Reasoning in *tock*-CSP with FDR

James Baxter, Pedro Ribeiro, and Ana Cavalcanti

Department of Computer Science, University of York, York, YO10 5GH, UK
{James.Baxter,Pedro.Ribeiro,Ana.Cavalcanti}@york.ac.uk

Abstract. Specifying budgets and deadlines using a process algebra like CSP requires an explicit notion of time. The *tock*-CSP encoding embeds a rich and flexible approach for modelling discrete timed behaviours with powerful tool support. It uses an event *tock*, interpreted to mark passage of time, and the model checker FDR. Analysis, however, has traditionally used the standard semantics of CSP, which is inadequate for reasoning about timed refinement. The most recent version of FDR provides tailored support for *tock*-CSP, including specific operators, but the standard semantics remains inadequate. In this paper, we characterise *tock*-CSP as a language in its own right, rich enough to model budgets and deadlines, and reason about Zeno behaviour. We present a tailored semantic model for *tock*-CSP that captures timedwise refinement. To enable use of FDR4 to check refinement in this novel model, we use an encoding of refusals via traces. Our results have been mechanised using Isabelle/HOL.

1 Introduction

In the realm of cyber-physical systems, time is a crucial concern. Such reactive systems can be modelled as interacting with their environment via named events that correspond to atomic and instantaneous interactions of interest over their lifetime. This is the view adopted by process algebras such as CCS and CSP [1], where the occurrence of events can be ordered. However, without a notion of time it is impossible to specify timed properties, like budgets and deadlines, and to reason about liveness and safety over time.

To encompass the notion of real time, several timed semantics have been proposed for CSP [2,3,4,5,6]. Early works on continuous Timed CSP include those of Reed and Roscoe [5], Davies [3] and Schneider [6]. The solid foundations of CSP with algebraic, denotational, and operational semantics gave rise to practical refinement checking, namely via model-checking with FDR [7] and other tools [8,9]. However, no such tool has, so far, emerged specifically for Timed CSP.

Instead, Roscoe has introduced *tock*-CSP, where the event *tock* is used to mark the passage of discrete time. It enables, for example, the specification of deadlines using timestops, that is, the refusal of *tock*, and the decomposition of models into timed and untimed processes, thus facilitating abstraction and modularity. Extensive use of *tock*-CSP has been reported, including, for example, in the verification of security properties [10], the design of general-purpose I/O controllers [11], the study of railways [12], the verification of distributed adaptive systems [13], and more recently, in the verification of simulations for robotics [14].

The model-checker FDR4 has specialised facilities for *tock*-CSP. For example, it provides a version of external choice ($P \,\Box_{\{tock\}}\, Q$) where the processes P and Q synchronise on *tock* until the first event that is not *tock* resolves the choice in favour of P or Q. This captures the view where P and Q have a uniform notion of time and that passage of time in itself does not resolve a choice.

FDR4 also offers a syntactic environment called a timed section[1] that translates untimed processes into *tock*-CSP. Maximal progress, where time only advances after internal behaviour has stabilised, can be enforced by prioritising internal actions τ, and $\checkmark$, that signals termination, over *tock*.

The analysis of *tock*-CSP processes, however, has typically been performed using the standard traces, failures, and failures-divergences semantics of CSP. This is also the case in FDR4. In that setting it is not possible to give a suitable semantics to processes for reasoning about timed refinement, as illustrated below.

Example 1. $R = (a \longrightarrow \mathbf{Skip} \,\Box\, b \longrightarrow \mathbf{Skip} \,\Box\, tock \longrightarrow R) \sqcap RUN(\{tock\})$
$S = (a \longrightarrow \mathbf{Skip} \,\Box\, tock \longrightarrow S) \sqcap RUN(\{tock\})$

Process R makes an internal choice ($\sqcap$). It may offer events a, b and *tock* in an external choice ($\Box$), where *tock* is followed by a recursion on R, and a and b lead to termination (**Skip**), or offer *tock* indefinitely. $RUN(\{tock\})$ is a timed deadlock, offering only the event *tock* indefinitely. Similarly, process S may also offer *tock* indefinitely, or decide to offer a and *tock* in an external choice, where a leads to termination, and *tock* to a recursion on S.

In the failures and failures-divergences models, R is refined by S, because, even though S does not offer b, refusal of b is a possible behaviour of R. In a timed setting, however, this should not be the case.

We consider the scenario where an experimenter decides to let time pass before attempting to perform a. If b is observed to be refused at time zero, and afterwards time advances by one unit, marked by *tock*, and the event a is accepted, the experimenter can conclude that the experiment is with S, not R. If R refuses b early on, it behaves as $RUN(\{tock\})$ and does not later accept a. S presents a behaviour that is not possible for R, and so it should not be the case that S refines R. However, the failures model is not rich enough to disallow such a refinement as refusal sets, which capture the events being refused, are only recorded at the end of a trace, that is, a sequence of interactions, including *tock*, and not over time as required, that is, between *tock* events.

Schneider [6] suggested the use of the refusal testing model [15,16], where refusals are recorded at the end of a trace, and also before each event. However, while in that model R is not refined by S, it is possible to make distinctions within a single time unit that are not necessarily desirable. In particular, distributivity of internal choice through external choice, for example, even when *tock* is not a possibility, does not hold in the refusal testing model as illustrated next.

Example 2. $T = (a \longrightarrow \mathbf{Stop} \,\Box\, b \longrightarrow \mathbf{Stop}) \sqcap c \longrightarrow \mathbf{Stop}$
$U = (a \longrightarrow \mathbf{Stop} \sqcap c \longrightarrow \mathbf{Stop}) \,\Box\, (b \longrightarrow \mathbf{Stop} \sqcap c \longrightarrow \mathbf{Stop})$

[1] cs.ox.ac.uk/projects/fdr/manual/cspm/definitions.html#csp-timed-section

The refusal testing semantics of T records that a happens from a state where c is refused, but not b. U, however, allows a to happen from a state where b is refused, because the right-hand side of the external choice may be internally resolved to offer c. The failures semantics of CSP, however, equates both T and U. So, refusal testing is not compatible with the view of *tock*-CSP as a language with a failures-divergences semantics within each time unit.

