The automated verification of timewise refinement (Draft)

A.W. Roscoe,

Oxford University Department of Computer Science

Abstract. While Hoare's CSP models reactive systems without assigning an exact time to events, Timed CSP records the exact times as non-negative reals. Timed CSP therefore provides a more exact semantics of systems, but it still makes sense to ask whether a timed process satisfies an untimed specification. Indeed the question of whether such specifications are satisfied often reduces to the timing details of the implementation. Schneider showed how this could be understood at an abstract level via the concept of timewise refinement. The recent implementation of Timed CSP in the CSP refinement checker FDR (using Ouaknine's theory of digitisation) has at last provided the framework for automating timewise refinement. In this paper we show how to do this, discovering that it is subtle because of the need to reconcile infinite behaviours with finite ones

1 Introduction

Hoare's process algebra CSP [5, 11, 12] provides a framework for describing systems patterns of actions representing communications. It provides operators that allows the description of implementations, typically consisting of parallel networks whose parts communicate by handshaking on their own action. It also provides the means – including representations of nondeterminism, deadlock, and other pathologies – for creating specifications.

The semantics of CSP in its original form cared about the order in which actions happen, but not their exact times. The author and Mike Reed [8] introduced an alternative view of the same notation, with the addition of the timed constant process $WAIT\ t$ that terminates successfully after t time units.

Substantial theories have been developed for both the untimed and timed versions of CSP including abstract models and operational semantics. In the timed models one records more details than in the untimed ones, and so one would expect to have mappings which forget this extra detail and therefore cast any system modelled in timed theories to values in corresponding untimed ones.

Timed systems may be expected to meet specifications which restrict the times at which events happen, as well as ones that do not. In the latter case the correctness of the system may well depend on timing details within a system even though the specification is untimed.[1] For such cases we would need the

[1] A good example of this is provided by the level crossing described in [3, 11], where the untimed specification that the gate is down when a train passes by it is only

detail of Timed CSP to create a model that establishes the specification, and then to have a way of testing it against an untimed specification.

For untimed trace, or safety, specifications, there can be no doubt what this means since there can be no doubt how to extract an untimed trace from a timed behaviour: simply extract the sequence of events that occur in it.

Beyond traces, most used untimed models for CSP are *failures* and *failures-divergences*. Each of these depends, in the usual operational formulation, on the dual ideas of divergence (an unbroken infinite series of internal τ actions and *stability* (reaching a state where no τ action is possible). While it is possible to build timed models that capture this information such as the timed failures-stability model [9], it is far from clear that this provides the most useful link between Timed CSP the untimed failures model.

One argument for this is that divergence is a much less dangerous and difficult phenomenon in the timed world than it is in the untimed one. On the "no-Zeno" assumption that only finitely many events happen in a finite time, divergence can only occur over an infinite interval, during which we can see the offers that the process makes in a way that makes no sense over untimed models. Whereas refusal sets only appear in the untimed models when they become permanent through stability, in the timed models there is no choice but to record what is refused at every instant – event when a τ will later become enabled from the current state.

Schneider [15, 16] therefore described the *timewise refinement* relation between the failures model of untimed CSP and the timed failures model, in which events are given times from the non-negative real numbers, and we record what was (observed to have been) refused at each moment of time. $P_{SF} \sqsubseteq_{TF} Q$[2] if all of the untimed traces that can be extracted from Q are permissible for P, and furthermore whenever Q can perform the untimed trace s and from some time beyond the end of s always refuse the whole of the set X of visible actions, then (s, X) is a failure of P.

Thanks to Ouaknine's work on *digitisation* [6, 7], it has become possible to establish results about (continuous) Timed CSP processes[3] by proving them about a version of Timed CSP formulated over discrete time, which in turn is equivalent to proving analogous results about a *tock*-CSP translation of the discrete version. *tock*-CSP [11] is ordinary "untimed" CSP but with the special event *tock* interpreted as the regular passage of time. This translation was described in [6], and its implementation in the FDR refinement checker in [1].

That made it straightforward to check untimed *trace* properties of Timed CSP processes on FDR: all one had to do was to check untimed trace refinement of the untimed specification P by $Q \setminus \{tock\}$ for the timed implementation Q. In other words we can simply hide the special time event.

---

met because, inter alia, of the relationship of the speed of trains to the timing characteristics of the physical gate.

[2] This notation is borrowed from [16].

[3] These results are restricted to processes in which all programmer- and implementation-created delays are of integer length: *integer* Timed CSP.

The situation is not so simple for failures, since the permanent refusal of a set of events in a *tock*-time process necessarily involves an infinite series of *tock*s and quite possibly states that vary throughout time. In particular $Q \setminus \{tock\}$ will have no stable failures at all, since no discrete Timed CSP process ever refuses *tock*. Therefore the inclusion of Timed CSP in FDR did not in itself solve the problem of automating the question of timewise refinement.

