Model checking Timed CSP

Philip Armstrong Gavin Lowe Joël Ouaknine
A.W. Roscoe

Oxford University Department of Computer Science

Abstract

Though Timed CSP was developed 25 years ago and the CSP-based refinement checker FDR [25] was first released 20 years ago, there has never been a version of this tool for Timed CSP. In this paper we report on the creation of such a version, based on the digitisation results of Ouaknine [16, 17] and the associated development of discrete-time versions of Timed CSP with associated models [19, 14, 11, 27].

Dedication: I have happy memories of chasing time in the 1980s with Howard Barringer and others. Now it seems to be catching us up!

Bill Roscoe

1 Introduction

In this paper we report on what we believe to be the first attempt to create a model checking tool for the Timed CSP language, introduced by Reed and Roscoe [23, 24] as a real-time interpretation of Hoare's CSP notation [10]. In its usual form, Timed CSP adds a single construct to CSP, namely *WAIT t* which waits t units of time[1] before terminating successfully ($\checkmark$) like the *SKIP* process does immediately. It is possible to express a wide variety of time-based operations such as time-outs in terms of *WAIT t* and standard CSP.

Thanks to the idea of *digitisation* introduced by Henzinger, Manna and Pnueli [9] and developed for Timed CSP by Ouaknine [16, 17], it proves possible to do this by a relatively modest modification to FDR along the lines suggested in [16, 27]. This modification takes the form of a directive `Timed(et){...}` within FDR that tells it to interpret the syntax within the `{...}` as a (discretely) Timed CSP.

In the next section we give a summary of the history of Timed CSP, the discretely-timed dialect of CSP called *tock*-CSP and verification techniques for continuously timed systems. We then summarise the main details of the semantic models of untimed, continuous time, and discrete time versions of CSP. We show how the discrete variant of Timed CSP is related to the continuous version by digitisation and some practical implications for correctness and refinement checking. Section 5 describes the translation of Timed CSP into *tock*-CSP augmented by special versions of the external choice $\Box$ and interrupt $\triangle$ operators, and making use of the priority features recently implemented in FDR [28]. Section 6 describes how this is embedded in FDR. Finally, we give some case studies.

2 History

Timed CSP [23, 24] was developed by Mike Reed and Roscoe in the mid and late 1980s as a formalism which added exact "real" time to the CSP process algebra of [10]. As such it was one of a number of contemporaneous efforts to bring real time into formal semantics and

[1]It is generally left unspecified what the units of time are in Timed CSP and its models.

formal methods. Many of those working on these developments, including Howard Barringer and several people no longer with us, such as Rob Gerth and Amir Pnueli, collaborated on the ESPRIT projects SPEC and REX, with Reed bringing Timed CSP to the group. It was of course unrealistic to suppose that such an effort would bring agreement on a single appropriate theory, but it was tremendously helpful in cross fertilising and comparing ideas. Indeed [24] notes how similar the basic underlying assumptions of Timed CSP are to those of [2].

One of the key principles of Timed CSP, and one which will be hugely important in the present paper, is *maximal progress* (which we paraphrase as "as soon as an internal τ action becomes available, some action (which might be τ or a visible action) occurs". Timed CSP was originally given a continuous time domain (the non-negative real numbers $\mathbb{R}^+$), so events could be recorded at arbitrary times in $\mathbb{R}^+$ and the t of $\mathit{WAIT}\ t$ could similarly take any such value.

Timed CSP was developed in many further works such as [6], and was described extensively in books by Davies [5] and Schneider [33]. A "retrospective" can be found in [18].

At the time of its creation, the continuous time domain was seen as a major barrier to automated tool support. However, in the context of other timed formalisms, it became understood that techniques such as region graphs [1] could reduce certain questions about continuously timed systems to finite-state decision procedures provided that delays, and constants used to test and assign to clocks, were restricted to integers. (Thus time and clocks would still take values in $\mathbb{R}^+$, but the range of operations and comparisons on them would be limited to ones involving integers.) First developed in papers such as [1, 8], David Jackson [13] showed that these same ideas could be applied to Timed CSP via *action timed graphs*.

In the meantime FDR had been developed for "untimed" CSP: it was and remains a tool that supports the exploration of the large discrete state spaces that result from concurrent CSP systems. The creation of a tool to support Jackson's work on Timed CSP would have needed to have been a separate exercise based on completely different modes of verification, and this was never done since in any case other tools such as Uppaal [3] were developed to verify timed systems using the region graph approach.

In 1992 Roscoe developed a discrete time dialect of "untimed" CSP based on using a special event *tock* to represent the regular passage of time. This was done in response to the challenge [7] to model a level crossing gate (subsequently used as an example in Chapter 14 of [26]), but later became widely used across a range of practical applications such as fault tolerant systems [4] and protocols [31].

This "*tock*-CSP" is syntactically very different from Timed CSP, because it includes this event explicitly. It is possible to achieve a very wide range of timing effects in *tock*-CSP including ones not achievable in standard Timed CSP, such as *urgent* states (ones that will not allow time to progress until some other visible event has occurred). It is also possible to write processes with contradictory timing models: ones that have no way of letting time progress, called *time-stops*.

A translation from Timed Automata to *tock*-CSP was described in [12].

tock-CSP is probably the right CSP variant to use for describing systems where the clock signal explicitly affects the control behaviour of the processes in a network. Then, including *tock* as an explicit event that processes synchronise on becomes natural.

However, as described in Chapter 14 of each of [26, 27], it can also be used to describe systems where regular *tock*s are rather some sort of external measure of elapsed time in a real system, but where there is no explicit clock signal in the real implementation. The main problem in that case is that there is a disconnect between the program representing a component of the real system (where no *tock* event is mentioned) and the corresponding *tock*-CSP component (where they are).

For example, the *tock*-CSP that corresponds most naturally to the simple one-place buffer process $COPY = \mathit{left}?x \rightarrow \mathit{right}!x \rightarrow COPY$ is

$$\begin{array}{rcl} TCOPY2 & = & \mathit{left}?x \rightarrow \mathit{tock} \rightarrow TCOPY2'(x) \\ & & \Box\ \mathit{tock} \rightarrow TCOPY2 \\ TCOPY2'(x) & = & \mathit{right}!x \rightarrow \mathit{tock} \rightarrow TCOPY2 \\ & & \Box\ \mathit{tock} \rightarrow TCOPY2'(x) \end{array}$$

which takes one time unit after inputting or outputting to move to a state where it can do the next output or input respectively, and which can wait indefinitely to perform outputs and inputs.

While that can be said to paint a reasonable picture of how a process behaves relative to an external clock, it is remote from an actual implementation where there is no direct causal relationship with that clock.

In fact Timed CSP is much better suited to this second purpose, since the underlying timing model is one where we observe how processes behave as time passes rather than having them directly signalled by it. Thus Timed CSP gives an interpretation to the syntax of $COPY$ above which is the same timed behaviour implied by the *tock*-CSP syntax $TCOPY2$, at least on the assumption that after performing a in $a \rightarrow P$ it takes one time unit before P starts: the observer with a discrete clock must see at least one *tock* event between each $\mathit{left}.x$ or $\mathit{right}.x$ and the next non-*tock*.

When *tock*-CSP was introduced, no formal semantic connection with Timed CSP was known or particularly anticipated, not least because Timed CSP had only its continuous time semantics. However, in [16, 17], Ouaknine showed that Timed CSP could be given discrete-time semantics in which the passage of time is represented by *tock*s in traces and that these semantics could be related to the continuous time semantics using a version of the theory of digitisation. In effect, each piece of Timed CSP syntax in which all delays are integers, and all events take an integer time to complete, is mapped to the semantics of a piece of *tock*-CSP syntax. $COPY$, under the assumptions above, is essentially mapped to $TCOPY2$.