In this paper, we characterise *tock*-CSP as a language in its own right, with operators whose behaviour is that defined when they are used in a timed section of FDR with two crucial properties. First, events are instantaneous, so that passage of time has to be explicitly defined. Second there is maximal progress of internal events with respect to time. Our contribution is a definition of the operators of *tock*-CSP and of a novel semantic model for *tock*-CSP that allows the specification of deadlines, caters for termination, and where the refinement relation is timewise refinement. The model and operators are fully specified in Isabelle/HOL [17]. Thus another contribution is an environment for mechanical theorem proving of laws that paves the ground for the development of symbolic refinement tools for *tock*-CSP. Finally, we discuss how FDR4 can be used to check for timed refinement via an extension of a technique in [18] to deal with termination, thus providing a fully automated way for reasoning.

In Section 2 we introduce the *tock*-CSP language. The denotational model is defined in Section 3, with the semantics of the operators given in Section 4. In Section 5 we show how processes, and refinement, can be encoded in FDR4. In Section 6 we compare our semantics with existing discrete-time models employing *tock*. Finally, we conclude in Section 7 by summarizing our contributions and discussing pointers for future work.

2 *tock*-CSP

As mentioned earlier, *tock*-CSP is effectively CSP with the special event *tock* that marks the passage of time. In CSP behaviours are specified by processes using operators. In Section 2.1, we provide an overview of the operators of *tock*-CSP. In Section 2.2, we show how deadlines and Zeno behaviour can be modelled.

2.1 Overview of *tock*-CSP operators

As we have explained, the operators of *tock*-CSP are those available in FDR timed sections when events are defined not to take any time, with maximal progress implicitly enforced for each operator. The operators are listed in Table 1.

The first operator, divergence (**div**), represents a process that is in an unstable state and performs no observable events. Due to maximal progress, time can only advance when a process is in a stable state, so divergence prevents time from passing. The second operator, termination (**Skip**), represents a process that terminates immediately. A state in which termination is possible is not stable. So, as with divergence, time does not pass before termination.

Table 1. The operators of *tock*-CSP

Operator	Name
div	Divergence
Skip	Termination
Stop	Timed Deadlock
$\mathbf{Stop}_U$	Deadlock
$\mathbf{Wait}\, n$	Delay
$g \,\&\, P$	Guarding
$e \longrightarrow P$	Timed Event Prefixing
$P \sqcap Q$	Internal Choice
$P \,\Box\, Q$	Time-synchronising External Choice
$P \,;\, Q$	Sequential Composition
$P \bigtriangleup Q$	Time-synchronising Interrupt
$P \bigtriangleup_U Q$	Interrupt
$P \,[\![\, X \,]\!]\, Q$	Time-synchronising Parallel Composition
$P \setminus X$	Hiding
$P[f]$	Renaming

The third operator, timed deadlock (**Stop**), waits in a stable state, refusing all events except for *tock*. The untimed deadlock operator ($\mathbf{Stop}_U$) refuses all events and also timelocks, refusing *tock*. This is included since the ability to refuse the passage of time is an important of feature of *tock*-CSP and can be used to specify deadlines. Untimed versions of operators are marked by a subscript $_U$. Their encoding in a timed section is discussed in Section 4.

The next operator we include is the delay operator of timed CSP, $\mathbf{Wait}\, n$. This allows exactly n units of time to pass before terminating. As with **Skip**, termination happens immediately after the first n time units. In particular, note that this makes $\mathbf{Wait}\, 0$ equivalent to **Skip**.

The guarding operator ($g \,\&\, P$), takes a boolean g and a process P. It behaves as P when g is true and as **Stop** when g is false. This allows events to be conditionally offered, with events refused when the condition is false.

The timed prefixing operator ($e \longrightarrow P$) offers the event e, and then behaves as P after e has occurred. It allows time to pass while waiting for e to occur, but not between e and P, since we have instantaneous events. If e is the event *tock*, this operator allows a nondeterministic but nonzero number of units of time to pass before P starts.

We illustrate the use of timed event prefixing with the example shown below. It defines a process C, which represents a controller for a robot whose task is moving, and which quickly comes to a halt if an obstacle is detected.

Example 3.

$$C = (move \longrightarrow \mathbf{Stop}) \bigtriangleup (obs \longrightarrow ((halt \longrightarrow \mathbf{Skip}) \,\Box\, (\mathbf{Wait}\, s \,;\; \mathbf{Stop}_U)))$$

The events *move* and *halt* represent commands to a robotic platform, to initiate movement and brake; the event *obs* represents indication of an obstacle. Initially

C offers the possibility to perform *move* using the timed event prefixing operator followed by a timed deadlock. At any point this behaviour may be interrupted by the event *obs*, which we specify using the time-synchronising interrupt operator ($P \triangle Q$). This operator behaves as P, offering the events of Q while P is executing, and behaving as Q when one of the events initially offered by Q occurs. The passage of time is synchronised so that time passes in P only if it passes in Q. The occurrence of an event in P does not resolve the interrupt, allowing it to continue until P terminates or Q takes over.

As with deadlock, we also provide an untimed interrupt operator ($P \triangle_U Q$), which allows the occurrence of *tock* in Q to interrupt P. Like $\mathbf{Stop}_U$, this is useful to model deadlines. An example is given in Section 2.2, where we use it to define an interrupt that triggers after a specific number of time units.

Following *obs* in C, we have an external choice ($P \Box Q$), which offers the initial events of P and Q, behaving as the corresponding process after one of its events has occurred. External choice synchronises passage of time between P and Q, so that *tock* does not resolve the choice and time passes at the start only if both P and Q allow. The external choice in C imposes a deadline for *halt* by offering $\mathbf{Wait}\, s$ to allow time to pass for up to s time units, then behaving as $\mathbf{Stop}_U$ to prevent further time from passing while waiting for *halt* to occur (making *halt* urgent).

$\mathbf{Wait}\, s$ and $\mathbf{Stop}_U$ are composed in C using the sequential composition operator ($P \,;\, Q$), which initially behaves as P and then, when P terminates, behaves as Q. There is no time synchronisation in sequential composition since Q does not start until P finishes, so time passes in Q after time passes in P.