Timed CSP's semantics have the property known as *maximal progress*, meaning that a process cannot do nothing when a τ action is possible. In the translation to *tock*-CSP this translates to the statement that no *tock* action can happen when a τ is possible. Another way of saying the same thing is that τ must have priority over *tock*.

For this and other reasons a priority operator has been implemented in FDR, as proposed in [12]. We shall see in the present paper that priority also provides the solution to the problem of characterising timewise refinement, using a similar idea to that used to solve a problem of abstraction in [13]. Specifically, we will find that way to map a Timed CSP process to a value that can be compared against an untimed failures specification is to use the *slow abstraction* defined in that paper.

This, in the present case, has the effect of turning any refusal (as opposed to trace) counter-example to a failures refinement into a divergence that can be detected by FDR.

The rest of this paper is organised as follows. In the next (background) section we recall the essential details of CSP, Timed CSP and their models, giving the formal definition of timewise refinement. In Section 4 we shown how slow abstraction of $\{tock\}$ provides an accurate characterisation of timewise refinement for the discrete interpretation of Timed CSP, with digitisation extending this result to the continuous interpretation.

In this paper we make the assumption that the underlying alphabet of actions Σ is finite. This is necessary because of the way we check timewise refinement later, and is common to [13]. Since FDR can only handle finite alphabets, this is therefore no restriction at all at that level.

2 Background

2.1 CSP and its semantics

In this section we give an overview of untimed CSP and its models, as relevant to the present paper. Far more extensive presentations can be found in [5, 11, 12]. CSP is based on instantaneous actions handshaken between a process and its environment, whether that environment consists of processes it is interacting with or some notional external observer. It enables the modelling and analysis of patterns of interaction. The books [5, 11, 12, 16] all provide thorough introductions to CSP. The main constructs that we will be using in this paper are set out below.

- The processes $STOP$, $SKIP$ and **div** respectively do nothing, terminate immediately with the signal ✓ and diverge by repeating the internal action τ. Run_A and $Chaos_A$ can each perform any sequence of events from A, but while Run_A always offers the environment every member of A, $Chaos_A$ can nondeterministically choose to offer just those members of A it selects, including none at all.
- $a \rightarrow P$ *prefixes* P with the single communication a which belongs to the set Σ of normal visible communications. Similarly $?x : A \rightarrow P(x)$ offers the choice A and then behaves accordingly.
- CSP has several *choice* operators. $P \,\Box\, Q$ and $P \sqcap Q$ respectively offer the environment the first visible events of P and Q, make an internal decision via τ actions whether to behave like P or Q.
  The asymmetric choice operator $P \rhd Q$ offers the initial visible choices of P until it performs a τ action and opts to behave like Q. In the cases of $P \,\Box\, Q$ and $P \rhd Q$, the subsequent behaviour depends on what initial action occurs.
- $P \setminus X$ (hiding) behaves like P except that all actions in X become (internal and invisible) τs.
- $P[\![R]\!]$ (renaming) behaves like P except that whenever P performs an action a, the *renamed* process must perform some b that is related to a under the relation R.
- $P \underset{A}{\parallel} Q$ is a *parallel* operator under which P and Q act independently except that they have to agree (i.e. synchronise or handshake) on all communications in A. A number of other parallel operators can be defined in terms of this, including $P \,|||\, Q = P \underset{\emptyset}{\parallel} Q$ in which no synchronisation happens at all.
- $P;\ Q$ behaves like P until it terminates successfully, and then like Q.

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

It is always asserted that the meaning, or semantics, of a CSP process is the pattern of externally visible communication it exhibits. As shown in [11, 12], CSP has several styles of semantics, that can be shown to be appropriately consistent with one another. The two styles that will concern us are *operational* semantics, in which rules are given that interpret any closed process term as a labelled transition system (LTS), and *behavioural* models, in which processes are identified with sets of observations that might be made from the outside.

An LTS models a process as a set of states that it moves between via actions in Σ^τ, where τ cannot be seen or controlled by the environment. There may be many actions with the same label a single state, in which case the environment has has no control over which is followed. The best known behavioural models of CSP are based on the following types of observation. *Traces* are sequences of visible communications a process can perform. *Failures* are combinations (s, X) of a finite trace s and a set of actions that the process can refuse in a *stable* state reachable on s. A state is stable if it cannot perform τ. *Divergences* are traces after which the process can perform an infinite uninterrupted sequence of τ actions, in other words diverge. The models are then

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

2.2 Timed CSP and its semantics

For thorough presentations of Timed CSP and its models, the reader should study [16, 8, 9].

