developed in this paper). In each of them the parameters have been chosen to give a moderate-scale check. Results are shown in Table 2, for both the original and revised contexts described above; the performance of the $\mathcal{FL}$ check is also shown. As may be seen, performance is somewhat worse but not catastrophically so. Note however that these processes involve rather small alphabets that are exposed to the transformation; performance is expected to be worse for larger alphabets.

	Inbuilt $\mathcal{F}$			CF			CF'			FL		
File	$\|S\|$	$\|\Delta\|$	$T(s)$	$\|S\|$	$\|\Delta\|$	$T(s)$	$\|S\|$	$\|\Delta\|$	$T(s)$	$\|S\|$	$\|\Delta\|$	$T(s)$
inv	21	220	15	21	220	53	21	220	86	21	220	87
nspk	6.8	118	15	7.1	128	55	7.8	141	83	3.6	63	28
swp	24	57	12	30	123	43	43	76	71	42	93	76

Table 2 Experimental results comparing the performance of our construction with FDR4's inbuilt failures refinement check. $|S|$ is number of states in millions, $|\Delta|$ is number of transitions in millions, T is time in seconds. As the nspk checks are negative, there is a good degree of nondeterminism in all the figures on this row because they depend on where in the relevant ply of the search the counterexample is found.

We remarked earlier, in a footnote, that we do not expect model shifting to work as efficiently when checking $P \sqsubseteq_F Q$ where P is a process where for one or more traces there are many maximal refusals for P after s and these refusals have many distinct intersections. The worst possible case of this is the common failures specification of deadlock freedom

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

which, though it has only one normal form state over $\mathcal{F}$, that state has $|\Sigma|$ maximal refusals, all of whose intersections are distinct. Thus with $|\Sigma| = 20$ there are over 1,000,000 normal form states for $\mathcal{C}_1[DF]$ and FDR4 takes 130 seconds to check $\mathcal{C}_1[DF] \sqsubseteq \mathcal{C}_1[DF]$ in contrast to negligible time for FDR4's own failures check.

The $\mathcal{C}_1[P]$ coding optimises the number of states in the model shifting process to double the original, while the approach we used for $\mathcal{FL}$, by forcing the acceptance and refusal reporting to be in strict order, typically multiplies the state space by $|\Sigma| + 1$. On the other hand this coding approach substantially curtails the normalisation blow-up illustrated above, though does not eliminate it. Thus there is a trade-off between different measures of efficiency.

6.3 Debugging output

When a refinement check on FDR4 fails, it gives a syntax-driven multi-level behaviour of the implementation that fails the specification in its GUI, which allows the user to expand or contract syntax and pan through the states and actions that have led to the failure to meet a specification. It naturally gives slightly different forms of output for traces, refusal sets and divergences. In this paper we have shown how to reduce the behaviours in diverse models to "decorated" traces, namely traces of events, some of which are actually annotations of what the underlying process can accept or refuse.

When a model-shifting refinement fails, the user will be able to see the trace and states of the underlying process P by focussing on P in the syntax tree of $\mathcal{C}[P]$, and will also be able to see the acceptance and/or refusal events that contributed to the failure of the specification. The fact that FDR always finds a shortest

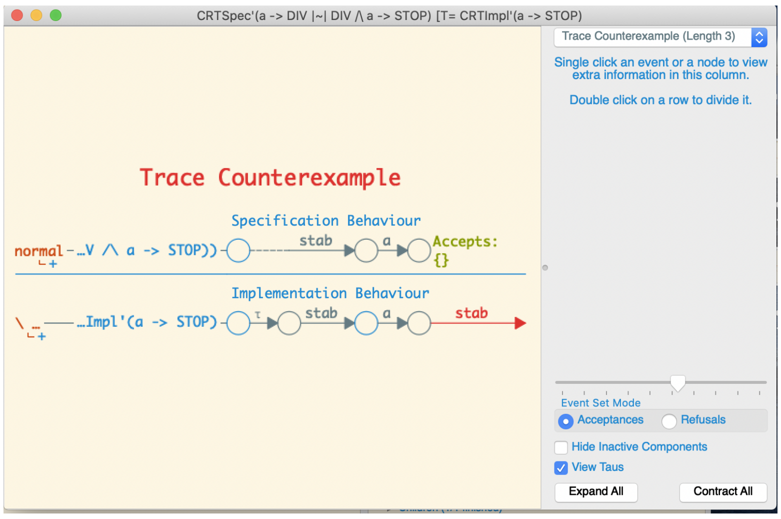
CRTSpec'(a -> DIV |~| DIV /\ a -> STOP) [T= CRTImpl'(a -> STOP)
Trace Counterexample (Length 3)
Single click an event or a node to view extra information in this column.
Double click on a row to divide it.
Trace Counterexample
Specification Behaviour
normal - ...V /\ a -> STOP))
stab
a
Accepts: {}
Implementation Behaviour
\ ... ...Impl'(a -> STOP)
τ
stab
a
stab
Event Set Mode
Acceptances
Refusals
Hide Inactive Components
View Taus
Expand All
Contract All

Fig. 2 FDR4 debugging output for model shifting.

counter example means that events and annotations that are irrelevant to failing the specification do not appear.

We have found the resulting debugging feedback quite easy to understand and navigate for Fig. 2 shows the trace reported by FDR4 for the failure of the model-shifted version of the refusal-testing model refinement check of

$$((a \to STOP) \,\square\, \mathbf{div}) \sqcap (a \to \mathbf{div})$$

by $a \to STOP$. This fails though the corresponding failures and revivals checks succeed: the failure of refinement manifests itself via the right hand process being stable initially, and then performing a before being stable again.

6.4 Example: Conflict detection

We now illustrate the usefulness of richer semantic models than just traces and stable failures by giving a sample application of the revivals model. Consequently we have a property that can be decided using model shifting but not with any model supported directly by FDR4.

Suppose that we have a process P consisting of the parallel composition of two sub-processes Q and R. The stable failures model is able to detect when P can refuse all the events of their shared alphabet, or deadlock in the case when they are synchronised on the whole alphabet. However, it is unable to distinguish between the two possible causes of this: it may be that one of the arguments is able to refuse the entire shared alphabet, or it may be that each accepts some events from the shared alphabet, but the acceptances of Q and R are disjoint. We refer to the latter situation as a ‘conflict’. The absence of conflict (and similar situations) is at the core of a number of useful ways of proving deadlock-freedom for networks of processes running in parallel [26].