Proving properties of the discrete interpretation of a piece of Timed CSP syntax can, thanks to digitisation, establish properties of the continuous interpretation. This is because one can show that, for *integer* Timed CSP (i.e. where all delays introduced by $\mathit{WAIT}\ t$ and other syntax are non-negative integers), the continuous and discrete semantics are congruent to each other in various interesting ways.

3 CSP's and Timed CSP's semantic models: a summary

To understand the relationship between CSP and Timed CSP, and between discrete and continuous Timed CSP, it helps to know something of their semantic models, in which a process is represented as a set of observable behaviours.

The two most abstract (i.e., identifying most processes) models for untimed CSP (see [27]) represent processes either as a prefix-closed and nonempty set T of finite traces, or as the pairing of such a T with an extension-closed subset D, representing the set of traces on which the process can diverge. These are the *traces* model $\mathcal{T}$ and the *divergence-strict* traces model $\mathcal{T}^{\Downarrow}$.

Divergence strictness means that as soon as a process can diverge, we ignore any detail on extensions of the behaviour (here a trace) on which divergence happens. At first sight one might think that adding divergence information in $\mathcal{T}^{\Downarrow}$ means that this model gives strictly more information about a process than $\mathcal{T}$, but in fact divergence strictness means that $\mathcal{T}^{\Downarrow}$ does not distinguish $\mathbf{div} \sqcap (a \rightarrow STOP)$ and $\mathbf{div}$ (where $\mathbf{div}$ is a simply divergent process like $\mu\, p.p \sqcap p$), but $\mathcal{T}$ does.

The next most refined models of untimed CSP are the *stable failures* model $\mathcal{F}$ and the *failures divergences* model $\mathcal{N}$. These both use the concept of a *failure* (s, X), namely a finite trace s coupled with a *refusal* set X that the process refuses in a stable (i.e. τ-free) state after s. $\mathcal{F}$ also records a process's finite traces, because there may be traces on which P never becomes stable. $\mathcal{N} = \mathcal{F}^{\Downarrow}$ is a divergence-strict model and records the same divergence traces as $\mathcal{T}^{\Downarrow}$.

There is a hierarchy of untimed CSP models above these, as discussed in [27], the most interesting of which from the point of view of Timed CSP are those based on *Refusal Testing*, since these record a refusal set before every event. The stable refusal testing model $\mathcal{RT}$ models a process via behaviours such as $\langle\{a\}, b, \bullet, b, \emptyset, a, \Sigma\rangle$ in which observed 'refusals' and events alternate. The observed refusal can take the value $\bullet$, meaning that no actual refusal was observed because the process was not seen to be in a stable (τ-free) state. This is different from observing the refusal $\emptyset$, which can only occur in a stable state. $\bullet$ means the non-observation of stability, rather than the observation of instability: wherever it is possible to observe the refusal of $X \neq \bullet$ it is also possible to observe the refusal of any $Y \subseteq X$, or $\bullet$. We will see this model in action in Section 6.1.

The classic model for continuous Timed CSP is the Timed Failures model $\mathcal{F}_{\mathbf{T}}$ [24] which represents a process as a sequence s of events with exact times attached (with the times increasing, though not necessarily strictly), namely a *timed trace*, together with a record $\aleph$ of a refusal set at every moment. Technically, $\aleph$ is the union of a finite collection of Cartesian products $X \times [t_1, t_2)$ for X a set of events and $0 \leq t_1 < t_2$ real numbers, representing the refusal of X from the moment t_1 up to, but not including, the moment t_2.

In fact a timed failure is equivalent to having an untimed failure at every moment, with all but finitely many of them having an empty trace, and with the refusal sets belonging to these failures only varying finitely over time.

Several consistent variants on the abstract semantics of discrete Timed CSP can be found in [16, 17, 19, 14]. Like continuous Timed CSP, thanks to the demands of maximal progress, any working semantic model of a process P needs to record what is refused at every moment that time progresses. This means that a refusal set is recorded before every *tock* event. This is necessary to get a compositional semantics for the CSP hiding operator, since in $P \setminus X$, we must force all available X events to take place before letting a *tock* happen: in other words, we cannot let *tock* happen in $P \setminus X$ unless P is refusing the whole of X.

The papers mentioned differ in their extensions to the language and whether refusals are recorded at other points in the traces.

There is therefore no model for Timed CSP – either discrete or continuous – that is as simple as traces or even failures.

The most abstract model for discrete Timed CSP records a process's behaviour as some representation of a finite sequence of failures over the ordinary alphabet Σ (not augmented by *tock*) followed by the trace that occurs after the last recorded *tock*.[2] This is the *discrete timed failures model* $\mathcal{F}_{\mathbf{DT}}$. We will represent its behaviours in the same way as for refusal testing, but now with refusal sets (proper ones, not $\bullet$) occurring only before *tock*s; for example $\langle\{a\}, tock, b, b, \emptyset, tock, a, \Sigma, tock\rangle$.

In the continuous as in the discrete model, the process $P \setminus X$ cannot progress through time except when X is refused: in other words its behaviours derive from ones of P where X is

[2]It would be equivalent in expressive power to record the untimed failure that occurs after the last *tock*, because any (discrete) Timed CSP process necessarily becomes stable and never refuses *tock*.

refused at every *tock* in the discrete model and at every moment in the timed failures model.

To make this last construction work well theoretically, we need to be able to assume that P cannot perform an infinite sequence of events (e.g. from X) in a finite interval. For otherwise, when hiding X, we might require an infinite number of events in P to reach some finite time in $P \setminus X$. To avoid this difficulty most presentations of Timed CSP find some way of specifying the absence of Zeno behaviour: infinitely many events in a finite time interval.

Timed CSP can be given a semantics over either the discrete or continuous versions of the timed failures model, the former only if all t's in $WAIT\ t$ constructs, and all other delays introduced by the semantics, are non-negative integers. We call that restricted language *integer* Timed CSP. In that restricted case the theory of *digitisation* which we summarise in the next section shows that there are strong relationships between a process's continuous and discrete semantics, and that we can frequently infer properties of its continuous semantics by analysing its discrete semantics.

That type of result is one of the main motivations for the implementation of discrete Timed CSP in FDR.

Time and priority in *tock*-CSP

In order to achieve maximal progress, as described earlier, we need to stop time progressing, or *tock* events happening, while there are events enabled. Since the environment has the ability to disable (by not offering) all other events, the only ones that need concern us are the invisible event τ representing internal steps, and the termination signal $\checkmark$. Thus we need to adapt the operational model of how LTS semantics are executed, just as we need to incorporate extra refusal sets into abstract semantics.

If we connect the output of one $TCOPY2$ to the input of another, hide the internal channel and synchronise on *tock*, maximal progress is required to ensure that a data item input by the first process ever reaches the second: otherwise an infinite chain of *tock*s could happen with the τ action representing the transfer of this item enabled.

This form of prioritised execution is necessary for the correct behaviour of both *tock*-CSP and the *tock*-CSP translations of Timed CSP.

The priority operator implemented in FDR[3] takes either of the equivalent forms:

```
prioritise(P,X1,...,Xn)
prioritise(P,<X1,...,Xn>)
```

In other words, its arguments are a process and a number of sets of events, which can be presented as a single list. The sets of events should be disjoint, and the first can be empty. They need not cover all the visible events of `P`: other actions are outside the priority order. There is no point in using this operator when `n<2` since it is then the identity.