Timed CSP has the same operators as CSP, plus a single constant with explicit time: $WAIT\ t$ where t is a non-negative number. This terminates with ✓ t time after it is started. More operators such as timeout and timed interrupt can be defined in terms of the basic ones. The difference between the interpretation of operators in CSP and Timed CSP is that in the latter we are precise about when communications happen and become available, while remaining faithful to the understanding contained in the untimed interpretation. So, for example the process $a \rightarrow P$ still offers the event a until it occurs. P then starts some fixed time (representing the time it takes the process to recover from, or perhaps complete, a) before P starts. And $P \,\Box\, Q$ still offers the choice of the initial (visible) events of P and Q until one of them performs one. Thinking about this makes it clear that P and Q evolve in time – perhaps performing their own τ events – side by side until one of them has an action accepted, allowing the other to be turned off.

Untimed CSP carries the assumption that an unstable state – one in which a τ action is enabled – cannot persist, with a τ or some other action definitely happening quickly. In Timed CSP this has to be quantified, leading to the concept of *maximal progress*: when a τ is enabled, some action must happen *immediately*. This simple and seemingly straightforward translation of intuition has a major effect on the semantic models that are available for Timed CSP.

In forming a semantics for Timed CSP it is clear that we have to attach times to the events that happen, since otherwise we would not be making distinctions that we evidently want to make. We assume the *No Zeno* principle that no process can perform infinitely many events (whether visible or invisible) in a finite time, but do allow a sequence of events all to happen at the same time. The main operator that constrains available models is hiding: in $P \setminus X$ no X event can be offered by P for more than zero time, for otherwise the process with X hidden would violate the principle of maximal process. It turns out that this means that we need to know what a timed process refuses at every moment of time, with the refusal at the time t of an event corresponding to what is refused instantaneously after the last event at t.

The combination of the principle of maximal progress and the need to make models compositional under the CSP hiding operator (which turns visible actions into τs that are forced before time passes) makes the range of models for Timed CSP more restricted than for untimed. It is necessary to record the set of events refused at every point in a behaviour where time advances. Divergence is a much reduced issue, since thanks to the no-Zeno assumption any divergence

is necessarily spread over infinite time – which when we are modelling time simplifies things greatly. In fact divergence will not be considered in the models we use in this paper.

In the case of continuous time this means that we have to record refusals as a subset of $\Sigma \times \mathbb{R}^+$ to accompany traces which attach a time in $\mathbb{R}^+$ (the non-negative real numbers) to each event, where the times increase, not necessarily strictly, through the trace. In fact, timed refusals are unions of sets of the form $X \times [t_1, t_2)$ where $0 \leq t_1 < t_2 < \infty$ –*refusal tokens*. $[t_1, t_2)$ is a *half-open interval* that contains t_1, all x with $t_1 < x < t_2$ but not t_2. This corresponds to the idea that if an event happens *at* time t then the refusal recorded at that time is the set of events refused at the same time *after* the event. So in $a \rightarrow P$, there will be behaviours in which a occurs at time 1, all events other than a are refused in the interval $[0, 1)$ and, on the assumption that the event a takes time δ to complete, all events including a are refused in the interval $[1, 1 + \delta)$.

So the *Timed Failures model* ($\mathcal{F}_T$) representation of a process consists of pairs of the form $(t, \aleph)$ (*timed failures*), where t us such a timed trace, and $\aleph$ is such a timed refusal. First introduced in [8], There have been a number of variants of this model over the years. The author analysed these in [14] and in this paper adopts the same version of $\mathcal{F}_T$, namely one

- Where causality is permitted within a single instant: for example one can have the timed trace $\langle (a, 1), (b, 1) \rangle$ but not the timed trace $\langle (b, 1), (a, 1) \rangle$.
- Where timed traces (as recorded) are finite (i.e. have only finitely many timed events) but where timed refusals can extend through all time, though they are finitary in the sense that they are the union of a countable set of refusal tokens $X \times [t_1, t_2)$ where the number of t_1s less than any fixed $t \in \mathbb{R}^+$ is finite.
- Where unfolding recursion takes no time, but only *time guarded* recursions in which no recursive call can be made before some $\delta > 0$ are permitted. The latter is to ensure that no process has a Zeno behaviour in which infinitely many actions can occur in a finite time.

Timed failures can be extended, if we wish, by an analogue of the divergence information used in $\mathcal{N}$. However, rather than record that a particular timed failure $(s, \aleph)$ is divergent (i.e. is accompanied by an infinite series of τs) we record the smallest time after the end of s (or ∞ if there is none) after which, when observing $(s, \aleph)$, we can be sure it must have become stable (i.e. no further τ will be enabled if no further visible event occurs. So the *Timed Failures Stability Model* $\mathcal{FS}_T$ represents a process as a set of triples $(s, \aleph, t)$ where $(s, \aleph)$ is one of its timed failures and t is the unique stability time associated with this.