The revivals model can be used to detect conflicts. For a process $P = Q\ {}_X\|_Y\ R$, we introduce a fresh event a to represent a generic event from the shared alphabet, and form the process $P' = Q'\ {}_{X'}\|_{Y'}\ R'$, where $Q' = Q[\![\{(x, x), (x, a) : x \in X\}]\!]$, $X' = X \cup \{a\}$, and similarly for R' and Y'. Conflicts of P now correspond to revivals $(s, X \cap Y, a)$, where s is a trace not containing a.

7 Timed failures and Timed CSP

Timed CSP is a notation which adds a $WAIT\ t$ construct to CSP and reinterprets how processes behave in a timed context. So not only does it constrain the order that things happen, but also when they happen. Introduced in [28], it has been widely used and studied [30,29,4]. $WAIT\,t$ behaves like $SKIP$ except that termination takes place exactly t time units after it starts. It introduced and uses the vital principle of *maximal progress*, namely that no action that is not waiting for some other party's agreement is delayed: such actions do not sit waiting while time passes. That principle fundamentally changes the nature of its semantic models.

Consider how the hiding operator is defined. It is perfectly legitimate to have a process P that offers the initial visible events a and b for an indefinite length of time, say $P = a \to P1 \;\square\; b \to P2$. However $P \setminus \{a\}$ cannot perform the initial b at any time other than the very beginning because the a has become a τ: either it or the b must happen the moment the τ is available. So $P \setminus X$ only uses those behaviours of P which refuse X whenever time is passing. This means that timed traces (i.e. traces in which all events have times) do *not* provide a compositional model for Timed CSP.

Timed CSP was originally described on the basis of continuous (non-negative real) time values. The basic unit of semantic discourse is a *timed failure*, the coupling of a timed trace – a sequence of events with non-strictly increasing times – and a timed refusal, which is the union of a suitably finitary set of products of a half-open time interval $[t_1, t_2)$ (containing t_1 but not t_2) and a set of events. Thus the refusal set changes only finitely often in a finite time, coinciding with the fact that a process can only perform finitely many actions in this time. This continuous model of time takes it well outside the finitary world that model checking finds comfortable. However it has long been known that restricting the t in $WAIT\ t$ statements to integers[7] makes it susceptible to a much more finitary analysis by region graphs [9]. However the latter represents a technique remote from the core algorithms of FDR so it has never been implemented for CSP, though it has for other notations [11]. In [16,15], Joel Ouaknine reported the following important discoveries:

- It makes sense to interpret Timed CSP with integer $WAIT$ over the positive integers as the time domain.
- The technique of *digitisation* (effectively a uniform mapping of general times to integers) provides a natural mapping between the continuous and discrete representations.
- Properties that are *closed under inverse digitisation* can be decided over continuous Timed CSP by analysis over Discrete Timed CSP, and these include many practically important specifications.

[7] This does not constrain all events to happen at integer times. Rather it makes all differences between event times that are determined by the process (as opposed to environment) integers.

– It is in principle possible to interpret Timed CSP in a modified (by the addition of two new operators: versions of $\Box$ and $\triangle$ that are not triggered by *tock*) *tock*-CSP (a dialect developed by Roscoe in the early 1990's for reasoning about timed systems in FDR). In a way that is equivalent to its semantics in the integer version of timed failures. Therefore it is possible to reason about continuous Timed CSP in FDR. The definition of Timed CSP hiding over LTSs involves prioritising τ and $\checkmark$ over *tock*.
  Ouaknine's translation of $STOP$ is $TSTOP = tock \to TSTOP$, and that of $a \to P$ is $R = a \to P' \Box tock \to R$, where P' is the translation of P. These are designed to observe that these processes both let time pass through the occurrence of *tock*s.

This was implemented as described in in [2], originally in the context of the last versions of FDR2 and Timed CSP continues to be supported in FDR4. One can write in the Timed CSP notation and FDR performs the translation into *tock*-CSP. There is an important thing missing from these implementations, however, namely refinement checking in the timed failures model, the details of which we describe below. That means that although it is possible to check properties of complete Timed CSP systems, there is no satisfactory compositional theory for (Discrete) Timed CSP that FDR supports directly. For example one cannot automate the reasoning that if $C[P, Q]$ (a term in Timed CSP) satisfies $SPEC$, and $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ then $C[P', Q']$ satisfied $SPEC$, because FDR does not give us a means of checking the necessary refinements.

The purpose of this section is to show how timed failures refinement can be reduced to things FDR can do, filling this hole, while at the same time exploring the possibilities for doing so. Given the methods described in this paper to date, it is natural to try model shifting, and we will do this below, both using the formulaic approach Section 5 creates, and a custom approach. There is another option offered to us by late versions of FDR2, namely reduction to the Refusal Testing model which is implemented in that but not (at the time of writing) later versions of FDR. We will discuss these in turn.

7.1 A summary of discrete timed failures

The *discrete timed failures model* $\mathcal{D}$ consists, in its usual presentation, of sequences of the form

$$(s_0, X_0, tock, s_1, X_1, tock, \ldots, s_{n-1}, X_{n-1}, tock, s_n, X_n)$$

where each of s_i is a member of Σ^*, each of X_i is a subset of Σ, and $tock \notin \Sigma$. Since *tock* never happens from an unstable state, there is no need to have the possibility of $\bullet$ as discussed above for other models before *tock*, and it would be misleading to have it. However $\bullet$ is an option for X_n.

If we are allowing for $\checkmark$, then this can replace X_n in the above form.

What this means, of course, is that the trace s_0 occurs, after which it reaches a stable state refusing X_0 where *tock* occurs, and this is repeated for other s_i and X_i until, after the last *tock*, the trace s_n is performed followed by the refusal x_n (not including *tock*) or potentially instability or $\checkmark$. Recall that we apply the principle of maximal progress, so that *tock* only happens from a stable state: this means that if, after behaviour $\ldots s_n$, stability is not observable, then *tock* can never happen and we have reached an error state. It is, however, convenient to have this type of error or partial state in our model because misconstrued or partially defined systems can behave like this.