The internal action τ and termination signal $\checkmark$ are always given priority equivalent to all the members of `X1`. The operator acts on the operational semantics of `P` by preventing any action in `Xi` (for `i>1`) when τ, $\checkmark$ or an action in some `Xj` (`j<i`) is possible.

So `prioritise(P,{},{tock})` stops `tock` from happening when τ or $\checkmark$ is available. This is exactly what is required to impose maximal progress. For further uses of priority in CSP, see Chapter 20 of [27] and [28].

[3]Priority in this form is a recent innovation, though modes supporting the use of priority in timed systems have existed for some time.

4 Digitisation

The theory of digitisation enables one to formalise the relationship between the *continuous* and *discrete*—or more precisely *integral*—behaviours of Timed CSP process, thereby reducing verification questions about the former to automatable problems about the latter. Digitisation was originally introduced by Henzinger *et al.* [9] to reason about timed systems equipped with a timed trace semantics.

As stated in Section 3 the standard semantics for (continuous) Timed CSP is the timed failures model, whereby to each process P one associates a set $\mathcal{F}_{\mathbf{T}}(P)$ of timed failures. Recall that a timed failure consists of a pair $(s, \aleph)$, where s is a timed trace and $\aleph$ is a timed refusal, i.e., a finite union of sets of the form $X \times [t_1, t_2)$, where X is a set of events and the t_i's are non-negative real numbers.

A timed failure is said to be *integral* if its timed trace comprises only integral timestamps and its timed refusal only features integral endpoints. It is a simple observation that integral timed failures are in one-to-one correspondence with refusal traces in which (proper) refusals are only recorded immediately prior to occurrences of *tock*s: indeed, in this correspondence the special event *tock* corresponds to the passage of exactly one time unit. Integral timed failures can therefore alternatively be viewed as elements of the $\mathcal{F}_{\mathbf{DT}}$ model.

A real-time specification S can be expressed as a set of timed failures. A Timed CSP process P is deemed to satisfy S iff $\mathcal{F}_{\mathbf{T}}(P) \subseteq S$. This approach is of course entirely consistent with the standard notion of refinement in CSP. Unfortunately, sets of timed failures are typically uncountably large, and it is often more convenient—especially from the point of view of automation—to reason instead about integral timed failures.

For U a set of timed failures, let us therefore write $\mathbf{Z}(U)$ to denote the set of integral timed failures in U. The theory of digitisation provides sufficient conditions to infer that $\mathcal{F}_{\mathbf{T}}(P) \subseteq S$ from $\mathbf{Z}(\mathcal{F}_{\mathbf{T}}(P)) \subseteq \mathbf{Z}(S)$.

To this end, let us introduce one additional piece of notation. For t a real number, let us write $t = \lfloor t \rfloor + t'$ as its decomposition into integral and fractional parts. Now given $0 \le \varepsilon \le 1$, if $t' < \varepsilon$, let $[t]_\varepsilon$ be $\lfloor t \rfloor$, and otherwise let $[t]_\varepsilon$ be $\lceil t \rceil$. The $[\cdot]_\varepsilon$ operator therefore shifts the value of a real number t to the preceding or following integer, depending on whether the fractional part of t is less than the 'pivot' ε or not. $[\cdot]_\varepsilon$ naturally extends to timed failures by pointwise application to the timestamps of timed traces and the endpoints of timed refusals (noting, for the latter, that the operation is independent of the particular representation of the timed refusal as a finite union of sets of the form $X \times [t_1, t_2)$).

We say that a set of timed failures is *closed under digitisation* if, for any $0 \le \varepsilon \le 1$, $[U]_\varepsilon \subseteq U$.

It turns out that all Timed CSP processes in which all delays are integral are closed under digitisation. A proof of this fact can be found in [16].

We say that a set of timed failures S is *closed under inverse digitisation* if, whenever a timed failure $(s, \aleph)$ is such that $[(s, \aleph)]_\varepsilon \in S$ for all $0 \leqslant \varepsilon \leqslant 1$, then $(s, \aleph) \in S$.

The following is now a simple observation: for U and S sets of timed failures, if U is closed under digitisation and S is closed under inverse digitisation, then $U \subseteq S$ iff $\mathbf{Z}(U) \subseteq \mathbf{Z}(S)$. Combining everything together, let P be a Timed CSP process in which all delays are integral, and let S be a specification (set of timed failures) that is closed under inverse digitisation. Then

$$\mathcal{F}_{\mathbf{T}}(P) \subseteq S \quad \text{iff} \quad \mathbf{Z}(\mathcal{F}_{\mathbf{T}}(P)) \subseteq \mathbf{Z}(S).$$

Examples of specifications closed under inverse digitisation include the following:

- *Qualitative* (i.e., untimed) properties. For example, the requirement that the event a never occur.

- *Bounded-response* properties. For example, that every a be followed within k time units by a b (for k an integer).

- *Bounded-invariance* properties. For example, that whenever an a occurs, the event b should be refused over the following k time units (again, for integer k).

Further and more elaborate instances of specifications closed under inverse digitisation can be found in [19].

Note that the use of digitisation is predicated upon the calculation of the set $\mathbf{Z}(\mathcal{F}_{\mathbf{T}}(P))$ of integral timed failures of a Timed CSP process P. As noted earlier, integral timed failures are in one-to-one correspondence with elements of the $\mathcal{F}_{\mathbf{DT}}$ model. This observation forms that basis of our implementation of Timed CSP refinement checking into FDR, described in the following two sections.

5 From discrete Timed CSP to *tock*-CSP

The semantics of discrete Timed CSP maps any process to a process that communicates both the events that it is built out of and the special event *tock*. The first reaction of a CSP aficionado would be to give it a semantics directly in the discrete timed failures model. However it is also possible to give an *equivalent* semantics by translating to the *tock*-CSP language, namely mapping each Timed CSP term to a process defined using the operators of untimed CSP from this extended set of events.

These patterns can be described by a simple syntactic translation from Timed CSP to *tock*-CSP provided we introduce two special operators that take account of the special role of *tock*, and provided we make appropriate use of priority to ensure that *tock* cannot happen when τ is available. This translation is described below giving, as is traditional in semantics, a separate clause for each Timed CSP construct.

As in [27], we will define a function $\mathit{time}(\cdot)$ that maps Timed CSP syntax into *tock*-CSP. The timed analogue of STOP is TOCKS, the process that just lets time pass:

$$\begin{array}{rcl} \mathit{time}(\mathit{STOP}) & = & \mathit{TOCKS} \quad \text{where} \\ \mathit{TOCKS} & = & \mathit{tock} \rightarrow \mathit{TOCKS} \end{array}$$

CSP admits two different interpretations of the termination action ✓: in one (used in all works on CSP prior to Roscoe's 1997 book, including the main works on Timed CSP), ✓ behaves like an ordinary event and can be refused by the observer. In the other, advocated in [26, 27], ✓ can be observed, but not refused, by the environment: there ✓ is a *signal.* That distinction shows up rather clearly in Timed CSP, for in the first interpretation SKIP can wait to perform ✓:

$$\begin{array}{rcl} \mathit{time}(\mathit{SKIP}) & = & \mathit{TSKIP} \quad \text{where} \\ \mathit{TSKIP} & = & (\mathit{tock} \rightarrow \mathit{TSKIP}) \,\square\, \mathit{SKIP} \end{array}$$

whereas in the other ✓ is bound to happen immediately and therefore $\mathit{time}(\mathit{SKIP}) = \mathit{SKIP}$. The latter definition is set out in [27], but as pointed out in [26, 27], FDR follows the first interpretation of ✓ in its internal workings.[4] We have to be careful about this distinction in translating for FDR since in the world of *tock*-CSP the FDR operational semantics of parallel