Just as it is easy to extract the untimed traces of a process by deleting the times from timed traces, it is possible to extract natural values in the untimed models $\mathcal{F}$ and $\mathcal{N}$ from a process's representation in $\mathcal{FS}_T$. Untimed divergences come from timed triples $(s, \emptyset, \infty)$ by deleting the times in s to get $untime(s)$. Stable failures from triples $(s, \aleph, t)$ with $t < \infty$: the latter gives the untimed failure whose trace is $untime(s)$ and which refuses $\{a \mid \exists\, t'.(a, t') \in \aleph \wedge t' \geq t\}$.

In other words, anything that is refused after the time when the process must have become stable becomes part of the untimed refusal set.

So, for any Timed CSP process, we have a way of constructing values in each of the canonical untimed models. In each case this will refine the value you could have calculated by mapping the syntax into untimed CSP (i.e. mapping each $WAIT\ t$ component to $SKIP$). This substantiates the statement that the timed semantics is consistent with the untimed one, but the refinement might well be strict since modelling at the timed level can give us certainty about how nondeterminism in the untimed model will be resolved.

For example, consider $((a \rightarrow P) \,\Box\, (WAIT\ 1;\ a \rightarrow Q)) \setminus \{a\}$ (where $WAIT\ 1$ could be replaced by any process whose untimed semantics has just the traces $\{\langle\rangle, \langle\checkmark\rangle\}$ and which always terminates on the empty trace after a non-zero time). The untimed semantics will identify this with $P \setminus \{a\} \sqcap Q \setminus \{a\}$ since they cannot tell that the τ resulting from the hiding of the left-hand a will always happen at time 0, with the a resolving the choice and excluding Q from doing anything before it starts. So in this case we get proper refinement. While the above approach is arguably the most natural way of linking timed and untimed theories from the perspective of the untimed theory, it misses out on two important things from the point of view of the timed models.

- Firstly, it ignores the most natural Timed CSP model $\mathcal{F}_T$: divergence and stability play no essential role in the semantics of Timed CSP as they do in the untimed version.
- Secondly, it ignores the fact that we can easily observe permanent refusal of a set of events X without stability: if no member of X is accepted in the states that appear between an infinite series of τ events, necessarily taking an infinite time, then we can reasonably equate this with untimed (i.e. permanent) refusal.

The natural way of extracting failures from timed failures, recalling that our version of $\mathcal{F}_T$ permits refusals that extend over an infinite period, is to map $(s, \aleph)$ to (s', X), where X is the largest X such that there is $t \geq end(s)$ with $X \times [t, \infty) \subseteq \aleph$. We will call this function from $\mathcal{F}_T$ to sets of failures Ψ. This gets the $\mathcal{F}_T$ value of $STOP$ just right, though applying it to the same value with $\mu\, p.\, WAIT\ 1;\ p$ in mind gives (of course) the untimed semantics of $STOP$ which is nothing like that in any standard untimed model of $\mu\, p.\, WAIT\ 1;\ p$.

What we have to accept is that, for processes that untimed CSP regards as divergent, the above mapping calculates something that is different from – and for some purposes superior to – the results calculated directly in the untimed models. After all $\mu\, p.SKIP;\ p$ (the untimed analogue of $\mu\, p.\, WAIT\ 1;\ p$) will, from the perspective of the external user, sit there failing to accept any communication offered to it, just like $STOP$.

The range of Ψ is precisely the sets of failures that can occur in $\mathcal{N}$ and smaller than the set of those that occur in $\mathcal{F}$ because every untimed trace s' of the underlying process P has $(s', \emptyset)$ in $\Psi(P)$: if s is any timed trace that maps to s' then certainly $(s, \emptyset) = (s, \emptyset \times [end(s), \infty))$ is in P. In other words, Ψ maps $\mathcal{F}_T$ to the sets of failures of divergence-free processes.

Timewise refinement, as defined by Schneider, is a relation between untimed specifications *Spec*, generally expressed as divergence-free CSP processes and certainly members of the range of Ψ, and Timed CSP processes P:

$$Spec_{SF} \sqsubseteq_{TF} P \;\; \Leftrightarrow \;\; Spec \sqsubseteq \Psi(P)$$

where $\sqsubseteq$ is reverse containment over sets of failures.

We will see some examples to illustrate this definition later in this paper.