Like other CSP models, it has healthiness conditions, or in other words properties that the representation of any real process must satisfy. These are analogous to those of related untimed models, such as prefix closure and subset closure on refusal sets, and the certain refusal of impossible events. A property that it inherits from continuous Timed CSP is *no instantaneous withdrawal*, meaning that if, following behaviour β, it is impossible for a process to refuse a leading up to the next *tock*, then the process must still have the possibility of performing a after $\beta\langle tock\rangle$. This amounts to the statement that the passage of time as represented by *tock* is not directly visible to the processes concerned, and is much discussed in the continuous context in [18,27].

One healthiness condition the above formulation has is that if

$$(s_0, X_0, tock, s_1, X_1, tock, \ldots, s_{n-1}, X_{n-1}, tock, s_n, X_n)$$

belongs to a member then so does

$$(s_0, X_0, tock, s_1, X_1, tock, \ldots, s_{n-1}, X_{n-1}, tock, s_n, X_n, tock, \langle\,\rangle, \bullet)$$

because in any stable state, *tock* is available.

Since the reverse implication holds as well, we realise that it is not in fact necessary to record stability and refusal sets at the end of behaviours. We get exactly the same equivalence on processes and refinement relation by re-defining the model to consist of behaviours of the form

$$(s_0, X_0, tock, s_1, X_1, tock, \ldots, s_{n-1}, X_{n-1}, tock, s_n)$$

with $n \geq 0$ where s_n may end in $\checkmark$.

We will adopt this presentation, since it turns out to fit slightly more easily with model shifting. We will mean this version by $\mathcal{D}$.

It is a rational model, since it can be obtained from the standard representation of $\mathcal{RT}$ by the rational transduction which deletes all refusal sets not preceding *tock* (and replaces occurrences of $\bullet$ before *tock* by $\emptyset$, since *tock* can only occur in stable states). Hence by Theorem 1 it can be model shifted: there exists a context $\mathcal{C}_{\mathcal{D}}$ such that trace refinement under $\mathcal{C}_{\mathcal{D}}$ is equivalent to refinement in $\mathcal{D}$.

We will examine a variety of approaches to expressing $\sqsubseteq_{TF}$ in models accessible to FDR below. We will not consider variants including $\checkmark$ because this makes all the transformations more involved: as in [23] we could extend all the transformations to incorporate this, frequently using versions of the transformation used earlier for $\mathcal{F}$ with $\checkmark$:

$$P = (P\,; term \to SKIP) \setminus \{term\}$$

for a new event *term* which becomes a cipher for termination in the transformations which, unlike $\checkmark$, can be synchronised cleanly with a regulator process.[8]

[8] Thus $RUN = \Box_{a \to \Sigma}\, a \to RUN$ is the unit of parallel composition $\|$ (even with processes involving $\checkmark$), but there is no easy way to involve $\checkmark$ as an event in RUN and maintain that property. However if P has been replaced by $P;term \to SKIP$ we can use $RUN' = \Box_{a \to \Sigma}\, a \to RUN' \,\Box\, term \to SKIP$ and use this as a prototype regulator. Of course if this route is followed then *term* is hidden at the outside of the construction.

The operational semantics of Discrete Timed CSP processes are calculated as an LTS in two stages: first the conventional CSP operational semantics are calculated of Ouaknine's transformation as described and implemented in [2]. Then the timed priority construct which prioritises τ and $\checkmark$ over *tock* is applied. These have the property that *tock* is available in every stable state and no unstable state. It is obvious that no unstable state can have a *tock* because of the application of timed priority. That all stable states of the pre-prioritised semantics have *tock* is a consequence of Ouaknine's translations of $STOP$ and $a \to P$ quoted above and the fact that the operators used do not block *tock*.

7.2 Model shifting timed failures

In model shifting Timed CSP we have FDR turn a process into this language into a standard LTS, one which involves the *tock* action and which has the property discussed above that *tock* is available from precisely the stable states.

In judging refinement between them we can thus use model shifting as rooted in ordinary CSP: it is not done in Timed CSP itself.

We offer two choices for how to do this: we can follow the formulaic recipe for model shifting offered in Section 6, or we can create a customised version that it hoped to be more efficient. These are covered in the next two sections.

7.2.1 Via $\mathcal{FL}$ and an automaton

In many ways the approach set out in our proof of Theorem 1 is like the approach suggested above involving refusal testing: take a representation of refinement in a finer model and then show how to forget, precisely, the details of it not needed for the target. To be precise the method of the theorem is to generate the $\mathcal{FL}$ model-shifting (i.e. decorated trace) representation of a process, and then pipe that into an automaton which converts that representation into one for the chosen model. In other words, the $\mathcal{FL}$ representation becomes the input to the automaton and is hidden, leaving only the automaton's output visible. The automaton, because of the way it is intended to be plumbed in, was termed a transducer.

In this section we demonstrate how this can be done for $\mathcal{D}$. We build a transducer from the decorated traces generated by $\mathcal{C}_{\mathcal{FL}}$ (the model shifting context we defined for for $\mathcal{FL}$) to those of $\mathcal{D}$. There are two challenges to doing this: the first is only to output a refusal set before a *tock*. The second is to convert the exact *acceptances* of $\mathcal{FL}$ into the subset-closed refusal sets of $\mathcal{D}$. We solve the first of these with nondeterminism, and the second by some careful coding, which depends crucially on the fact that $\mathcal{C}_{\mathcal{FL}}$ generates acceptances in a strictly defined order.

The second requires a little more indirection as, having hidden refusal Σ' actions in the context, we need to re-introduce them and not pass on the Σ''s. For this, the automaton records two parameters, typically the two most recent Σ'' (acceptance) events seen, and permits the refusal events strictly between them. This uses the fact that $\mathcal{C}_{\mathcal{FL}}[P]$ generates a full set of acceptance events in increasing order, and requires boundary cases for the beginning and end of the set. It lets $\mathcal{C}_{\mathcal{FL}}[P]$ output an acceptance in this order and remembers the two most recent values output. It then can contribute the refusals of events strictly between them.

In the following the function `rght(x)` renames event `x` into a distinct name for output by the transducer. The programs are in the machine-readable CSP$_M$ notation quoted from our implementation.