We also include the internal choice operator ($P \sqcap Q$), which can nondeterministically behave as either P or Q. Control over time is delegated to P or Q.

Parallel composition ($P \llbracket X \rrbracket Q$) executes P and Q in parallel, synchronising on both the events in X and *tock*. The events not in X are interleaved, occurring independently in P and Q. The parallel composition terminates when both P and Q have terminated. When one of the processes has terminated, time is still allowed to pass in the other process until it is ready to terminate.

The hiding operator ($P \setminus X$) hides the events in X, making them into internal events. Due to maximal progress, the hidden events become urgent, since internal events take priority over the passage of time. We allow the hiding of the event *tock*, so that hiding, in this case, can be used to remove time from a process, inserting an internal event wherever time could pass in P.

Finally, the renaming operator ($P[f]$) renames each of the events in P according to the function f. We do not allow renaming to or from the event *tock*. If *tock* could be renamed, then the implicit inclusion and synchronisation of *tock* events, provided by the other operators, could be applied to other events, which is not desired. Allowing renaming events to *tock* is also problematic, since it would make it possible to violate maximal progress.

This concludes the overview of the operators of *tock*-CSP. In the following section we focus on two aspects that show the expressivity of *tock*-CSP in defining deadlines and capturing Zeno behaviour.

2.2 Deadlines and Zeno behaviour

As illustrated by the previous example, *tock*-CSP allows the specification of deadlines using timestops, that is, **Stop**. For example, a common pattern in timed specifications is to impose a deadline on a process P to terminate within d time units, which can be abbreviated algebraicly $(P \blacktriangleright d)$ as follows.

Definition 1. $P \blacktriangleright d \mathrel{\widehat{=}} P \triangle (\mathbf{Wait}\ d\ ;\ \mathbf{Stop})$

Here the time-synchronising interrupt ($\triangle$) is used to ensure that P can only engage in at most d number of *tock* events. A similar, but different construction is used in Example 3 to impose a deadline on communicating an event, using the time-synchronising external choice ($\Box$) instead.

In *tock*-CSP, we can also capture Zeno behaviour, where an infinite number of events take place in a finite time. Like divergence and deadlock, this is typically undesirable behaviour. It can, however, arise from modelling errors.

Example 4. $Z = ((a \longrightarrow \mathbf{Skip}) \blacktriangleright 0)\ ;\ b \longrightarrow Z)$

Z offers to perform the event a immediately, followed by b and then the recursion. If we consider $Z \setminus \{b\}$, then the interaction with b becomes internal and urgent, and therefore an infinite sequence of a events is possible in zero time.

Just like the ability of CSP to capture divergence and deadlock is a positive feature essential to allow reasoning about (absence of) these undesirable behaviours, the ability of *tock*-CSP to express Zeno behaviour is also positive.

Next, we describe the semantic model ✓-*tock*, giving the healthiness conditions that processes are required to fulfil. In Section 4, we present the formal semantics of the operators described informally in this section.

3 Semantic model and healthiness conditions

In this section we present a new denotational model for *tock*-CSP, which we call ✓-*tock*. We define it and describe its healthiness conditions. Afterwards, in Section 4, we present the semantics of the operators of *tock*-CSP. The mechanisation of the model and operators can be found in [19].

We define the semantics of ✓-*tock* in terms of a given set Σ of events specific to the model being defined. To Σ we add two events that have a special role in the model: ✓, which signals termination of a process, and *tock*, which signals the passage of time. We refer to Σ with these special events added as $\Sigma^{\checkmark}_{tock}$.

$$\Sigma^{\checkmark}_{tock} == \Sigma \cup \{\checkmark, tock\}$$

Table 2. The healthiness conditions of ✓-*tock*

Name	Definition
TT0(P)	$P \neq \varnothing$
TT1(P)	$\rho \lesssim \sigma \land \sigma \in P \Rightarrow \rho \in P$
TT2(P)	$\rho \frown \langle ref\ X \rangle \frown \sigma \in P \land$ $Y \cap \{ e : \Sigma^{\checkmark}_{tock} \mid (e \neq tock \land \rho \frown \langle evt\ e \rangle \in P) \lor$ $(e = tock \land \rho \frown \langle ref\ X, evt\ tock \rangle \in P)$ $\} = \varnothing$ $\Rightarrow \rho \frown \langle ref\ (X \cup Y) \rangle \frown \sigma \in P$
TT3(P)	$\rho \frown \langle ref\ X, evt\ tock \rangle \frown \sigma \in P \Rightarrow tock \notin X$
TT4(P)	$\rho \frown \langle ref\ X \rangle \frown \sigma \in P \Rightarrow \rho \frown \langle ref\ (X \cup \{\checkmark\}) \rangle \frown \sigma \in P$

The semantics of each ✓-*tock* process is a set of sequences of observations, represented by the type *Obs* below. These observations may be either the occurrence of an event in $\Sigma^{\checkmark}_{tock}$ or of a refusal of some subset of $\Sigma^{\checkmark}_{tock}$.

$$Obs ::= evt \langle\!\langle \Sigma^{\checkmark}_{tock} \rangle\!\rangle \mid ref \langle\!\langle \mathbb{P}\, \Sigma^{\checkmark}_{tock} \rangle\!\rangle$$

We place constraints on the structure of the sequences of observations that form the semantics of a ✓-*tock* process, defining traces of the *TickTockTrace* type below. Constraint (1) is that a ✓ may only occur at the end of a trace, since ✓ signals termination. Constraint (2) states that any refusal set that occurs before the end of a trace must be followed by a *tock*. We thus ensure that refusals can only occur at the end of a trace and before a *tock*. Finally, constraint (3) states that every *tock* event must be preceded by a refusal. This is due to the fact that we take the presence of a refusal to indicate stability, and requiring stability before *tock* is part of how we ensure maximal progress.

$$\begin{aligned} TickTockTrace == \{ t : \operatorname{seq} Obs \mid \forall\, i : \operatorname{dom} t \bullet \\ (i < \# t \Rightarrow t\, i \neq evt\, \checkmark) \land & \quad (1) \\ (i < \# t \land t\, i \in \operatorname{ran} ref \Rightarrow t\, (i+1) = evt\, tock) \land & \quad (2) \\ (t\, i = evt\, tock \Rightarrow i > 1 \land t\, (i-1) \in \operatorname{ran} ref) & \quad (3) \\ \} \end{aligned}$$

We do not impose any requirement for a refusal to occur at the end of a trace, allowing us to represent instability after a particular sequence of events.