3 Digitisation and discrete time

FDR, as documented in [10, 12, 1] is a model checker for untimed CSP, whose algorithms manipulate discrete representations of discrete state machines.The key to verifying Timed CSP on it has been the theory of digitisation, in which the notation is re-interpreted over a discrete time domain (the natural numbers $\mathbb{N}$) and results proved to establish links between the semantics of a process over the two domains.

This restricts attention to *integer* Timed CSP, where all $WAIT\, t$ processes have $t \in \mathbb{N}$, with these the only delays introduced by a process as opposed to its environment. In the continuous semantics of this language, events can still happen at any time in $\mathbb{R}^+$, but in the discrete semantics they only happen at members of $\mathbb{N}$. Corresponding to $\mathcal{F}_T$ there is a *discrete timed failures model* $\mathcal{F}_{DT}$ in which there is a single refusal set following the zero or more events that happen at each integer time. There are a number of possible representations of this model, but we follow [14]: the extra event *tock* (which allows us to count the present time, and has no analogue in the continuous model) represents the regular passage of time, with all events between each pair of *tock*s being considered to happen at the same time as the preceding *tock*. (Events preceding the first *tock* happen at time 0.) There is a refusal set before each *tock* in each such trace. Traces are infinite but contain only finitely many non-*tock* events.

The theory of *digitisation* was introduced by Henzinger, Manna and Pnueli in [4] as a way of proving properties about continuous systems (specifically, timed automata) by analysing discrete approximations. It was adapted for Timed CSP by Ouaknine [6, 7] who showed that one can prove certain properties of systems over the continuous model $\mathcal{F}_T$ by demonstrating analogous properties of the same process's discrete semantics over $\mathcal{F}_{\mathbf{DT}}$. In particular he showed that every integer Timed CSP program has the property of being *closed under digitisation*, meaning that if $(s, \aleph)$ is in its $\mathcal{F}_T$ representation, then so is $[(s, \aleph)]_\epsilon$ for each $0 < \epsilon \leq 1$: this transforms each event and end-point of a refusal token $X \times [t_1, t2)$ to itself is an integer, and to one of the two surrounding integers otherwise:

- t_1 is $\lfloor t \rfloor$, where $\lfloor t \rfloor$ is the largest integer no greater than t.
- For $\epsilon < 1$, $t_\epsilon = \lfloor t \rfloor$ for if $frac(t) < \epsilon$ and $\lceil t \rceil$ otherwise, where $frac(t) = t - \lfloor t \rfloor$ and $\lceil t \rceil$ is the smallest integer no less than t.

Such integer behaviours map naturally to those recorded in $\mathcal{F}_{\mathbf{DT}}$ and are in fact members of the process's semantics in that model. It follows that if,

whenever the continuous time semantics of a process have a timed failure $(s, \aleph)$ that violates some specification S, there is some ϵ such that $[(s, \aleph)]_\epsilon$ also fails it, then we can determine whether an integer Timed CSP process satisfies S by considering only the discrete semantics.

A good example is provided by timewise refinement: if P fails to be a timewise refinement of the untimed specification S, this can only be because $\Psi(P)$ contains a failure not in S, or in other words there is a timed failure of the form $(s, X \times [t, \infty))$ for some s, $t \geq end(s)$ and X such that $(untime(s), X) \notin S$.

The digitisation property set out above implies that, in fact for *any* ϵ, $[(s, X \times [t, \infty)]_\epsilon$ is an integer behaviour that Ψ maps to $(untime(s), X)$. It follows that $S_{SF} \sqsubseteq_{TF} P$ if and only the same thing holds when judged over the discrete time semantics.

As set out in [6, 12], the discrete time semantics for Timed CSP can be calculated by systematically translating the Timed CSP language to *tock*-CSP, namely the usual CSP language with the addition of the special event *tock* representing the regular passage of time[4]. This translation has been automated in both FDR2 [1] and prototypes of FDR3. The essence of this that any process syntax contained within a `Timed(et){...}` section is automatically translated into the corresponding *tock*-CSP processes, using the *event timer* `et`, namely a mapping from events to the integer times taken to complete them.

The maximal progress property of Timed CSP is implemented by prioritising internal τ actions over *tock*. In other words, *tock* actions can only occur from stable states of the standard CSP operational semantics of the *tock*-CSP process.

This uses the priority operator $\mathbf{Pri}_{\leq}(P)$ specified in [13, 12], and now implemented as `prioritise(P,As)` in which `P` is a process and `As` a list of disjoint sets of events. `head(As)` consists of the events whose priority is equivalent to τ, which successive sets having lower priority. The operational semantics is that if the priority of event x is higher than that of event y, then y cannot occur from a state where P has x available. Events not in the union of `As` have no place in the priority order: they neither prevent and nor are they prevented by others. In the "blackboard" version $\mathbf{Pri}_{\leq}(P)$, $\leq$ represents a partial order on $\Sigma \cup \{\tau\}$ with some restrictions on the position of τ that are discussed in [13], and which are automatically satisfied by this machine readable version.