The transducer nondeterministically decides whether the next visible event will be `tock` or not. When it is not going to be tock we can ignore decorations and not pass them on. Only regular events other than `tock` are output, and one is it makes the initial choice again.

```
T3 = T3t [] T3nt

T3nt = ([] x:Sigma @ x -> rght(x) -> T3)
        [] ([] x:union(Sigma',Sigma'') @ x -> T3nt)
        [] (tockp -> T3nt [] tockpp -> T3nt)
```

If the next event proper will be `tock` then this is passed on, and acceptance decorations are translated into a nondeterministic collection of refusal ones using the approach discussed above. We have to allow for cases at the beginning and end of the acceptance as well as the one where there are two most recent outputs.

```
T3t = tock -> rght(tock) -> T3
      [] ([] x:{y | y <- Sigma} @ acc(x) -> T3t1(x) )
      [] (done -> T3t2)
      [] (tockp -> T3t [] tockpp -> T3t)

T3t1(u) = tock -> rght(tock) -> T3
      [] ([] x:{y | y <- Sigma, ord(y)>ord(u)} @
                                acc(x) -> T3t3(u,x))
      [] (done -> T3t4(u))
      [] ([] x:{y | y <- Sigma, ord(y)<ord(u)} @
                                rght(ref(x)) -> T3t1(u))
      [] (tockp -> T3t1(u) [] tockpp -> T3t1(u))

T3t2 = tock -> rght(tock) -> T3
      [] ([] x:{y | y <- Sigma} @ rght(ref(x)) -> T3t2)
      [] (tockp -> T3t2 [] tockpp -> T3t2)

T3t3(l,u) = tock -> rght(tock) -> T3
      [] ([] x:{y | y <- Sigma, ord(y)>ord(u)} @ i
                                acc(x) -> T3t3(u,x))
      [] (done -> T3t4(u))
      [] ([] x:{y | y <- Sigma, ord(y)<ord(u), ord(y)>ord(l)} @
                                rght(ref(x)) -> T3t3(l,u))
      [] (tockp -> T3t3(l,u) [] tockpp -> T3t3(l,u))

T3t4(l) = tock -> rght(tock) -> T3
      [] ([] x:{y | y <- Sigma, ord(l)<ord(y)} @
                               rght(ref(x)) -> T3t4(l))
```

We will report on the performance of the resulting model shift in Section 8.1.

The accompanying files, in addition to the above, contain a transducer that creates a representation of $\mathcal{D}$ directly from the context $\mathcal{C}_2$ as an alternative to $\mathcal{C}_{\mathcal{FL}}$.

7.2.2 Custom approach

Here we avoid the passage through $\mathcal{FL}$ and go directly to the structure of $\mathcal{D}$. The approach here differs first by not using acceptance events at all and, instead of using a transducer, restricts the natural model shifting representation of the refusal testing model.

As before we introduce a primed copy a' of each $a \in \Sigma$ to represent refusals, and using the following construct involving a regulator which ensures that an ordinary event cannot follow a refusal flag. This means that its traces consist of pairs of traces of Σ and traces of Σ' (which can be empty) interspersed with a *tock* between consecutive pairs. We do not need *stab* because in a $\mathcal{D}$ behaviour there is no ambiguity about where in a recorded behaviour the underlying process is known to be stable: immediately before the *tock* actions.

$$CS_{TF}(P) = \mathbf{prioritise}(\leq, P \mid\mid\mid RUN(\Sigma'), \Sigma) \parallel Reg$$

$$\begin{aligned}
Reg &= tock \to Reg \\
&\quad \Box\, (\Box_{a \in \Sigma}\, a \to Reg) \\
&\quad \Box\, (\Box_{a \in \Sigma}\, a' \to Reg1) \\
Reg1 &= tock \to Reg \\
&\quad \Box\, (\Box_{a \in \Sigma}\, a' \to Reg1)
\end{aligned}$$

and $a' < a$, (as well as the implicit $a' < \tau$) for each $a \in \Sigma$.

We have assumed here, as before, that any prioritisation needed to ensure maximal progress has already been applied before this, so that the LTS being operated on here has the correct behaviour under a normal interpretation of how LTSs behave.

The correctness of this construction is argued as follows. It comes in two stages:

$$CS^1_{TF}(P) = \mathbf{prioritise}(\leq, P \mid\mid\mid RUN(\Sigma'), \Sigma)$$

$$CS^2_{TF}(Q) = Q \parallel Reg$$

The first of these is almost identical to the natural model-shifting transformation for the refusal testing model (for any alphabet). The difference is the exclusion of *tock*$'$ and *stab*. It is clear that it delivers a set of traces including the refusal indicators a' for $a \in \Sigma$ which includes representations of every behaviour of P of the form

$$\{(X_0, b_1, \ldots, X_{n-1}, b_n, X_n) \mid b_i \in \Sigma \cup \{tock\}, X_i \subseteq \Sigma, b_i \notin X_i\}$$

where P's operational semantics has a sequence of states linked by actions, such that $\langle b_1, \ldots, b_n \rangle$ are the visible actions (i.e. the ones that are not τ), and when X_{i-1} is non-empty then b_i occurs from a stable state refusing X_{i-1}.

Furthermore if X_n is non-empty then the final state of the behaviour is stable and refuses it.

There is no distinction between $\emptyset$ and $\bullet$ because this is not necessary for $\mathcal{D}$.

Because there is no event *tock*$'$ used here it ignores the possibility of the process refusing *tock*, which we actually know is impossible for P a Discrete Timed CSP process.

The second stage imposes a regulator which restricts which decorated traces appear and thus which of the refusal testing behaviours are seen.

Note that this regulator allows only allows refusal events and *tock* after refusal events a', thus forcing the decorated traces (namely combinations of real events and the a' ones signifying refusals) to exactly follow the structure set out for timed failures above except that there are now final refusals as well as ones before *tock*. However we can identify such a trace with a non-empty sequence of final Σ' actions the $\mathcal{D}$ behaviour with an additional $tock, \langle\,\rangle$ at the end.

So when put in parallel with $CS^1_{TF}(P)$ the regulator only retains, with this last interpretation, the decorated traces that are consistent with the structures for $\mathcal{D}$. The value in that model of process P are simply derived point-wise from the remaining decorated traces of $CS_{TF}(P)$.

Furthermore, for any behaviour of this form, any decorated trace which this mapping would send to the above behaviour (there are many because the a' events from a given X_i can be re-ordered or repeated, bearing in mind that Σ' events do not change $CS^1_{TF}(P)$'s state.)

It follows that $CS_{TF}(P) \sqsubseteq_T CS_{TF}(P)$ if and only if $P \sqsubseteq_{TF} Q$.