[4]As pointed out in those references, in untimed CSP the semantics of P; SKIP in the first interpretation are equivalent to those of P in the second.

operators are not consistent with the definition $time(SKIP) = SKIP$ because, for example $time((WAIT(2);\ SKIP) \parallel SKIP)$ would deadlock (refusing even *tock*) immediately, rather than terminating after two *tock*s as it should. Therefore we adopt the *TSKIP* version for the translation in this paper.[5]

The $WAIT\,n$ command is just an extended *SKIP*:

$$time(WAIT\,0) = SKIP, \quad time(WAIT\,n{+}1) = tock \rightarrow time(WAIT\,n)$$

The semantics of sequential composition is very straightforward:

$$time(P;\ Q) = time(P);\ time(Q)$$

We have defined $WAIT\,t$ and sequential composition above because they are necessary to define the version of prefixing that we have implemented. We assume that there is a function et from Σ to $\mathbb{N}$ which represents the time it takes between the event a in $a \rightarrow P$ and similar constructs, and the successor process starting. With that assumption we can define

$$\begin{array}{rcl} time(a \rightarrow P) & = & \mu\, p.tock \rightarrow p \\ & & \quad \Box\ a \rightarrow (WAIT\ et(a);\ time(P)) \\ time(?x : A \rightarrow P(x)) & = & \mu\, p.tock \rightarrow p \\ & & \quad \Box\ ?x : A \rightarrow (WAIT\ et(x);\ time(P(x)) \end{array}$$

Timed CSP recursions $\mu\, p.F(P)$ translate to the restructured recursion $\mu\, p.time(F)(p)$, where $time(F)$ is the function which, when applied to the formal identifier p which is not itself changed by applying *time* (so $time(p) = p$), gives $time(F(p))$. All this says is that in applying the *time* translation to a recursive definition, all one has to is to apply the usual transformations to the constructs used in the recursion: the recursive structure itself is unchanged. Thus

$$time(\mu\, p.left?x \rightarrow right!x \rightarrow p) = \mu\, p.time(left?x \rightarrow right!x \rightarrow p)$$

Expanding this out tells us that $time(COPY)$ is precisely $TCOPY2$ if $et(a) = 1$ for all relevant events.

The constants *CHAOS* and *RUN* are problematic to translate, since their natural interpretations, namely

$$\begin{array}{rcl} RUN^t & = & tock \rightarrow RUN^t \ \Box\ ?x : \Sigma - \{tock\} \rightarrow RUN^t \\ CHAOS^t & = & TSKIP\ \sqcap \\ & & (tock \rightarrow CHAOS^t\ \Box \\ & & \ (STOP \sqcap ?x : \Sigma - \{tock\} \rightarrow CHAOS^t)) \end{array}$$

violate the principle that only finitely many events can happen in a finite amount of time: they are *Zeno* processes. Note that, unless all events are reckoned to take zero time, these are not the same as applying the operator *time* to the usual recursive definitions of

$$RUN = \Box_{a \in \Sigma}\, a \rightarrow RUN \quad \text{and} \quad CHAOS = STOP \sqcap SKIP \sqcap \Box_{a \in \Sigma}\, a \rightarrow CHAOS$$

[5] In fact, provided the standard time priority operator that we describe below is always used at the outermost syntactic level of Timed CSP, again $P;\ SKIP$ behaves identically to the signal interpretation of ✓.

In a strong sense *there are no* useful finitely nondeterministic processes which might plausibly fill their roles that *do* satisfy this principle[6].

Similarly divergence (**div**) causes problems, since (a) a process that diverges in a finite time contradicts the idea that processes perform finitely many events up to any finite time, and (b) a process that diverges over an infinite time (i.e. with infinitely many *tock*s on the way) is indistinguishable from *STOP* (i.e. the *tock* CSP process *TOCKS*) in our timed theories. You could regard **div** as a process which, by performing an infinite number of events before the first *tock*, shows invisible Zeno behaviour.

In this paper we will assume that all completely formed Timed CSP processes are constructed so that they can only perform a finite number of actions up to any finite time, though it is sometimes convenient to use processes like RUN^t and $CHAOS^t$ as parallel components on the assumption that whatever they are put in parallel with will prevent them breaching this principle. There is in fact a very simple FDR check on the resulting *tock*-CSP to prove this: $P \setminus \Sigma$ must be divergence-free.[7]

We can simply define $time(P \sqcap Q) = time(P) \sqcap time(Q)$ since there is no reason why nondeterministic choice should not be resolved immediately.

The most challenging basic operator to translate is external choice $\Box$, because *tock* is not an event that resolves $P \Box Q$. Rather, P and Q should run lockstep in time until one or other of them performs any other visible event. This requires a new operator $\Box_T$ whose SOS operational semantics can be written:

$$\frac{P \xrightarrow{\tau} P'}{P \Box_T Q \xrightarrow{\tau} P' \Box_T Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box_T Q \xrightarrow{\tau} P \Box_T Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \Box_T Q \xrightarrow{a} P'}[a \notin T] \qquad \frac{Q \xrightarrow{a} Q'}{P \Box_T Q \xrightarrow{a} Q'}[a \notin T]$$

$$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q'}{P \Box_T Q \xrightarrow{a} P' \Box_T Q'}[a \in T]$$

Here T is a set of events ($\{tock\}$ in its use in $time(P \Box Q)$) that are synchronised between P and Q rather than being allowed to resolve the choice. So

$$time(P \Box Q) \quad = \quad time(P) \Box_{\{tock\}} time(Q).$$

In fact $\Box_T$ has a *combinator operational semantics* of the sort defined in [27]. The existence of that semantics, which can be found in Chapter 15 of [27], implies that $\Box_T$ can be translated into the CSP language (involving the extra "throw" operator Θ_A) used in [27]. In fact, Schneider had anticipated this result by proposing the following translation for $P_1 \Box_{\{tock\}} P_2$ by private communication to Ouaknine as reported in [16]:

$$((P_1[\![R_1]\!] \underset{\{tock\}}{\|} P_2[\![R_2]\!]) \|_{\Sigma_1 \cup \Sigma_2} (RUN_{\Sigma_1} \Box RUN(\Sigma_2))[\![R_1^{-1} \cup R_2^{-1}]\!]$$

[6]This topic has been much discussed in the literature of Timed CSP. Schneider [32, 15] pointed out for continuous Timed CSP that the solution to defining *CHAOS* is to use infinitely long behaviours: you can then specify that only finitely many things can happen up to any finite time without expressing a uniform bound on how many. The same is undoubtedly true for discrete models and if proposing a purely mathematical model we would be following this route. However, our motivation is to make Timed CSP accessible to FDR. This would be made far more difficult if we allowed unboundedly nondeterministic basic constants.

[7]This is a version of the *timing consistency check* advocated in [26, 27].

where the injective functions R_1 and R_2 both map *tock* to itself and every other action x of P_1 and P_2 to values in disjoint sets Σ_1 and Σ_2: say x maps to $x.1$ under R_1 and $x.2$ under R_2. In fact this translation does not cope accurately[8] with P_1 or P_2 terminating (given the standard interpretation of $\|_X$) or diverging, and is therefore simpler than the one that the theory developed in [27] creates.[9] Both, however share the structure of running modified versions of P_1 and P_2 in parallel and applying constructs including renaming to the outside.