4 Slow abstraction

The idea of slow abstraction was introduced in [13]. $\mathcal{S}_A(P)$ represents how P appears to a user who can see the complement of A, on the assumption that events in that set are controlled by a user who habitually delays them, but not permanently. Thus the process is not prevented in making its offers outside A, but equally it is never permanently blocked by the abstracted user.

To consider $\mathcal{S}_A(P)$ we assume (as with lazy abstraction) that P is divergence-free. It differs from $P \setminus A$ in that as well as recording the refusals P makes when

[4] In fact this translation is to an extended version of *tock*-CSP since some Timed CSP operators need new untimed operators as their analogues.

it can refuse the whole of A (so $P \setminus A$ is stable), we also look at the series of sets it can refuse prior to it accepting each member of A in an infinite sequence of these. If all of these refuse a set X, then P will obviously not accept any member of X along the sequence.

This makes sense in the context of *slow* abstraction because we can suppose that all but finitely many A events happen from stable states, the abstracted user having waited for the divergence-free process to complete all its urgent τs before performing the next member of A.

Because such an *unstable refusal* can only happen at the end of an observed behaviour, and necessarily each finite trace of $P \setminus A$ is followed either a stable or unstable refusal in $\mathcal{S}_A(P)$, we identify it with a set of failures which always corresponds to the failures of a member of the failures divergence model $\mathcal{N}$.

Slow abstraction was introduced and studied in [13], and in particular a technique was introduced for deciding using FDR whether a failures specification *Spec* is refined by $\mathcal{S}_A(P)$. When an unstable failure of the latter violates *Spec*, this necessarily takes an infinite number of steps, and so the only way that FDR can detect this is as a divergence. This means that $\mathcal{S}_A(P)$ cannot be realised directly in the language of FDR, but rather we have to check the refinement somewhat obliquely. We will introduce it below for the case relevant to the present paper, namely when $A = \{tock\}$ and P is the *tock*-CSP translation of an integer Timed CSP implementation under the timed-priority model which ensures that all *tock*s happen from stable states.

For such a process, $P \setminus \{tock\}$ is never stable since no Timed CSP process ever refuses *tock*. In fact it should not be too hard to see that $\mathcal{S}_{\{tock\}}(P)$ consists of exactly the failures $\Psi(P)$ used in determining timewise refinement, so in fact the statements $Spec \sqsubseteq \mathcal{S}_{\{tock\}}(P)$ and $Spec_{SF} \sqsubseteq_{TF} P$ are equivalent.

The method for deciding this is best introduced in two stages. Consider the process

$$(\mathbf{Pri}_{\leq}((P[\![tock, tock'/tock, tock]\!])) \setminus \{tock\})$$

where $\leq$ prioritises every event other than the new event $tock'$ over $tock$; $tock'$ is incomparable with all. In this, the τs created by hiding *tock* can only happen from states of P in which only *tock* is possible. On the other hand, every state of P is reachable thanks to $tock'$. Therefore the above process can diverge if and only if P has a state which has an infinite behaviour consisting of a mixture of τs and *tock*s, the latter from states offering just *tock*.

This is exactly equivalent to P being a timewise refinement of the deadlock free specification

$$DF = \sqcap\{a \rightarrow DF \mid a \in \Sigma \setminus \{tock, tock'\}\}$$

We therefore know how to solve our problem for this specific *Spec*.

Following [13], we can generalise this to arbitrary *Spec* with the combination of the trace check $Spec \sqsubseteq_T P \setminus \{tock\}$ and checking the combination of P and a testing process $Test(Spec)$ against the unstable deadlock specification above. $Test(Spec)$ is constructed as follows.

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

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

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

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

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

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

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

Our construction turns any case where Q fails $Spec$ into a deadlock. It is very similar to the "watchdog" transformation for the usual failures model set out in [2]. The main difference is that ours is constructed with no τ actions: the visible action d replaces τ.

Consider the case where Q is replaced by P. This has the additional event $tock$ which is not synchronised with $Test(Spec)$, so the combination $Test(Spec) \underset{\alpha Spec}{\parallel} P$ can only deadlock in states where $P \setminus \{tock\}$ has a stable failure illegal for $Spec$. In fact this never happens for us because, as remarked above, $P \setminus \{tock\}$ is never stable.