Each ✓-*tock* process is represented by a subset of *TickTockTrace*. However, not all such sets characterise a valid process. We define five healthiness conditions that ✓-*tock* processes satisfy, shown in Table 2. The first, TT0, states that each process, P, must have at least one trace, even if it is just the empty trace.

The second healthiness condition, TT1, is defined in terms of a prefix and subset relation, $\lesssim$, defined as shown below. We have that $t_1 \lesssim t_2$ if t_1 is obtained

from a prefix of t_2 by possibly replacing some refusals with a subset. The healthiness condition TT1 thus imposes prefix and subset closure, which is that, given any trace ρ of a healthy process, any prefix of ρ where the refusals are replaced with subsets is also a trace of that process. This corresponds to the prefix and subset closure healthiness condition of the stable failures semantics for CSP, but accounts for the fact that refusals occur before *tock* events. A consequence of TT1 is that the there must be a prefix of the trace of events before a *tock* that ends with a refusal, indicating stability.

$$
\begin{array}{|l}
\_ \lesssim \_ : \mathbb{P}(\text{seq } Obs \times \text{seq } Obs) \\
\hline
\forall\, s, t : \text{seq } Obs;\ e : \Sigma^{\checkmark}_{tock};\ X, Y : \mathbb{F}\, \Sigma^{\checkmark}_{tock} \bullet \\
\quad \langle\rangle \lesssim t \land \\
\quad (s \lesssim t \Rightarrow \langle evt\ e\rangle \frown s \lesssim \langle evt\ e\rangle \frown t) \land \\
\quad (s \lesssim t \land X \subseteq Y \Rightarrow \langle ref\ X\rangle \frown s \lesssim \langle ref\ Y\rangle \frown t)
\end{array}
$$

The offer of a *tock* event is thus incompatible with instability, as required by maximal progress.

The remaining three healthiness conditions constrain the contents of refusal sets in the traces. The third, TT2, states that any event that cannot be performed after a particular trace must be included in a refusal set of a similar trace. This is specified by stating that a set Y disjoint from the set of events that can occur can be added to the refusal set to yield another trace of the process. This is similar to the healthiness condition of the stable failures model that says events that cannot be performed must be refused but, as with TT1, TT2 handles the fact that refusals occur before *tock* throughout the trace. TT2 applies only where a refusal already occurs in a trace of a process, so it does not require the inclusion of a refusal where instability occurs.

The fourth healthiness condition, TT3, requires that a refusal before a *tock* event does not include *tock*, since *tock* cannot be both refused and performed. This does not prevent *tock* from being refused just before it can be performed, as in $a \longrightarrow \mathbf{Skip} \sqcap tock \longrightarrow \mathbf{Skip}$. This process has a trace $\langle ref\ \{a\}, evt\ tock\rangle$, where *tock* is performed after observing a refusal of a, but it also contains a trace $\langle ref\ \{tock\}\rangle$, where *tock* is refused. This process is nondeterministic but healthy, and TT3 requires that $\langle ref\ \{tock\}, evt\ tock\rangle$ is not one of its traces.

The final healthiness condition, TT4, states that anywhere a refusal occurs, the ✓ event must also be refused. This follows from the fact that termination occurs unstably, and so no refusal can be observed if ✓ is not refused. However, as with TT3, TT4 does not exclude nondeterministic processes such as $\mathbf{Skip} \sqcap a \longrightarrow \mathbf{Skip}$. This process can terminate immediately, having a trace $\langle evt\ \checkmark\rangle$, but it also has a trace $\langle ref\ \{\checkmark\}\rangle$ indicating a stable state where ✓ is refused.

Similarly to other semantic models, refinement in ✓-*tock* is subset inclusion. A process Q refines P exactly when every trace of Q is also a trace of P.

$$P \sqsubseteq Q \mathrel{\widehat{=}} Q \subseteq P$$

Next, we give the semantics of the operators of ✓-*tock*, which satisfy the healthiness conditions described here, as proved in our mechanisation [19].

4 Operator semantics

We define the semantics of a process P using a function $tttraces[\![P]\!]$, which gives the set of traces corresponding to P. In each of the sections below, we give the semantics of an operator of $\checkmark$-*tock* described in Section 2.1.

Divergence The simplest $\checkmark$-*tock* operator is **div**, which represents a divergent process and has the semantics shown below. Such a process is unstable and produces no observable behaviour, so the only trace of **div** is the empty trace. The empty trace is a trace of every process, as a consequence of TT1.

$$tttraces[\![\mathbf{div}]\!] = \{\langle\rangle\}$$

div cannot allow the passage of time, since it is never in a stable state, as indicated by the lack of a refusal in any of its traces.

Termination The process **Skip**, which terminates immediately, has the semantics shown below. Similarly to **div**, it contains no refusals, since termination is unstable due to the fact that it happens immediately without permitting the passage of time. In addition to the empty trace, **Skip** also has a trace containing the observation of a $\checkmark$ event, signalling termination.

$$tttraces[\![\mathbf{Skip}]\!] = \{\langle\rangle, \langle evt\ \checkmark\rangle\}$$

Timed Deadlock We define **Stop** using a function *tocks*, defined by the predicate below, which takes a set X and outputs sequences of *tock* events with refusals drawn from the subsets of X. We define $tocks\ X$ recursively as including the empty trace and including any traces in $tocks\ X$ prepended with a refusal (which is a subset of, or equal to, X) and a *tock* event. In addition to being used here, the *tocks* function is used in the definitions of several other operators of $\checkmark$-*tock*.

$$\begin{lparray}
\forall\, X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \\
\quad \langle\rangle \in tocks\ X \land \\
\quad (\forall\, t : tocks\ X;\ Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \mid Y \subseteq X \bullet \langle ref\ Y, evt\ tock\rangle \frown t \in tocks\ X)
\end{lparray}$$

The semantics of **Stop**, show below, is then defined to include both the traces from *tocks* themselves and traces from *tocks* with an extra refusal appended.

$$tttraces[\![\mathbf{Stop}]\!] = tocks\ \Sigma^{\checkmark} \cup \{t : tocks\ \Sigma^{\checkmark};\ X : \mathbb{P}\, \Sigma^{\checkmark} \bullet t \frown \langle ref\ X\rangle\}$$

The refusals used in this definition are taken from the subsets of $\Sigma^{\checkmark}_{tock}$ excluding the *tock* event, so that everything except *tock* is refused. For brevity in definitions, we use $\Sigma^{\checkmark}$ as an abbreviation for $\Sigma^{\checkmark}_{tock} \setminus \{tock\}$.