As first pointed out by Tom Gibson-Robinson, this means that in practice one cannot use this translation in conjunction with FDR in any case where a recursion reaches through the $\Box_T$ operators, such as

$$Q = a \rightarrow Q \;\Box_{\{tock\}}\; b \rightarrow Q$$

For as this process performs more and more actions, the parallel simulations of $\Box_{\{tock\}}$ add more and more layers of parallel and renaming constructs into the CSP that the FDR compiler has to evaluate: it is not clever enough to spot that terms are semantically equivalent when doing the compilation, and so the compilation fails to terminate.

We have therefore implemented $\Box_T$ directly in FDR as the operator `[+T+]`.

Exactly similar considerations apply to the interrupt operator:

$$\mathit{time}(P \bigtriangleup Q) \quad = \quad P \bigtriangleup_{\{tock\}} Q$$

where $\bigtriangleup_T$ is an operator with the following operational semantics:

$$\frac{P \xrightarrow{\tau} P'}{P \bigtriangleup_T Q \xrightarrow{\tau} P' \bigtriangleup_T Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \bigtriangleup_T Q \xrightarrow{\tau} P \bigtriangleup_T Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \bigtriangleup_T Q \xrightarrow{a} P' \bigtriangleup_T Q}[a \notin T] \qquad \frac{Q \xrightarrow{a} Q'}{P \bigtriangleup_T Q \xrightarrow{a} Q'}[a \notin T]$$

$$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q'}{P \bigtriangleup_T Q \xrightarrow{a} P' \bigtriangleup_T Q'}[a \in T]$$

In other words it behaves just like the usual interrupt operator except that it synchronises members of T between the arguments rather than letting them happen independently. So $P \bigtriangleup_{\{tock\}} Q$ says that time passes in both processes together before eventually Q performs a non-tock action and takes control.

We have implemented $\bigtriangleup_T$ in FDR as the operator `/+T+\` to enable us to handle the translation of $\mathit{time}(P \bigtriangleup Q)$.[10]

The only other CSP operators with interesting translations are parallel ones. In every case we want the processes in parallel to synchronise on *tock* in addition to the events they already

[8]It was correct in the context where he used it, since the assumption of "well timedness" meant that divergence was impossible, and he used a non-standard parallel operator in which either process could individually force termination. FDR, in common with most work on CSP, uses distributed termination in which the parallel composition only terminates when all components have done so.

[9]That, in addition to a construction similar to the above, uses the interrupt operator to turn off P_2 or P_1 at the point at which P_1 or P_2 (respectively) performs an action other than *tock*, and a harness to allow either to force the other to terminate.

[10]Technical point: at the time of writing we have only implemented $\Box_T$ and $\bigtriangleup_T$ as *low-level* operators in CSP, so FDR cannot efficiently deal as yet with these operators being applied to parallel combinations with large state spaces.

synchronise on, so for example

$$\begin{array}{rcl} time(P \mid\mid\mid Q) & = & time(P) \underset{\{tock\}}{\parallel} time(Q) \\ time(P \underset{X}{\parallel} Q) & = & time(P) \underset{X \cup \{tock\}}{\parallel} time(Q) \end{array}$$

This is simpler with the *TSKIP* model of termination than the signal one, meaning that the above translations are more straightforward than those in [27].

The rest of CSP's operators are easy to translate, though the ones for hiding, ; and $\rhd$ are dependent on the use of priority as discussed earlier.

- $time(P \setminus X) = time(P) \setminus X$.
- $time(P[\![R]\!]) = time(P)[\![R]\!]$.
- $time(P \rhd Q) = time(P) \rhd time(Q)$
- $time(P;\ Q) = time(P);\ time(Q)$
- $time(P\, \Theta_A\, Q) = time(P)\, \Theta_A\, time(Q)$

6 Implementation in FDR

Our implementation produces an identical result to the translation set out in the previous section except that the *tock* CSP is never generated directly. Instead, we have created a mode in which Timed CSP is compiled into the same sort of internal FDR objects that the translation would have been compiled into, including direct implementations of the delayed choice and timed interrupt operators described there.

We have simply added the possibility of including `Timed (et){ ... }` sections into the CSP$_M$ scripts that FDR uses. Any process defined in such a section is considered to be Timed CSP and therefore interpreted as the corresponding translation into *tock*-CSP. The parameter `et`, which we will discuss below, allows the user to define how long each action `a` takes to complete in constructs such as `a -> P`. For the time being we will assume that each such event takes one time unit (i.e. one `tock`) to complete, using:

```
OneStep(a) = 1
```

The event `tock` is not normally used inside `Timed` sections, but it (and the *tock*-CSP process `TOCKS = tock-> TOCK` which corresponds to the Timed CSP process *STOP*) are automatically made available outside them. One can mix Timed CSP and untimed (including *tock*-) CSP in the same file. Timed and untimed CSP can be mixed in the same `assert` statement or even in the same process definition. For example given the script (omitting channel declarations)

```
Timed (OneStep) {
P = a -> b -> P   -- Timed CSP
}

P' = a -> b -> P' -- untimed definition

TP1 = tock -> TP1
     [] a -> tock -> TP2
```

```
TP2 = tock -> TP2
     [] b -> tock -> TP1

assert P [FD= TP1
assert TP1 [FD= P
assert P\{tock} [T= P'
assert P' [T= P\{tock}
```

all of the assertions give answer *true*: `TP1` is simply the *tock*-CSP translation of the Timed CSP process `P` and, when we hide the event `tock` that the `Timed` section introduces into `P`, it gives the same traces as `P'`.

In any Timed CSP process with τ actions, it is necessary to use the appropriate prioritisation of τ over `tock`. FDR supplies, for scripts with Timed CSP included, a function `timed_priority` that is equivalent to `prioritise(P,{},{tock})` which is applied to any timed process that we want to interpret under maximal progress. This function gives τ priority over `tock`, with all other actions being independent of these, as discussed earlier. For example, the following script builds chains of timed, one-place buffers and compares a particular chain (with 4 members) against a specification.

```
Timed(OneStep){
COPY(ii,oo) = ii?x -> oo!x -> COPY

CS(1) = COPY(c.0,c.1)
CS(n) = (CS(n-1)[|{|c.n-1|}|]COPY(c.n-1,c.n))\{|c.n-1|}
}

Resp(n) = Resp'(n,n)

Resp'(n,0) = tock -> Resp'(n,0)
             [] (|~| x:Sigma @ x -> Resp'(n,n)

Resp'(n,m) = tock -> Resp'(n,m-1)
             [] (STOP |~| ([] x:Sigma @ x -> Resp'(n,n)))

assert Resp(4) [F= timed_priority(CS(4))
```

If one defines a Timed CSP process, such as `CS(4)` above, in which it is possible that both τ and `tock` are choices from the same state, then one *must* apply `time_priority` to it to get accurate Timed CSP semantics.

We have discovered that it is frequently helpful to define timed specifications of Timed CSP behaviour in *tock*-CSP rather than Timed CSP: `Resp(n)` above says that, from the beginning of time and from each non-*tock* event, some non-*tock* event must be offered when at least n units of time have elapsed. In this case we could have expressed the same specification elegantly in Timed CSP:

```
TResp(n) = (WAIT(n)[](
            STOP |~| [] x:Sigma @ x -> TResp(n)));
            (|~| x:Sigma @ x -> TResp(n))
```

where we assume that the built-in wait for each event is 0.

6.1 Timed Failures refinement

In practice most of the properties that we have checked of Timed CSP on FDR to date have been formulated outside Timed CSP and checked over the traces or failures models of untimed CSP. Since these models are not compositional for Timed CSP it follows that such results cannot naturally be combined in the Timed CSP language. For stand-alone analyses this may not matter. However it is clearly highly desirable to have a checkable theory of refinement with respect to which Timed CSP has the usual compositional and monotonic properties. We need to be able to check refinement in the (discrete) timed failures model, between a pair of processes each of which is written in Timed CSP.