7.3 Reducing timed failures to refusal testing

The constructions of the previous section show how to build a model shifting representation of a process P in the refusal testing model and then project that to a model shifting representation of P in $\mathcal{D}$.

In this section we show how to do the same thing in the space of regular CSP, without the extra events used to decorate traces but where we do have to worry about refusal sets instead.

So we are creating an analogue of $C^2_{TF}[P]$ that works in the space of CSP with refusal sets rather than decorated traces and the traces model.

We want $C[\cdot]$ such that $C[P]$ contains precisely the refusal testing behaviours of P that are required for $\mathcal{D}$, so that $C[P] \sqsubseteq_{RT} C[Q]$ if and only if $P \sqsubseteq_{TF} Q$.

This turns out to be somewhat more subtle than the creation of $C^2_{TF}[\cdot]$. In that case we could ensure that refusals only happen before *tock* events or at the end of a trace by preventing events from Σ appearing in traces after $\Sigma' \cup \{stab\}$. However this cannot be done so easily in $C[P]$. The states where *tock*s occur need to have the correct refusal set, but if an event other than *tock* occurs cannot continue after it.

This apparent paradox can be resolved by making $C[P]$ the parallel composition of P and a regulator which after any trace has two choices:

- it can perform any event in Σ from an unstable state and carry on,
- it can offer the combination $\Sigma \cup \{tock\}$, but if an event other than *tock* occurs it goes to state **div** which is the refinement-maximum member of the refusal testing and timed failures models.

These choices can be given via nondeterministic choice, based on the fact that in any finite observation model $\mathbf{div} \sqcap P = P$ as **div** has no non-trivial finite observations, so

$$((a \to P) \,\square\, \mathbf{div}) \sqcap (a \to \mathbf{div})$$

can refuse $\Sigma - \{a\}$ on the empty trace or perform a and behave like P, *but not* on the same observation. Or they can be defined by sliding choice $\rhd$, where $(a \to P) \rhd a \to \mathbf{div}$ has the same behaviour. This regulator process contains – in $\mathcal{D}$ – precisely the representation in $\mathcal{D}$ of the process

$$RUN = \Box_{a\in\Sigma}\, a \to RUN$$

Using the second of these ideas, the regulator (expressed as a regular as opposed to Timed CSP process) is:

$$\begin{aligned} REGP = {} & (\Box_{a\in\Sigma}\, a \to REGP) \\ & \rhd \\ & (tock \to REGP \\ & \Box\, \Box_{a\in\Sigma}\, a \to \mathbf{div}) \end{aligned}$$

As before, this regulator is synchronised with P to perform the transformation. Since $P \parallel Q$ is only stable – and so has refusals – when both P and Q are, the following parallel composition contains (over the refusal testing model) exactly the timed failures representation of P. We think of it as a projection.

$$\Pi_{TF}(P) = P \parallel REGP$$

Thus we can test $P \sqsubseteq_{TF} Q$ for Timed CSP processes P and Q by taking their operational semantics as discussed earlier and testing

$$\Pi_{TF}(P) \sqsubseteq_{RT} \Pi_{TF}(Q)$$

It is interesting to note that, though the approach given here does not use model shifting in the sense we have introduced in this paper, the proof of Theorem 1 also takes the approach of reducing a representation of a finer model to the chosen one.

8 Case study: Timed Sliding Window Protocol

The sliding window protocol has long been used as a case study with FDR: it is well known and reasonably easy to understand, at least in an untimed setting. It is a development of the alternating bit protocol in which the messages in a fixed-length window on the input stream are simultaneously available for transmission and acknowledgement across an erroneous medium which, in our version, can lose and duplicate messages but not re-order them. At any one time the sender and receiver processes each hold an equal length window on the stream passing through the protocol, in the sense that the places in that window represent (possibly unknown) values taken from consecutive places on the stream. They are not always the same window but (i) the union of them is a window of length up to twice the length of the individual ones, so there is no gap between them even though they may not overlap. Furthermore (ii) the receiver window is always either the same as, or ahead of, the sender one. If it is properly ahead then the last places in the receiver window are necessarily unknown. Messages and acknowledgements are tagged with an indication of what position on the stream of messages they refer to. We have re-interpreted this protocol in Timed CSP with the following features:

- There is a parameter W which defines the width of the window. Because the windows held by the sender and receiver processes may be out of step, we need to define $B = 2W$ to be the bound on the amount of buffering the system can provide. (Trying a smaller value in one of our FDR models will show this is needed.)
- In common with other CSP codings of this protocol, we need to make the indexing space of places in the input and output streams finite by replacing the natural non-negative integers by integers modulo some N which must be at least $2W$ (though there is no requirement that B and N are the same). This is sufficient to ensure that acknowledgement tags never get confused as referring to the wrong message.
- Round robin sending of message components from unacknowledged items in the current window: this clearly has a bearing on the timing behaviour of the transmission and acknowledgements that the system exhibits.
- The occurrence of errors is limited by a parameter which forces them to be spaced: at least K time units must pass between consecutive ones. To achieve this elegantly we have used the controlled error model [19] in which errors are triggered by events that can be restricted by external regulators, and then lazily abstracted. It turns out that lazy abstraction (originally proposed in [19]) needs reformulating in Timed CSP. We will detail this below since it is an important aspect of the Timed CSP modelling of systems with error-prone components. Clearly it would be possible to use different error assumptions.
- We have assumed for simplicity that all ordinary actions take one time unit to complete.
- Where a message is duplicated, we need to assume that the duplicate is available reasonably quickly, say within 2 time units of the original send. If it can be deferred indefinitely this causes subtle errors in the sense that deferred duplication can prevent the system from settling sufficiently.

The full Timed CSP of this implementation can be found in the files which accompany this paper. The core, in CSP$_M$, is (without many details)

```
Timed(AllZero){  -- denotes Timed CSP section
SEND(sw,n,last) =
   (head(sw)==Null&
   left?x -> SEND(tail(sw)^<x>,(n+1)%N,last))
[] d?v -> SEND(ack(sw,n,v),n,last)
[](nonempty(sw,n)&
   let (r,v) = next(sw,n,last) within
   a.r.v -> SEND(sw,n,r))

RCV((rw,n)) = if (head(rw)!=Null) then
        right!head(rw) -> RCV((tail(rw)^<Null>,(n+1)%N))
        else
        b?t?v -> c!t ->  RCV((update((t+N-n)%N,v,rw),n))

SND = SEND(blanks,N-1,N-1)

RCVA = RCV((blanks,0))
SYSTEM = (SND[|{|a,d|}|]((MM|||AM)\{timeout}
            [|{|b,c|}|]RCVA))\ {|a,b,c,d|}
}

PSYSTEM = tpri(SYSTEM)  -- apply Timed priority
```

Here `Timed(T)` declares a Timed CSP part of a CSP script where `T` (for example `AllZero`) maps all event to the non-negative integer time they take to complete. `MM` and `AM` are controlled error models of aspects of communication that communicate messages and acknowledgements respectively between `SND` and `RCVA` but are subject to loss and duplication of their contents.