Deadlock We also provide $\mathbf{Stop}_U$, a version of deadlock that refuses all events, including *tock*, which has the semantics shown below. It only contains the empty trace and traces containing a single refusal, which is a subset of $\Sigma^{\checkmark}_{tock}$.

$$tttraces[\![\mathbf{Stop}_U]\!] = \{\langle\rangle\} \cup \{X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet \langle ref\ X\rangle\}$$

There are no other traces since $\mathbf{Stop}_U$ does not allow any events to occur.

Delay We define the semantics of **Wait** n as a union of three sets as shown below. The first set, (4), contains all the traces of less than or equal to n *tock* events, specified using *tocks* and with refusals drawn from $\Sigma^\checkmark$. We specify restrictions on the number of *tock* events by filtering them into a sequence containing only *tock* events using the filter operator, $\upharpoonright$, and restricting its length.

$$
\begin{aligned}
&\mathit{tttraces}[\![\mathbf{Wait}\ n]\!] = \\
&\quad \{t : \mathit{tocks}\ \Sigma^\checkmark \mid \#(t \upharpoonright \{\mathit{evt}\ \mathit{tock}\}) \leq n\} && (4)\\
&\quad \cup \{t : \mathit{tocks}\ \Sigma^\checkmark;\ X : \mathbb{P}\,\Sigma^\checkmark \mid \#(t \upharpoonright \{\mathit{evt}\ \mathit{tock}\}) < n \bullet t \frown \langle \mathit{ref}\ X\rangle\} && (5)\\
&\quad \cup \{t : \mathit{tocks}\ \Sigma^\checkmark \mid \#(t \upharpoonright \{\mathit{evt}\ \mathit{tock}\}) = n \bullet t \frown \langle \mathit{evt}\ \checkmark\rangle\} && (6)
\end{aligned}
$$

The second set, (5), contains traces of less than n *tock* events with a refusal appended, drawn from $\Sigma^\checkmark$, since we have stability before each *tock*. The final set, (6), contains traces of exactly n *tock* events followed by a $\checkmark$, since **Wait** n terminates after n time units have elapsed. As for **Skip**, we do not have a refusal after n *tock* events because termination is immediate.

Example 5. Assuming $\Sigma = \{a, b, c\}$, we sketch below the set of traces of **Wait** 2.

$$
\begin{aligned}
&\mathit{tttraces}[\![\mathbf{Wait}\ 2]\!] = \{ \\
&\quad \langle\rangle,\ \langle \mathit{ref}\ \{a, b, c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \langle \mathit{ref}\ \{a, b, c\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (7)\\
&\quad \langle \mathit{ref}\ \{a, b, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{a, b, c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (8)\\
&\quad \langle \mathit{ref}\ \{a, b, c, \checkmark\}\rangle,\ \langle \mathit{ref}\ \{a, b, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{a, b, c, \checkmark\}\rangle,\ \ldots && (9)\\
&\quad \langle \mathit{ref}\ \{a, b, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{a, b, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ \checkmark\rangle,\ \ldots && (10)\\
&\}
\end{aligned}
$$

The traces of **Wait** 2 on lines (7) and (8) are those traces of *tocks* $\Sigma^\checkmark$ that contain 2 or fewer *tock* events, contributed by set (4). Recall that, due to the healthiness condition TT1, each trace containing a refusal has a trace where that refusal is replaced with a subset. This can be seen on line (7), where $\mathit{ref}\ \{a, b, c, \checkmark\}$ in one trace is replaced with its subset $\mathit{ref}\ \{a, b, c\}$. In general, in examples, we only show the traces containing the maximal refusals, omitting the subsets.

The traces on line (9) are contributed by set (5), which contains the traces from set (4) with fewer than 2 *tock* events with a refusal appended to them that is a subset of $\Sigma^\checkmark$. The traces on lines (7), (7) and (9) represent the stability before each *tock* event, in which everything but *tock* is refused.

Finally, the trace shown on line (10) is contributed by set (6), which contains the traces from set (4) that contain exactly 2 *tock* events (i.e. the trace shown on line (9)) with a $\checkmark$ event appended. Note that there is no trace with a refusal after 2 *tock* events, since the traces with 2 *tock* events are excluded from set (5), ensuring that stability cannot be observed before $\checkmark$.

Timed Event Prefixing We define the semantics of the timed event prefixing operator, $e \longrightarrow P$, as the union of four sets, as shown below. The first two, (11)

and (12), are similar to those used to define the semantics of **Stop**, but refusals in these sets do not include the event e offered by the prefixing.

$$
\begin{aligned}
&tttraces[\![e \longrightarrow P]\!] = \\
&\quad tocks\,(\Sigma^{\checkmark} \setminus \{e\}) && (11)\\
&\quad \cup \{t : tocks\,(\Sigma^{\checkmark} \setminus \{e\});\ X : \mathbb{P}(\Sigma^{\checkmark} \setminus \{e\}) \bullet t \frown \langle ref\ X \rangle\} && (12)\\
&\quad \cup \{t : tocks\,(\Sigma^{\checkmark} \setminus \{e\});\ s : tttraces[\![P]\!] \mid e \neq tock \bullet t \frown \langle evt\ e \rangle \frown s\} && (13)\\
&\quad \cup \{t : tocks\,\Sigma^{\checkmark};\ X : \mathbb{P}\,\Sigma^{\checkmark};\ s : tttraces[\![P]\!] \mid e = tock \bullet \\
&\qquad t \frown \langle ref\ X, evt\ tock \rangle \frown s\} && (14)
\end{aligned}
$$

The last two sets used to define the semantics of $e \longrightarrow P$, (13) and (14), contain traces consisting of *tock* events followed by an occurrence of e, which is then followed by the observations of the traces of P. The first of these, (13), represents the case when e is an event other than *tock* and can simply be placed on its own between a trace from *tocks* and a trace from the semantics of P. The second, (14), represents the case where e is *tock* and inserts a refusal set before the occurrence of the event. Note that when the e is *tock*, at least one *tock* event must occur, so it included in the trace separately to the events from *tocks*.