At the time of writing this function is not implemented directly in FDR. It will be implemented in a future version, but at the expense of a little efficiency it can be calculated already using FDR's ability to check refinement over the stable refusal testing model $\mathcal{RT}$. For simplicity we assume that the process P never terminates ($\checkmark$).

For a Timed CSP process P, we can devise a *tock*-CSP context $TF[P]$ whose semantics in $\mathcal{RT}$ is in natural correspondence with P's semantics in $\mathcal{F}_{\mathbf{DT}}$. The latter differs from the $\mathcal{RT}$ semantics of P (in which *tock* is treated as a normal event) in the following way.

- The behaviours recorded in the $\mathcal{RT}$ semantics are allowed to record refusal sets at the end of a trace and before *every* event in the trace.

- The behaviours recorded in the $\mathcal{F}_{\mathbf{DT}}$ semantics are allowed to record refusal sets only at the end of a trace and before *tock*s.

However, since Timed CSP processes never refuse the event *tock*, the following equation always holds

$$\mathcal{F}_{\mathbf{DT}}(P) = \{\beta \mid \beta\hat{}\langle tock, \emptyset\rangle \in \mathcal{F}_{\mathbf{DT}}(P)\} \qquad (\dagger)$$

The process

$$TFR = (?x : \Sigma \to TFR) \rhd (tock \to TFR \,\Box\, ?x : \Sigma \to \mathbf{div})$$

has all possible traces involving elements of $\Sigma \cup \{tock\}$. It is a strict refinement of the process $RUN_{\Sigma\cup\{tock\}}$ over $\mathcal{RT}$. However the only places where its $\mathcal{RT}$ behaviours have refusals are (i) at the end of a trace (ii) immediately before a *tock* and (iii) immediately before the last member (other than *tock*) of a trace that is not itself followed by a refusal. It follows that $C[P] = TFR \underset{\Sigma\cup\{tock\}}{\|} P$ has just those behaviours of P which have these same three sorts of refusals.

The first two sorts are ones we need in the Timed Failures representation of P, but the third is not. And it turns out that $C[P]$ is (as a mapping from $\mathcal{RT}$ to itself) too discriminating to capture $\mathcal{F}_{\mathbf{DT}}$ refinement just because it retains these extra failures.

One cannot – within the healthiness conditions of $\mathcal{RT}$ – remove these failures, so the approach we have taken is to add in sufficient refusals just before the end of a trace to ensure that no distinctions are made on the basis of them.

Because of equation (†) we can safely add in any behaviours after the last *tock* a process performs without removing any distinction between Timed CSP processes. We have therefore taken the drastic approach – to concealing refusals of type (iii) – of adding in every possible behaviour not involving a *tock* after the last *tock* in any given trace. This can be achieved with the rather arcane *tock*-CSP construction

$$\begin{array}{rcl} TF[P] & = & (C[tock \to P][\![^{tock,\, tock'}/_{tock,\, tock}]\!] \\ & & \Theta_{tock'}\ CHAOS_2(\Sigma))[\![^{tock}/_{tock'}]\!] \\ \text{where} & & CHAOS_2(A) = STOP \sqcap \bigsqcap\{a \to CHAOS_2(A) \mid a \in A\} \end{array}$$

$CHAOS_2(A)$ is the least refined process over alphabet A in $\mathcal{RT}$. Every time $TF[P]$ performs *tock*, it can nondeterministically go into a state where it can do or refuse anything, but never do another *tock*.

The behaviours of $TF[P]$ that end in *tock* are precisely those of the $\mathcal{F}_{\mathbf{DT}}$ representation of $WAIT(1);\ P$, and in general $TF[P] \sqsubseteq_{RT} TF(Q)$ if and only if $P \sqsubseteq_{TF} Q$ (with P and Q being Timed CSP processes under the usual priority).

A file implementing this (including a way of handling termination) can be found amongst those supporting this paper.

7 Case studies

As we write this paper the embedding of Timed CSP in FDR is new, and there have been few case studies using it. Nevertheless those we have tried indicate good performance that scales as well as FDR does on untimed examples. Code illustrating these case studies can be downloaded via
`http://www.cs.ox.ac.uk/people/publications/personal/Bill.Roscoe.html`

Most of these files use the state space compression functions that are frequently critical to the successful use of FDR on large problems. Where (as in most cases) we need to use priority to enforce maximal progress, care is required in the use of compression. Of the functions implemented at the time of writing, only strong bisimulation and the relatively new divergence-respecting weak bisimulation described in [27] are permitted inside the priority operator. Experiments have shown that in most cases the latter, `wbisim`, is *nearly* as good as the popular combination of FDR's `diamond` (disallowed inside priority) with `sbisim` (strong bisimulation).

In several of the following example, the ability of bisimulation to eliminate symmetries proves important.

7.1 Alternating Bit and Sliding Window protocols

The Alternating Bit Protocol (ABP) is extremely well known and studied.

A Timed CSP version of ABP was given in Chapter 15 of [27]. We can use FDR to prove correctness properties of it, for example by showing that whatever errors are allowed, after hiding *tock* it trace-refines a one-place buffer *COPY*.

It is more interesting to see how it behaves under patterns of errors that are limited by time. For example, we can ascertain what rate of errors can be tolerated while still expecting the protocol to limit transmission latency and bandwidth to any given value. The protocol implementation we used is that from [27]:

```
SND = S(0)

S(s) = AC(1-s) /\ left?x -> S'(s,x,0)
S'(s,x,n) = AC(1-s) /\ S''(s,x,n)
```

```
AC(s) = d.s -> AC(s)
S''(s,x,n) = d.s -> S(1-s)
             [] (WAIT(n);a.s.x -> S'(s,x,D))

RCV = R(0)

R(s) = b!(1-s)?x -> c.(1-s) -> R(s)
       [] b!s?x -> right!x -> c!s -> R(1-s)
```

using a pair of channel processes which can lose and duplicate messages when (respectively) the events `loss` and `dup` occur. We analysed the state of affairs when all events take one time unit, except the error events, which take no time. Since this implies that at least 5 time units are necessary for communication from `SND` to `RCV` to be acknowledged, the delay `D` (which starts one unit after each send on `a`) used for retransmission was 4.

The error model we considered is one where we impose a limit k on how many errors can occur in any consecutive N units of time. (To illustrate the symmetry-eliminating power of bisimulation, this is implemented by interleaving k processes, each of which allows at most 1 error in N time units, and applying bisimulation.)

We can then obtain results such as that data transmission can be guaranteed provided there is less than one error in every 5 *tock*s.[11] The overall untimed behaviour of the alternating bit protocol is equivalent to a one-place buffer provided there are sufficiently few errors, and therefore inputs and outputs strictly alternate. Thus, a measure of latency is given by the maximum length of time between an output and the following input being enabled, or an input and the following output. These can easily be measured by FDR checks[12]: for example with up to 5 errors in every 26 the system is guaranteed to be able to perform the next input or output within 32 but not within 31.

The Sliding Window Protocol (SWP) [22] has both the sender and receiver work simultaneously on transmitting several different messages, so there is no need to wait to get one acknowledged before sending the next. It is a direct generalisation of the alternating bit protocol (which corresponds to a "window" size of 1) and uses cyclic tags from $\{0, \ldots, 2W - 1\}$ rather than just $\{0, 1\}$. The protocol is designed to keep these two windows aligned or nearly so. In our example we used a window size $W = 4$ and programmed the sender to send the unacknowledged messages it holds in rotation. Since this rotation in any case normally implies a delay in the retransmission of each individual message, we did not introduce further delay as above.