We can create a timed failures specification in CSP which says, following established models for regular CSP, that the resulting system is a buffer bounded by B (so it never contains more than B items) but is only obliged to input when it has nothing in it. Whenever it is nonempty it is obliged to output, but these two obligations do not kick in before some parameter D time units from the previous external communication.

This is slightly trickier than we might think because of the way in which the implementation process can, entirely legitimately, change its behaviour over time. So in an interval where it can legitimately accept or refuse an input $left.1$, at one point it can refuse to communicate it, while later accepting it after time has passed.

In hand-coded *tock*-CSP this can be expressed as

```
TFBUFF(n) =
let
TFB(s,k) =
 if k < n then
   ((#s>0 & right!head(s) -> TFB(tail(s),0)
            []
     #s<B & left?x -> TFB(s^<x>,0))
    [>
    tock  -> TFB(s,k+1))
  else
    ((#s>0 & right!head(s) -> TFB(tail(s),0))
    []
    (#s==0 & left?x -> TFB(<x>,0))
    []
    ((#s>0 and #s<B) & (left?x -> TFB(s^<x>,0) [> STOP))
    [] tock -> TFB(s,k))
within TFB(<>,0)
```

This says that if we have not yet reached the point where offers must be made (i.e. `k < n`) then it can perform permitted actions but can (expressed via `[>` or sliding choice) also refuse them and wait for time to pass.

For comparison we now present the same specification in the Timed CSP language, partly because we want to investigate that as a practical specification language now we can do compositional verification over it.

In Timed CSP, a completely equivalent specification can be divided into three separate parts: one to control the buffer behaviour, one to handle what the specification says about when offers must be made as opposed to can be made, and the final one to control nondeterminism by creating the most nondeterministic timed process on a given alphabet. The last of these is notably trickier than in the untimed world because where a process has the choice, over a period, to accept or refuse an event b, it is not sufficient for it to make the choice once and for all. So we have

```
TCHAOS(A) = let onestep = ([] x:A @ x -> onestep)
                          [> WAIT(1)
            within onestep;TCHAOS(A)
```

(This is equivalent to the following but maps better into FDR4.)

```
TCHAOS'(A) = [] x:A @ x -> TCHAOS'(A)
               [> (WAIT(1); TCHAOS'(A))
```

Lazy abstraction in CSP, first defined in its present form in [19], is a construct that assumes we have a process P with a partition $\{A, B\}$ of its alphabet. The lazy abstraction of sub-alphabet A creates the view of what P looks like to a user who interacts with it only in B. It is distinct from $P \setminus A$ because it is assumed there is a user controlling A who has the option to refuse as well as accept communications, while in hiding A events are always enabled. In untimed CSP it is implemented as

$$LAbs(A)(P) = (P \parallel CHAOS(A)) \setminus A$$

In Timed CSP lazy abstraction needs to be formulated with this revised $CHAOS$ definition because the user controlling A is allowed to change their mind as time progresses.

```
LAbs(A)(P) = (P [|A|] TCHAOS(A))\A
```

Note that the passage of time (*tock*) is implicitly synchronised here as well as A, and priority of τ over *tock* will also apply.

In the main part of the buffer specification we do not create this style of nondeterminism, but instead use two variants of the externally visible events: one that will be made nondeterministic by the above and one that will not:

```
TFB(s) =
   (#s>0 & right!head(s) -> TFB(tail(s)))
    []
   (#s>0 & rightnd!head(s) -> TFB(tail(s)))
    []
   (#s==0 & left?x -> TFB(<x>))
    []
   (#s<B & (leftnd?x -> TFB(s^<x>)))
```

The above always allows the nondeterministic variants of the events, and allows the “deterministic” ones when they should be offered if sufficient time has passed since the last visible event. Thus `left` is only offered deterministically when the buffer is empty, no matter how long since the last event.

The choice over whether the offers available must be made, implemented by allowing the deterministic versions of events, is made by the following process

```
TEnable(E,R,m) =
let Rest = diff(R,E)
    En = [] x:R @ x -> Dis(m)
 Dis(k) = if k==0 then En else
         (([] x:Rest @ x -> Dis(m))
         [] WAIT(1);Dis(k-1))
within Dis(m)
```

The three parameters here are the events that are enabled when there has been sufficient delay (here `{|left,right|}`), the ones that reset the clock (here `{|left,right,leftng,rightnd|}`) and the time by which offers have to be made. The full specification is put together by combining the above process, `TFB(<>)` and `TCHAOS({|leftnd,rightnd|})` and renaming `leftnd, rightnd` to respectively `left, right`.

Given the subtlety of the above and the fact that it is hard to be sure that `TBUFF` is right when it is written in *tock* CSP rather than Timed CSP, it is reassuring that FDR readily proves that the two versions of the specification are equivalent in the timed failures model.

8.1 Experiments

The authors have run a number of checks of versions of the Timed CSP version of the protocol against this and other specifications, using the approaches to coding timed failures refinement set out in the previous section.

Of these the custom approach of Section 7.2.2 is much the most efficient[9]. Specifically the method using going by the $\mathcal{C}_{\mathcal{FL}}$ shift of $\mathcal{FL}$ took 538 seconds, where the custom method took 32 seconds: the check this related to had specification `TFBUFF(20)` and whose implementation was limited to one transmission error (*loss* or *dup*) every 20 time units. As suggested by data independence [19], the data space had size 2 as this proves that the same check will succeed for any data type of messages.

This ratio is typical. Probably the biggest cause of the difference is the buffering present in the $\mathcal{FL}$ representation piping into a separate automaton in the Theorem 1 based versions. In the rest of this section the model shifting checks referred to all use the custom method.