Example 6. Assuming again $\Sigma = \{a, b, c\}$, the traces of $a \longrightarrow \mathbf{Stop}$ are below.

$$
\begin{aligned}
&tttraces[\![a \longrightarrow \mathbf{Stop}]\!] = \{\\
&\quad \langle\rangle,\ \langle ref\ \{b, c, \checkmark\}, evt\ tock\rangle,\ \ldots && (15)\\
&\quad \langle ref\ \{b, c, \checkmark\}, evt\ tock, ref\ \{b, c, \checkmark\}, evt\ tock\rangle,\ \ldots && (16)\\
&\quad \langle ref\ \{b, c, \checkmark\}\rangle,\ \langle ref\ \{b, c, \checkmark\}, evt\ tock, ref\ \{b, c, \checkmark\}\rangle,\ \ldots && (17)\\
&\quad \langle ref\ \{b, c, \checkmark\}, evt\ tock, ref\ \{b, c, \checkmark\}, evt\ tock, ref\ \{b, c, \checkmark\}\rangle,\ \ldots && (18)\\
&\quad \langle evt\ a\rangle,\ \langle ref\ \{b, c, \checkmark\}, evt\ tock, evt\ a\rangle,\ \ldots && (19)\\
&\quad \langle evt\ a, ref\ \{a, b, c, \checkmark\}\rangle,\ \langle evt\ a, ref\ \{a, b, c, \checkmark\}, evt\ tock\rangle,\ \ldots && (20)\\
&\}
\end{aligned}
$$

The traces on lines (15) and (16) are contributed by set (11). It consists of traces of *tock* events, with refusal of every event, except a and *tock*. Similarly, the traces on lines (17) and (18) are contributed by set (12). They are the traces on lines (15) and (16) with a refusal of every event except a and *tock* appended. The third set (13) contributes the traces on lines (19) and (20). These consist of traces of *tock* events followed by an occurrence of a, as can be seen on line (19), where the traces from line (15) are followed by the event a. After the occurrence of a, the traces of **Stop** are appended, as shown on line (20). Set (14) does not contribute any traces in this case, since $a \neq tock$.

Internal Choice The semantics of internal choice, $P \sqcap Q$, is simply the union of the semantics of P and Q, allowing the behaviour of either to be chosen.

$$tttraces[\![P \sqcap Q]\!] = tttraces[\![P]\!] \cup tttraces[\![Q]\!]$$

External Choice The semantics of external choice, $P \,\Box\, Q$, is shown below. It is defined in terms of a set that collects traces $r \frown p$ and $r \frown q$ from the semantics of P and Q, constrained by several conditions. The common prefix r from $tocks\ \Sigma^{\checkmark}_{tock}$ captures the synchronising behaviour of external choice by requiring the *tock* events at the start to be the same. The prefix r is required to be the longest such prefix of $r \frown p$ and $r \frown q$ by conditions (22) and (23), since all the *tock* events at the start must be synchronised.

$$
\begin{aligned}
&tttraces[\![P \,\Box\, Q]\!] = \{r : tocks\ \Sigma^{\checkmark}_{tock};\ p, q, t : TickTockTrace \mid \\
&\quad r \frown p \in tttraces[\![P]\!] \land r \frown q \in tttraces[\![Q]\!] \land && (21)\\
&\quad (\forall\, r2 : tocks\ \Sigma^{\checkmark}_{tock} \bullet r2 \text{ prefix } r \frown p \Rightarrow r2 \text{ prefix } r) \land && (22)\\
&\quad (\forall\, r2 : tocks\ \Sigma^{\checkmark}_{tock} \bullet r2 \text{ prefix } r \frown q \Rightarrow r2 \text{ prefix } r) \land && (23)\\
&\quad (\forall\, X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet p = \langle ref\ X \rangle \Rightarrow \\
&\qquad \exists\, Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet q = \langle ref\ Y \rangle \land X \setminus \{tock\} = Y \setminus \{tock\}) \land && (24)\\
&\quad (\forall\, X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet q = \langle ref\ X \rangle \Rightarrow \\
&\qquad \exists\, Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet p = \langle ref\ Y \rangle \land X \setminus \{tock\} = Y \setminus \{tock\}) \land && (25)\\
&\quad (t = r \frown p \lor t = r \frown q) \bullet t\}
\end{aligned}
$$

The refusals that occur after the initial *tock* events are intersected for events other than *tock*, since we offer the non-*tock* events of both P and Q. This is specified by conditions (24) and (25). They require that if p or q is a trace containing a single refusal then both must be such a trace, since lack of a refusal in P or Q indicates instability and so makes the external choice unstable at that point. The refusals X and Y in p and q must contain the same non-*tock* events. A refusal of *tock* can be included even if it is not matched by a refusal of *tock* in the other trace, since the *tock*-synchronising behaviour means that *tock* is refused unless both processes offer it. Traces other than single refusals must either be empty or start with a non-*tock* event, since *tock* events at the start of p and q are ruled out by conditions (22) and (23). These traces are not constrained since the occurrence of a non-*tock* event resolves the choice. We recall that *tock* is absent from refusals before *tock* events (by TT3), so this handling of *tock* in refusals does not need to be applied to r. The matching traces $r \frown p$ and $r \frown q$ are both included in the semantics of $P \,\Box\, Q$.