SWP tolerates a higher error rate than ABP: the limit is now one error in every two time units. For example with one error per three *tock*s, the worst case latency is 40.

As might be expected, the bandwidth achieved by SWP is greater: for example allowing 2 errors in every 15 time units, and inputting and outputting data from the system as soon as possible, gives a worst case of 5 messages transmitted in 100 time units for ABP as opposed to 12 for SWP.

These examples show that it is not only possible to verify timed implementations of systems, but also carry out quantitative analysis on them using our Timed CSP model in FDR.

[11]Specifically, we can show that if this error rate is allowed then data might never be delivered. We can test individual lower rates but cannot at present use FDR to demonstrate that *any* lower rate would work. This would require fairness capabilities not implemented at present.

[12]As illustrated in the accompanying files, this can be done either by means of a series of checks designed to identify the cut-off value, or by a single and more sophisticated single check that will give a single FDR counter-example for each value less than the latency, thereby enabling the value to be calculated in a single check

7.2 Soldiers on a bridge

Versions of the following problem – essentially a scheduling problem – can be found in [12, 30].

> A group of soldiers are on one side of a weak and narrow bridge at night, and they want to cross to the other side. To cross they need a torch, and there is only one of these. The bridge can carry at most two of them at once. What is the optimal time for all the soldiers to arrive on the other side, given that they all have different crossing times – perhaps some of them slower due to injury?

The fact that Timed CSP can create efficient and concise models is illustrated by the following complete description of a case with 22 soldiers with seven different speeds:

```
D = {4,5,7,10,12,23,30}

Count(4) = 1  -- the number of soldiers
Count(5) = 3  -- with each given time to
Count(7) = 4  -- cross the bridge
Count(10) = 2
Count(12) = 1
Count(23) = 6
Count(30) = 5

channel enterL,enterR:D
channel done

AllZero(x) = 0

Timed(AllZero){

-- The light can be picked up by one soldier,
-- or two at the same time.

LightL = enterL?d1 -> (enterL?d2 -> WAIT(max(d1,d2));LightR)
         [] enterL?d1 -> (WAIT(d1);LightR)
LightR = enterR?d1 -> (enterR?d2 -> WAIT(max(d1,d2));LightL)
         [] enterR?d1 -> (WAIT(d1);LightL)
         [] done -> LightR

-- The following light will only allow two left->right and one
-- right->left.

ALightL = enterL?d1 -> (enterL?d2 -> WAIT(max(d1,d2));ALightR)
ALightR =  enterR?d1 -> (WAIT(d1);ALightL)

max(x,y) = if x > y then x else y

-- A soldier can move to and fro, and when it is at RHS
-- can cooperate on done.

SoldierL(d) = enterL.d ->  SoldierR(d)
SoldierR(d) = done -> SoldierR(d) [] enterR.d -> SoldierL(d)
```

N	4	5	7	10	12	23	30	*tocks*	*Time*
22	1	3	4	2	1	6	5	332	$4s$
30	1	3	4	4	3	8	7	463	$23s$
36	1	4	5	5	4	9	8	546	$329s$
44	2	6	8	4	2	12	10	619	$865s$

Table 1: Statistics for some different sets of soldiers.

```
transparent sbisim

Soldiers(d,1) = SoldierL(d)
Soldiers(d,n) = sbisim(Soldiers(d,n-1) [|{done}|] SoldierL(d))

AllSoldiers = [|{done}|] d:D @ Soldiers(d,Count(d))

System = LightL [|{|enterL,enterR,done|}|] AllSoldiers
ASystem = ALightL [|{|enterL,enterR,done|}|] AllSoldiers
}

assert TOCKS [T= timed_priority(System)\{|enterL,enterR|}
assert TOCKS [T= timed_priority(ASystem)\{|enterL,enterR|}
```

Running the above check takes less than 5 seconds on a laptop and reveals that the shortest solution for this particular configuration takes 332 time units. This compares favourably for speed with the Uppaal and CSP-based timed automata implementations quoted in [12]. A wider range of results are shown in Table 1. The first column is the total number of soldiers, with the following ones giving the distribution. The last two columns give the minimum time that FDR calculates for the solution, and the elapsed time in seconds of the FDR run.

The fact that the counter-example found by FDR is as short (in time) as possible can be deduced because (a) FDR always gives a shortest possible counter-example (in events) and (b) one can demonstrate either mathematically or via FDR that temporally shortest (i.e. fewest `tocks`) solutions always (except in the case of one soldier) have two soldiers moving across the bridge in the intended direction and one backwards, repeatedly. This claim is justified in FDR by checking that the solutions to the two `assert`s above have the same number of `tock`s.

The use of `sbisim` (i.e. strong bisimulation) in the script eliminates the symmetries between different soldiers with the same speed: for example, if Bill and Jim are soldiers who both take 20 to cross the bridge, it does not matter for the purposes of solving the exercise which one `enter`s on any given occasion where they are both on the same side. This significantly improves the efficiency of our analysis.

7.3 Fischer's mutual exclusion protocol

This simple protocol has become a standard benchmark for comparing timed verification tools. In it, each of N processes might want to perform a critical section to the exclusion of all the others. They have identities $1, 2, \ldots, N$. There is a single variable v shared between the processes whose initial value is 0. When process i wants to perform a critical section it tests to see if $v = 0$ and if not waits for this state to occur. If $v = 0$ then within D units of time, v is assigned to be i. The process then waits T time units and tests if $v = i$. If so it performs the critical section before resetting v to 0. If not it goes back to the initial state.

The parameters of this protocol are the delays D and T, and the number N of processes. We can model this with the following simple Timed CSP processes:

```
Node(i) = [] j:{0..D} @ read.i.0 ->
                       (WAIT(j); write.i.i -> Node2(T,i))

Node2(x,i) = WAIT(x);(read.i?j ->
            if i==j then
                css.i -> cse.i -> write.i.0 -> Node(i)
                    else Node(i))
```

Note how this process has the option of performing `write.i.i` at any time between the `read.i.0` and `D` time units later because it can move after the `read` to different states that delay the write by these amounts. Simply giving the choice `(write.i.i -> Q) [] WAIT(D)` would not have this effect since in the complete system the `write` would be enabled and hidden as a τ meaning that maximal progress would force it to occur at once: there would actually be no possibility of it waiting beyond the moment it starts.

This sort of bounded time behaviour is arguably better expressed in *tock*-CSP without the use of priority:

```
TNode(i) = read.i.0 -> TNode1(D,i)
          [] tock -> TNode(i)

TNode1(0,i) = write.i.i -> TNode2(T,i)

TNode1(x,i) = tock -> TNode1(x-1,i)
             [] write.i.i -> TNode2(T,i)

TNode2(0,i) = read.i?j -> if i==j then CS(i)
                                 else TNode(i)
             [] tock -> TNode2(0,i)

TNode2(x,i) = tock -> TNode2(x-1,i)

CS(i) = css.i -> CS'(i)
        [] tock -> CS(i)

CS'(i) = cse.i -> write.i.0 -> TNode(i)
         [] tock -> CS'(i)
```

Note that this process expresses the bounds on when `write.i.i` occurs directly rather than relying on maximal progress. In fact it would be inappropriate to impose maximal progress on this model since it would not explore all the timings we wish it to.