We would like unstable failures of $\mathcal{S}_{\{tock\}}(P)$ to turn into unstable "deadlocks", namely unstable refusals of Σ, in $\mathcal{S}_{\{tock\}}(Test(Spec) \underset{\alpha Spec}{\parallel} P)$. This is confirmed by the following result, the proof of which is essentially the same as that about general $\mathcal{S}_A(P)$ in [13], when we take into account that in our case $P \setminus \{tock\}$ is never stable.

Theorem 1. *Under our assumptions, including* $Spec \sqsubseteq_T P \setminus \{tock\}$, $\mathcal{S}_{\{tock\}}(P)$ *has an unstable failure that violates Spec if and only if* $\mathcal{S}_{\{tock\}}(Test(Spec) \underset{\alpha Spec}{\parallel}$

P) has an unstable failure of the form (s, Σ)*. Furthermore* $\mathcal{S}_{\{tock\}}(P) \sqsupseteq_F$ *Spec if and only if*

$$(\mathbf{Pri}_{\leq}((Test(Spec) \underset{\alpha Spec}{\|} (P[\![tock, tock'/tock, tock]\!]))) \setminus \{tock\}$$

is divergence-free.

When combined with our earlier observation that, thanks to a digitisation argument, proving timewise refinement of a *Spec* for the *tock*-CSP translation P' of an integer Timed CSP process P also proves for P, the above result gives us a way of deciding questions of timewise refinement on FDR.

5 Examples

The above methods have proved both effective and efficient in the examples we have experimented with to date. A variety of CSP files is included with version of this paper posted on the author's web site. These include Timed CSP versions of the Alternating Bit and Sliding Window Protocols, which are naturally shown to give timewise refinements of the untimed buffer specification. These files use the same versions of these protocols used in example files published with [1], but whereas the specifications proved of the original versions were inevitably parameterised by timing details, the new versions show how to establish untimed correctness in a form requiring no timing details.

This has the efficiency advantage that there is no need to experiment with different timing parameters when verifying a timed system, and in many cases the absence of timing in the specification reduces the overall number of states visited for a given implementation P. However the need to use divergence checking places extra demands on FDR which are always likely to impose moderate restrictions on the size and speed of the check.[5]

Consider the following machine-readable Timed CSP description of a token ring:

```
datatype Packet = Full.(Nodes,Nodes,Data) | Empty

channel ring:Nodes.Packet
channel input,output:Nodes.Nodes.Data
succ(n) = (n+1)%N
```

[5] At the time of writing (May 2013) the author has the choice of using FDR2 or a prototype FDR3. In FDR2 the data structures used make relatively small (say 8M or less checks) typically almost as fast as checks for finitary properties, but slow down substantially beyond that. FDR3's checking for divergence is more efficient than FDR2's and less limited, but presently operates essentially sequentially, whereas traces and stable failures checking use parallelism across multi-core devices. This situation is likely to improve, but parallel checking of divergence properties is always likely to be more difficult and less efficient than the parallelism available for checking finitary properties.

```
allone(X) = 1

Timed(allone){
Token(k,Empty) = input.k?dest?x -> Token(k,Full.(k,dest,x))
                 [] (WAIT(1);ring.succ(k).Empty -> NoToken(k))

Token(k,Full.(f,t,x)) = if t==k then
                        output.k.f.x -> ring.succ(k).Empty -> NoToken(k)
                        else ring.succ(k).Full.(f,t,x) -> NoToken(k)

NoToken(k) = ring.k?p -> Token(k,p)

Alpha(k) = {|ring.k,ring.succ(k),input.k,output.k|}

init(k) = if k <  Tokens then Token(k,Empty) else NoToken(k)

RING = (|| k:Nodes @[Alpha(k)]init(k))\{|ring|}
}
```

This sets up an `N`-place circular ring with `Tokens` tokens that rotate around it, carrying messages between the nodes. We might ask the question of whether it acts as a buffer between each ordered pair of nodes. An untimed specification which expresses this for fixed nodes `i` and `j` is

```
Buff(i,j,<>) = input.i?k?x -> if j==k then Buff(i,j,<x>)
                                      else Buff(i,j,<>)

Buff(i,j,xs^<x>) = output.j.i.x -> Buff(i,j,xs)
                 [] if #xs == Tokens-1 then STOP
                  else (STOP |~|
                     input.i?k?y -> if j==k then Buff(i,j,<y>^xs^<x>)
                                    else Buff(i,j,xs^x>))
```

This has as its alphabet all inputs at node `i` and the outputs at `j` that come from `i`.

The right way to judge our ring against this is to hide all other outputs and lazily abstract all other inputs. This corresponds to the assumption that users accept all outputs from the ring eagerly, and can arbitrarily decide what messages other than those considered by the specification.