When compared against FDR's inbuilt stable failures refinement (a less discerning one than timed failures, so not inevitably producing the same results) the overheads were low, typically about 50% in states and time. This suggests to us that, used skillfully, model shifting is a much cheaper method for testing exotic refinements than creating a modified FDR4, which at least some of the time is hardly any worse.

Files illustrating this section can be downloaded[10] alongside this paper, along with some dumps of the data that FDR4 generates on performing the checks.

The following reports on the check of the Timed CSP sliding window protocol with two items of *DATA* and a window of width 4 against the specification that

[9] FDR3 and FDR4, which do not have native refusal testing refinement, are significantly faster at refinement in general than FDR2, which does have this. Therefore the method of Section 7.3 was not compared like-for-like.

[10] `http://www.cs.ox.ac.uk/people/publications/personal/Bill.Roscoe.html`

says it is an 8-bounded buffer when there is a minimum time between errors of 3. It is specified to make stable offers by 42 time units. (In general the longer between errors and lower the width W of the window, the faster the system makes settled offers.)

The first check does this by model shifting, but it fails when the check is nearly complete because it can fail to have the offer ready on time. In fact the corresponding check is passed when 42 is replaced by 45 (but not 44).

```
assert CTFMS(TFBUFF(42)) [T= CTFMS(TLAbs({loss,dup})(ELSYSTEM(3)))
```

The statistics from this check were as follows:

Visited 49,239,989 states and 166,698,488 transitions in 118.91 seconds (on ply 261)

The following is a failures check of the same system without model shifting, which happens to find the same problem.

```
assert TFBUFF(42) [F= TLAbs({loss,dup})(ELSYSTEM(3))
```

Visited 41,779,778 states and 107,648,549 transitions in 81.64 seconds (on ply 261)

It is noteworthy that the overhead of model shifting (relatively speaking) is here less than reported earlier for the untimed case. We expect this is explained because the un-shifted checks in the timed case already contain (timed) prioritisation before it is applied as part of model shifting.

It is straightforward to run experiments that reveal the exact performance of versions of the sliding window protocol with different window lengths and assumed maximal error rates. For example for the window W being 4, the weakest $TFBUFF(n)$ specifications satisfied are as follows: with no errors (*loss* or *dup*) $n = 19$, with one error per m time units: (m, n) can be $(20, 21)$,$(10, 26)$, $(6, 31)$, $(5, 32)$, $(4, 36)$, $(3, 45)$. With an error per 2 time units it is possible that the system will never make its required offers, because the errors happen fast enough to defer the offers for ever. In other words, with an error rate this high, the sliding window protocol we have implemented fails to be a *timewise refinement* as discussed in [24, 30] of the untimed buffer specification: this simply means that the timed traces are buffer traces with the times removed, and that any offer that a buffer is obliged to make is offered infinitely often after some sufficient time after the last visible action of a trace.

In [24], it is shown how to decide timewise refinement using a way that non-compliance with a required offer is turned into a CSP divergence that FDR can detect. These techniques, which turn refusals into events, are forerunners of the model shifting techniques we introduce in this paper. Therefore it will come as little surprise that they can be reformulated in terms of the methods we have already presented. The reader familiar with our notation will realise that a process that can, after trace s refuse the event a for ever will have the infinite trace $s\langle a', tock\rangle^\infty$ under the model shifting of the timed failures model to the traces model. For the buffer specification this is implemented via

```
TFBUFFd =
let
TFB(s) =
   ((#s>0 & right!head(s) -> TFB(tail(s))
               []
```

```
    #s<B & left.0 -> TFB(s^<0>))
    []
    tock  -> TFB(s))
    [] switch -> TFB'(s)
TFB'(s) =
   (#s>0 & rightp!head(s) -> tock' -> TFB'(s)
    [] #s==0 & leftp?x -> tock' -> TFB'(s))
within TFB(<>)
```

running in parallel with the implemented error-prone system with `tock` double renamed[11] to itself and `tock'`, with `{|leftp,rightp,tock'|}` hidden. The occurrence of the new event `switch` in this process stops it monitoring the trace of the process and starts it inviting the infinite trace which demonstrates the indefinite refusal of events the buffer specification says it must eventually offer.

If we modify the above testing process so that pairs of the form `<leftp.x,tock'>` or `<rightp.x,tock'>` can happen, hidden at any time and not just after `switch` (which is now removed), this represents a perhaps more subtle but in fact much more efficient check for infinite refusal.

In both cases the events `{|leftp,rightp,tock'|}` are hidden in the combination and infinite refusal is a divergence that FDR can check for.

Use of these specification shows that the boundary between our timed sliding window protocol timewise refining the untimed buffer specification or not lies between an error every two units and every three: only the later makes the property hold for each window size 2 to 8, the ones we tried it for. (For this `DATA` was size 1 since the said check's result is, by data independence, independent of the size of this type.)

The experiments in this paper were performed using FDR4 on a MacBook with a 2.7GHz Intel Core i7 processor and 16GB of RAM.

9 Conclusions

While this paper can reasonably be seen as very CSP focused, the basic structure of the mathematical arguments and constructions can extend well beyond this, potentially to other parts of the van Glabbeek hierarchy and beyond. Suppose one has a language $\mathcal{L}$ and a hierarchy of semantic models or equivalences $\mathcal{E}$ that are congruences. Suppose also that one has a formal analysis tool which decides properties about equivalence $\mathcal{E}_0$ which is coarser than the other $\mathcal{E}$. Then it may be possible to find an additional operator $\oplus$ (perhaps with parameters) to add to $\mathcal{L}$ which turns the $\mathcal{E}$ except perhaps for some finest one $\mathcal{E}_\omega$ into non-congruences. In our case $\mathcal{E}_0$ and $\mathcal{E}_\omega$ are $\mathcal{T}$ and $\mathcal{FL}$, and $\oplus$ is priority.

Then, given $\mathcal{E}$ other than $\mathcal{E}_0$ it may be possible to apply contexts involving $\oplus$ to processes P so that their semantics in $\mathcal{E}$ becomes apparent in $\mathcal{E}_0$, thus making $\mathcal{E}$ amenable to automatic analysis.