Fischer's protocol works – in achieving mutual exclusion – provided $D \leq T$ and it is easy to verify this on FDR for specific values of D, T and N using either of the above versions. In these versions without the use of FDR's compressions, `D=2` and `T=3` we see typical exponential growth in `N`. On a single core of a Xeon 3.47GHz processor the times taken for the Timed CSP version above were 23 seconds and 785 seconds for $N = 6$ and 7 respectively. The times for the *tock*-CSP version were $0, 11, 138$ seconds for $N = 6, 7, 8$

However FDR's compression functions can improve this performance. We have found that the most effective technique is to keep a separate copy of v for each process, enabling each to do its reads locally, and synchronising all of them on the `write` actions. These can then be arranged in groups before compressing these, and combining the groups. It is possible to eliminate the symmetries between the different nodes in each group by renaming before the compression.

The only compressions we can use in the Timed CSP version are strong and weak bisimulation (the latter giving more compression but being slightly slower). The latter enables $N = 12, 15, 18$ to be handled in $9, 43, 552$ seconds. Not using priority in the *tock*-CSP version gives us a free reign over compression and also removes the overhead of running under priority. That enables $N = 12, 16, 20$ to be handled in $5, 54, 450$ seconds.

8 Conclusions

Just as the creation of FDR inspired a huge amount of practical work and practically-inspired theory relating to untimed CSP, we hope that this work will inspire a renaissance in the use of Timed CSP. Our brief exposure to this new tool has already brought new practical insights into the use of Timed CSP. The existence of the theory of digitisation means that properties proved in our discrete setting – including all the analysis we did in the Case Study section – are automatically proved of the continuous semantics of Timed CSP.

We have not yet looked at any industrial-scale Timed CSP case studies using FDR, but it is clear that the tool is capable of handling practical systems. One interesting potential application is analysing for the existence of covert channels based on timing in supposedly secure systems. A second paper [29] will look more carefully into this possibility.

One of the advantages of building Timed CSP into the existing FDR tool, using only small perturbations of the latter's standard models, is that most of the machinery already created to support untimed CSP will apply directly to Timed CSP. Thus, for example, it should be possible to use the integration of SAT checkers [20] and CEGAR [21] into FDR, and to take advantage of FDR's use of state-space compression functions. The only obstacle to this is that any such method needs to be consistent with the use of the `prioritise` function for Timed CSP processes.

We have seen in the Case Studies section that Timed CSP, with the alternative of *tock*-CSP, provides an extremely powerful tool for real-time analysis. Intelligent use of FDR's compressions can produce spectacular results, but this does require some skill on the part of the user.

All the novel features of FDR described in this paper, namely `Timed` sections, `prioritise` and `wbisim` will be available in the next release of FDR (2.94).

Acknowledgements

The work in this paper was supported by grants from US ONR and EPSRC. The implementation of `prioritise` was supported by Verum BV (`www.verum.com`). It has benefited from discussions with Michael Goldsmith, James Worrell, Huang Jian and Tom Gibson-Robinson. The bridge-and-soldiers case study (previously used with Uppaal) was suggested to us by Maneesh Khattri.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, *Model-checking for real-time systems*, In Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS 90), pages 414-425. IEEE Computer Society Press, 1990.

[2] H. Barringer, R. Kuiper and A. Pnueli, *A fully abstract concurrent model and its temporal logic*, in: Proc. 18th POPL (1986) 173-183.

[3] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi *UPPAAL: a tool suite for automatic verification of real-time systems*, Proc Hybrid Systems III, LNCS 1066, 1996.

[4] S.J. Creese and A.W. Roscoe, *TTP: A case study in combining induction and data independence*, Oxford University Computing Laboratory Technical Report, 1998.

[5] J.W.M. Davies, *Specification and proof in real-time CSP*, Cambridge University Press, 1993.

[6] J.W.M. Davies, D.M. Jackson, G.M. Reed, A.W. Roscoe and S.A. Schneider, *Timed CSP: theory and applications*, in 'Real time: theory in practice' (de Bakker *et al, eds*), Springer LNCS 600, 1992.

[7] C.L. Heitmeyer and R.D. Jeffords, *Formal specification and verification of real-time system requirements: a comparison study*, U.S. Naval Research Laboratory technical report, 1993.

[8] T.A. Henzinger, Z. Manna, and A. Pnueli. *Temporal proof methodologies for real-time systems*, In Proceedings of the Eighteenth Annual Symposium on Principles of Programming Languages (POPL 90), pages 353-366. ACM Press, 1990.

[9] 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.

[10] C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

[11] Huang Jian, *Extending non-interference properties to the timed world*, Oxford University D.Phil thesis, 2010.

[12] M. Khattri, J. Ouaknine and A.W. Roscoe, *Translating Timed Automata to Tock-CSP* Proceedings of IASTED SE, ACTA Press 2011.

[13] D.M. Jackson, *Local verification of reactive software systems*, Oxford University D.Phil thesis, 1992.

[14] G. Lowe and J. Ouaknine, *On Timed Models and Full Abstraction*, ENTCS 155, pp 497-519, 2006.

[15] M.W. Mislove, A.W. Roscoe and S.A. Schneider, *Fixed points without completeness*, Theoretical Computer Science **138**, 2, 1995.

[16] J. Ouaknine, *Discrete analysis of continuous behaviour in real-time concurrent systems*, Oxford University D.Phil thesis, 2001.

[17] J. Ouaknine, *Digitisation and full abstraction for dense-time model checking*, TACAS Springer LNCS, 2002.

[18] J. Ouaknine and S.A. Schneider, *Timed CSP: A Retrospective*, ENTCS 162, pp 273-276, 2006.

[19] J. Ouaknine and J.B. Worrell, *Timed CSP = Closed Timed epsilon-automata*, Nordic Journal of Computing, **10**, 2003.

[20] H. Palikareva, J.Ouaknine, and A.W. Roscoe, *Faster FDR counterexample generation using SAT-solving*, Proceedings of AVOCS 09, 2009.

[21] H. Palikareva, J. Ouaknine and A.W. Roscoe, *Integrating CEGAR with FDR*, Forthcoming 2012.

[22] K. Paliwoda and J.W. Sanders, *An incremental specification of the sliding-window protocol*, Distributed Computing, **5** (2), pp 83–94, 1991.

[23] G.M. Reed, *A uniform mathematical theory for real-time distributed computing*, Oxford University D.Phil thesis, 1988.

[24] G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, Theoretical Computer Science **58**, 249-261, 1988.

[25] A.W. Roscoe, *Model checking CSP*, in 'A classical mind: essays in honour of C.A.R. Hoare', Prentice Hall, 1994.

[26] A.W. Roscoe, *The theory and practice of concurrency* Prentice Hall, 1997.

[27] A.W. Roscoe, *Understanding concurrent systems*, Springer, 2010.

[28] A.W. Roscoe, P.J. Hopcroft and P. Armstrong, *Fairness analysis through priority*, forthcoming 2012.

[29] A.W. Roscoe and Huang Jian, *Checking noninterference in Timed CSP,* forthcoming 2012.

[30] Theo C. Ruys and Ed Brinksma *Experience with Literate Programming in the Modelling and Validation of Systems*, Proc TACAS 1998, LNCS 1384.

[31] P.Y.A. Ryan, S.A. Schneider, M.H. Goldsmith, G. Lowe and A.W. Roscoe, *The modelling and analysis of security protocols: the CSP approach*, Addison-Wesley 2001.

[32] S.A. Schneider, *Unbounded non-determinism in timed CSP*, ESPRIT SPEC project deliverable, 1991.

[33] S.A. Schneider, *Concurrent and real-time systems: the CSP approach*, Wiley, 2000.