Example 7. In the example below, we use again $\Sigma = \{a, b, c\}$.

$$
\begin{aligned}
&tttraces[\![a \longrightarrow \mathbf{Stop} \,\Box\, b \longrightarrow c \longrightarrow \mathbf{Stop}]\!] = \{\\
&\quad \langle\rangle,\ \langle ref\ \{c, \checkmark\}, evt\ tock\rangle,\ \langle ref\ \{c, \checkmark\}, evt\ tock, ref\ \{c, \checkmark\}, evt\ tock\rangle,\ \ldots && (26)\\
&\quad \langle ref\ \{c, \checkmark\}\rangle,\ \langle ref\ \{c, \checkmark\}, evt\ tock, ref\ \{c, \checkmark\}\rangle,\ \ldots && (27)\\
&\quad \langle evt\ a\rangle,\ \langle ref\ \{c, \checkmark\}, evt\ tock, evt\ a\rangle,\ \langle evt\ a, ref\ \{a, b, c, \checkmark\}\rangle,\ \ldots && (28)\\
&\quad \langle evt\ b\rangle,\ \langle ref\ \{c, \checkmark\}, evt\ tock, evt\ b\rangle,\ \langle evt\ b, ref\ \{a, b, \checkmark\}\rangle,\ \ldots && (29)\\
&\}
\end{aligned}
$$

The traces on line (26) are those traces of initial *tock* events common to both $a \longrightarrow \mathbf{Stop}$ and $b \longrightarrow c \longrightarrow \mathbf{Stop}$. Those are all the traces of *tock* events with

refusals not including a, b and *tock*, effectively intersecting the refusals in the *tock* traces from both sides. The conditions (22) and (23) ensure that the refusals before all initial *tock* events are intersected.

For those traces of one of the processes in the choice that have a refusal after an initial trace of *tock* events, the conditions (24) and (25) ensure that they are matched by a corresponding trace of *tock* events from the other process, also ending in a refusal. In our example, both sides are event prefixes and so both have traces of *tock* events ending in refusals. These refusals are intersected, with the exception of *tock* events, yielding the traces on line (27).

Any traces beyond the initial sequence of *tock* events and a single refusal are included without restriction. In the case of our example, this means that traces from $a \longrightarrow \mathbf{Stop}$ are included as shown on line (28). Similarly, the traces on line (29) are contributed by $b \longrightarrow c \longrightarrow \mathbf{Stop}$.

Sequential Composition The semantics of sequential composition, $P \,;\, Q$, is defined as the union of two sets, as shown below. The first set includes all the traces in the semantics of P, except those that end with the event $\checkmark$. These traces represent the behaviour of P before termination.

$$
\begin{array}{l}
tttraces[\![P \,;\, Q]\!] = \\
\quad \{t : tttraces[\![P]\!] \mid \neg\, (\exists\, s : TickTockTrace \bullet t = s \frown \langle evt\,\checkmark \rangle)\} \\
\quad \cup \\
\quad \{s, t : TickTockTrace \mid \\
\qquad s \frown \langle evt\,\checkmark \rangle \in tttraces[\![P]\!] \land t \in tttraces[\![Q]\!] \bullet s \frown t\}
\end{array}
$$

The second set is formed from the traces of P ending in $\checkmark$, with the traces of Q appended to them. The $\checkmark$ event is removed, since it cannot occur in the middle of a trace. The traces in the second set represent the behaviour of the sequential composition after P has terminated.

Time-synchronising Interrupt For time-synchronising interrupt, $P \triangle Q$, we need a function to extract the *tock* events from a trace, since the *tock* events throughout P are synchronised with those at the start of Q. This is provided for by the function *filterTocks*, defined by the predicate below, which extracts the *tock* events and the refusals before them from a trace. It is defined recursively. For the empty trace, *filterTocks* results in the empty trace. When *filterTocks* is applied to a single refusal, we also get the empty trace, since such a refusal does not have a *tock* event attached to it. When applied to a trace that starts with an event other than *tock*, the result is that of applying *filterTocks* to the rest of the trace. When applied to a trace beginning with a refusal followed by a *tock* event, the refusal and *tock* event are retained, and are followed by the result of

applying *filterTocks* to the rest of the trace.

$$
\begin{array}{l}
\forall\, X : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark};\ t : TickTockTrace \bullet \\
\quad filterTocks\, \langle\rangle = \langle\rangle \land \\
\quad filterTocks\, \langle ref\ X\rangle = \langle\rangle \land \\
\quad filterTocks\, (\langle evt\ e\rangle \frown t) = filterTocks\, t \land \\
\quad filterTocks\, (\langle ref\ X, evt\ tock\rangle \frown t) = \langle ref\ X, evt\ tock\rangle \frown filterTocks\, t
\end{array}
$$

The semantics of time-synchronising interrupt are then defined, using *filterTocks*, as the union of three sets, as shown below. The first set, (30), contains the traces $p \frown \langle evt\ \checkmark\rangle$ from P that end in $\checkmark$. It is required that there is a trace q in the semantics of Q that is the result of applying *filterTocks* to p, since all *tock* events in P must be synchronised with ones in Q. Provided such a trace from Q exists, $p \frown \langle evt\ \checkmark\rangle$ is included without any modification, since time-synchronising interrupt cannot prevent P from terminating if it is ready to do so.

The second set, (31), contains the traces from P that end in a refusal X. As with the first set, there must be a corresponding trace in Q containing its *tock* events. In addition, the trace from Q is required to end in a refusal Y. The refusals Z at the end of the resulting traces are taken from subsets of the union of X and Y, which are required to be the same for all events except *tock*, since an event is only refused if it is refused by P and Q. This is similar to the requirement for an external choice, since the interrupt offers Q in choice throughout P. As with external choice, *tock* is refused if it is in X or Y, since *tock* can only happen when both P and Q can do it.