For all this to work it is *essential* that if $\mathcal{E}_1$ and $\mathcal{E}_2$ are two of the equivalences with $\mathcal{E}_0$, $\mathcal{E}_1$ and $\mathcal{E}_2$ successively more discerning (finer) then $\oplus$ cannot be compositional for $\mathcal{E}_1$. If it were, and P, Q are a pair of processes differentiated by $\mathcal{E}_2$ but

[11] This allows `tock` events the system performs before `switch` to remain visible, and those after `leftp.x` or `rightp.x` to be hidden.

not by $\mathcal{E}_1$ then no context involving $\mathcal{L}\{\oplus\}$ can differentiate P and Q in $\mathcal{E}_0$ either $P \equiv_1 Q$ so $C[P] \equiv_1 C[Q]$ to $C[P] \equiv_0 C[Q]$ as $\equiv_0$ is coarser than $\equiv_1$.

Though this may seem complex, we have seen the approach works well for one hierarchy of models, and hope it will work elsewhere too.

Focussing now on CSP, We have seen how its expressive power when extended by priority, allows seemingly any finite behaviour model of CSP to be reduced to traces. Indeed this extends to any finitely expressed rules for what can be observed within finite linear behaviours, whether the resulting equivalence is compositional or not.

This considerably extends the range of what can be done with a tool like FDR. The final section shows an alternative approach to this, namely reducing a less discerning model to a more discerning one without priority. This worked well for reducing timed failures to refusal testing, but other reductions (for example ones involving both acceptances and refusal sets) do not always seem to be so efficient. For example reducing a refusal sets process to the acceptances model seems unnecessarily complex as, for example, the process $CHAOS$ needs exponentially many acceptance sets where a single maximal refusal suffices.

We discovered that it is entirely practical to use this technique to reason about large systems. Furthermore the authors have found that the debugging feedback that FDR gives to model shifting checks is very understandable and usable.

In particular the authors were pleased to find that the results of this paper make automated compositional reasoning about Timed CSP practical. They have already found it most informative about the expressive power of the notation. It seems possible that, as with untimed CSP, the availability of automated refinement checking will bring about enrichments in the notations of Timed CSP that help it in expressing practical systems and specifications.

Model shifting means that it is far easier to experiment with automated verification in a variety of semantic models, so it will only very occasionally be necessary for a new one to be directly supported.

We believe that similar considerations will apply to classes of models that include infinite observations such as divergences, infinite traces, where these can be extended to incorporate refusals and acceptances as part of such observations. In such cases we imagine that model shifting will take care of the aspects of infinite behaviours that are present in their finite prefixes, and that the ways that infinitary aspects are handled will follow one of the three traces models available in CSP. These are

– finite traces (used in the present paper),
– divergence-strict finite and infinite traces, so as soon as an observation is made that can be followed by immediate divergence, we deem all continuations to be in the process model whether or not the process itself can do them operationally, and finally
– with full divergence strictness replaced by the weak divergence strictness discussed in [20] (here an infinite behaviour with infinitely many divergent prefixes is added as above).

Only the first of these is directly supported by FDR4 at present, though the second is easily derived from its support of the failures divergence model. There is no technical obstacle to supporting the third for finite state processes.

Thus it should be possible to handle virtually the entire hierarchy of models described in [22] in terms of variants on traces and model shifting. This will be the subject of future research.

Acknowledgements

The authors are grateful to Tom Gibson-Robinson for helpful discussions and practical assistance with FDR4. The paper has been greatly improved thanks to comments from anonymous referees. This work was been partially sponsored by DARPA under agreement number FA8750-12-2-0247 and by research grant EP/N022777/1 from EPSRC.

References

1. `www.cs.ox.ac.uk/people/bill.roscoe/model-shifting.csp`.
2. Philip Armstrong, Gavin Lowe, Joël Ouaknine, and A.W. Roscoe. Model checking Timed CSP. *In Proceedings of HOWARD (Festschrift for Howard Barringer)*, 2012.
3. Ed Brinksma, Arend Rensink, and Walter Vogler. Fair testing. In *International Conference on Concurrency theory*, pages 313–327. Springer, 1995.
4. Jim Davies and Steve Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.
5. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. 8413:187–201, 2014.
6. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 18(2):149–167, 2016.
7. Thomas Gibson-Robinson, Henri Hansen, A.W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NASA Formal Methods*, pages 188–203. Springer, 2015.
8. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
9. David M Jackson. Logical verification of reactive software systems. 1992.
10. Paris C. Kanellakis, and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation* 86(1): 43-68, 1990.
11. Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.
12. David Mestel and A.W. Roscoe. Reducing complex CSP models to traces via priority. *Electronic Notes in Theoretical Computer Science*, 325:237–252, 2016.
13. R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
14. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.
15. Joël Ouaknine. *Discrete analysis of continuous behaviour in real-time concurrent systems.* PhD thesis, Oxford University, 2000.
16. Joël Ouaknine. Digitisation and full abstraction for dense-time model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 37–51. Springer, 2002.
17. Iain Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241–284, 1987.
18. George M. Reed and A.W. Roscoe. The timed failuresstability model for CSP. *Theoretical Computer Science*, 211(1-2):85–127, 1999.
19. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
20. A.W. Roscoe. Seeing beyond divergence. In *Communicating Sequential Processes. The First 25 Years*, pages 15–35. Springer, 2005.
21. A.W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *The Journal of Logic and Algebraic Programming*, 78(3):163–190, 2009.
22. A.W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010.

23. A.W. Roscoe. On the expressiveness of CSP. 2011.
24. A.W. Roscoe. The automated verification of timewise refinement. *EIT- CPSE*, 2013.
25. A.W. Roscoe. The expressiveness of CSP with priority. *Electronic Notes in Theoretical Computer Science*, 319:387–401, 2015.
26. A.W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289 – 327, 1987.
27. A.W. Roscoe and Jian Huang. Checking noninterference in Timed CSP. *Formal Aspects of Computing*, 25(1):3–35, 2013.
28. A.W. Roscoe and G.M. Reed. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58, 1988.
29. Steve Schneider. An operational semantics for Timed CSP. *Information and computation*, 116(2):193–213, 1995.
30. Steve Schneider. *Concurrent and real-time systems.* John Wiley and Sons, 2000.
31. J. Shallit. *A Second Course in Formal Languages and Automata Theory.* Cambridge University Press, 2009.
32. Rob J van Glabbeek. The linear timebranching time spectrum II. In *International Conference on Concurrency Theory*, pages 66–81. 1993.
33. Rob J Van Glabbeek. The linear time-branching time spectrum I. the semantics of concrete, sequential processes. In *Handbook of process algebra*, pages 3–99. Elsevier, 2001.