The main complication in checking timewise refinement they way we are advocating is the need to need to transform the untimed specification into the corresponding testing process. In our case this is

```
Test(i,j,<>) = [] x:Data @ d -> input.i.j.x -> DS
               [] (|~| k:Nodes @ if k==j then
                                input.i.j?x -> Test(i,j,<x>)
                         else input.i.k?x -> Test(i,j,<>))
```

```
Test(i,j,xs^<y>) = d -> output.j.i.y -> DS
                   [] output.j.i.y -> Test(i,j,xs)
                   [] #xs < Tokens-1 & (|~| k:Nodes @ if k==j then
                                          input.i.j?x -> Test(i,j,<x>^xs^<y>)
                                       else input.i.k?x -> Test(i,j,xs^<y>))
```

Fortunately this process is automatable and could be implemented as primitive in a future version of FDR.

The ring fails this specification because the other nodes might repeatedly fill every token (in the lazy abstraction) before it reaches node `i`. In other words it fails the aspect of the failures specification that the ring, when there is no message in transport from `i` to `j`, it must eventually accept an input at `i`. The error shows up as a divergence in FDR, whose debugger decomposes this into the `Test` process constantly offering node `i` an input, and each of a cycle of states of the token ring refusing that input.

This can be repaired by adjusting the tokens that pass around so that upon being emptied they do not immediately gain the ability to transport another message. Rather, successive tokens gain this ability when they reach one of the nodes, where the node with this property rotates. Thus no node can indefinitely be denied the ability to accept a message from its user.

The versions of the ring with and without this added control can be found amongst the files available with this paper. The version with does satisfy the requirement under timewise refinement.

The modified ring, in common with the original one set out above, only makes input offers to each user for a single unit of time each time an empty token passes. This illustrates the fact that satisfying a specification formulated in terms of timewise refinement does not imply that offers the specification requires must be made permanently beyond some point; rather that they must be made at an infinite number of times beyond some point. The implementation is allowed to make progress (in our case via the tokens circulating) and changing the set of events offered.

6 Conclusions

We have given a practical and theoretically sound way of checking timewise refinement in FDR. It proved to be straightforward to link the usual relation between specifications and continuous Timed CSP, to a tractable relation between the same specification and the *tock*-CSP process corresponding to the continuous one.

Testing this requires both a renaming and hiding trick which converts illegal unstable failures into divergences, and the conversion of the specification into a suitable watchdog process.

We have demonstrated that this approach works for a range of examples, and hope that others find it useful. To enable this it would be helpful if functionality were added to FDR to create a $Test(Spec)$ process automatically from each *Spec*

References

1. P. Armstrong, G. Lowe, J. Ouaknine, and A.W. Roscoe, *Model checking Timed CSP*, To appear in Proceedings of HOWARD, Easychair.
2. M.H. Goldsmith, N. Moffat, A.W. Roscoe, T. Whitworth and M.I. Zakiuddin, Watchdog transformations for property-oriented model-checking, FME 2003: Formal Methods, LNCS 2805, 2003.
3. C.L. Heitmeyer, B.G. Labaw and R.D. Jeffords, *A benchmark for comparing different approaches for specifying and verifying real-time systems*, DTIC, 1993.
4. T.A. Henzinger, Z. Manna, and A. Pnueli, *What good are digital clocks? In Proceedings of the Nineteenth International* Colloquium on Automata, Languages, and Programming (ICALP 92), volume 623, pages 545-558. Springer LNCS, 1992.
5. C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.
6. J. Ouaknine, *Discrete analysis of continuous behaviour in real-time concurrent systems*, Oxford University D.Phil thesis, 2001.
7. J. Ouaknine, *Digitisation and full abstraction for dense-time model checking*, TACAS Springer LNCS, 2002.
8. G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, Theoretical Computer Science **58**, 249-261, 1988.
9. G.M. Reed and A.W. Roscoe, *The timed failures-stability model for CSP*, Theoretical Computer Science **211**, 85-127, 1999.
10. A.W. Roscoe, *Model checking CSP*, in 'A classical mind: essays in honour of C.A.R. Hoare', Prentice Hall, 1994.
11. A.W. Roscoe, *The theory and practice of concurrency* Prentice Hall, 1997.
12. A.W. Roscoe, *Understanding concurrent systems*, Springer, 2010.
13. A.W. Roscoe and P.J. Hopcroft, *Slow abstraction through priority*, To appear in Festschrift proceedings for He Jifeng, ICTAC 2013.
14. A.W. Roscoe, and Huang Jian *Checking noninterference in Timed CSP*, FAC, **25**, pp 1-33, 2013.
15. S.A. Schneider, *Timewise refinement for communicating processes*, Science of Computer Programming, **28**, pp 43–90, 1997.
16. S.A. Schneider, *Concurrent and real-time systems: the CSP approach*, Wiley, 2000.