$$
\begin{array}{l}
tttraces[\![P \triangle Q]\!] = \\
\quad \{p : TickTockTrace;\ q : tttraces[\![Q]\!] \mid \\
\qquad p \frown \langle evt\ \checkmark\rangle \in tttraces[\![P]\!] \land filterTocks\, p = q \bullet p \frown \langle evt\ \checkmark\rangle\} \qquad (30) \\
\quad \cup \{p, q : TickTockTrace;\ X, Y, Z : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \mid \\
\qquad p \frown \langle ref\ X\rangle \in tttraces[\![P]\!] \land q \frown \langle ref\ Y\rangle \in tttraces[\![Q]\!] \land \\
\qquad filterTocks\, p = q \land Z \subseteq X \cup Y \land X \setminus \{tock\} = Y \setminus \{tock\} \bullet \\
\qquad p \frown \langle ref\ Z\rangle\} \qquad (31) \\
\quad \cup \{p : tttraces[\![P]\!];\ q1, q2 : TickTockTrace \mid \\
\qquad (\neg\ \exists\, r : \mathrm{seq}\, Obs \bullet p = r \frown \langle evt\ \checkmark\rangle) \land \\
\qquad (\neg\ \exists\, r : \mathrm{seq}\, Obs;\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet p = r \frown \langle ref\ X\rangle) \land \\
\qquad filterTocks\, p = q1 \land q1 \frown q2 \in tttraces[\![Q]\!] \land \\
\qquad (\neg\ \exists\, r : \mathrm{seq}\, Obs;\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \bullet q2 = \langle ref\ X\rangle \frown r) \bullet p \frown q2\} \qquad (32)
\end{array}
$$

Finally, the third set, (32), handles the traces p of P that end in neither $\checkmark$ nor a refusal. These traces need to be matched by traces from Q of the form $q1 \frown q2$, where $q1$ is the trace of *tock* events corresponding to p. The trace $q2$ thus represents the behaviour of Q after the initial sequence of *tock* events in $q1$. It is required that $q2$ does not start with a refusal, since refusals at the end

of a trace are already handled by the second set and refusals before a *tock* must be synchronised (and hence should occur in $q1$). The traces in the third set are then made up of the concatenation of p and $q2$, representing the behaviour of P before interruption followed by the behaviour of Q after interruption. Note that, since $q2$ can be the empty trace, this set can include traces just from P, provided there is a corresponding $q1$ trace to synchronise with.

Example 8.

$$
\begin{aligned}
&\mathit{tttraces}[\![a \longrightarrow \mathbf{Stop} \triangle b \longrightarrow c \longrightarrow \mathbf{Stop}]\!] = \{ \\
&\quad \langle \mathit{ref}\ \{c, \checkmark\}\rangle,\ \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{c, \checkmark\}\rangle,\ \ldots && (33)\\
&\quad \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}\rangle,\ \ldots && (34)\\
&\quad \langle\rangle,\ \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle, \langle \mathit{evt}\ a\rangle,\ \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ a\rangle,\ \ldots && (35)\\
&\quad \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (36)\\
&\quad \langle \mathit{evt}\ b\rangle,\ \langle \mathit{evt}\ b, \mathit{ref}\ \{a, b, \checkmark\}\rangle,\ \langle \mathit{evt}\ b, \mathit{ref}\ \{a, b, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (37)\\
&\quad \langle \mathit{evt}\ a, \mathit{evt}\ b\rangle,\ \langle \mathit{evt}\ a, \mathit{evt}\ b, \mathit{ref}\ \{a, b, \checkmark\}\rangle,\ \ldots && (38)\\
&\quad \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ b\rangle,\ \ldots && (39)\\
&\}
\end{aligned}
$$

For $a \longrightarrow \mathbf{Stop} \triangle b \longrightarrow c \longrightarrow \mathbf{Stop}$, there are no traces contributed by the first set, (30), since $a \longrightarrow \mathbf{Stop}$ does not have any traces ending with a $\checkmark$ event.

The second set, (31), contributes the traces shown on lines (33) and (34). These are the traces of $a \longrightarrow \mathbf{Stop}$ that end with a refusal and have a corresponding trace of *tock* events ending with a refusal in $b \longrightarrow c \longrightarrow \mathbf{Stop}$. The refusals before the *tock* events must be the same as those before the *tock* events from $a \longrightarrow \mathbf{Stop}$, effectively intersecting the refusal sets before *tock* events. The refusals at the ends of the traces from each side are required to be the same in all but *tock* event, and we take all the subsets of the union of these end refusals. In the case of our example *tock* is not refused by either side, and both sides contain all subsets (since they both satisfy TT1), so the result is effectively the intersection of these subsets. The trace $\langle \mathit{ref}\ \{c, \checkmark\}\rangle$, for example, is present in the semantics of both $a \longrightarrow \mathbf{Stop}$ and $b \longrightarrow c \longrightarrow \mathbf{Stop}$, so it is included on line (33). The empty trace occurs before the refusal on both sides, since the empty trace results from applying *filterTocks* to the empty trace.

On line (34) a refusal is included after the occurrence of a. This also comes from the set (31), since it ends with a refusal and is included in the traces of the process $a \longrightarrow \mathbf{Stop}$. The event a is refused after the occurrence of a, but the event b is not refused at that point, since $b \longrightarrow c \longrightarrow \mathbf{Stop}$ can still interrupt after a has occurred. The initial trace of *tock* events preceding the refusal in $b \longrightarrow c \longrightarrow \mathbf{Stop}$ is obtained by filtering the *tock* events from the trace from $a \longrightarrow \mathbf{Stop}$. The trace $\langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}\rangle$ thus comes from the trace $\langle \mathit{ref}\ \{a, c, \checkmark\}\rangle$ of $b \longrightarrow c \longrightarrow \mathbf{Stop}$, where an empty trace of *tock* events precedes the refusal.

The traces contributed by the third set, (32), are those on lines (35), to (39). Those on lines (35) and (36) are from $a \longrightarrow \mathbf{Stop}$ and do not end in a $\checkmark$ or

a refusal. As with the traces ending in a refusal (from set (31)), their *tock* events and corresponding refusals must be matched by a trace of *tock* events from $b \longrightarrow c \longrightarrow \mathbf{Stop}$. The filtering of *tock* events means this applies equally to *tock* events before a (line (35)) and after a (line (34)). The result is that b is removed from each refusal observed before a *tock*, since it is offered throughout $a \longrightarrow \mathbf{Stop}$, as the initial event of $b \longrightarrow c \longrightarrow \mathbf{Stop}$.

The traces shown on lines (37), (38) and (39) consist of traces from $a \longrightarrow \mathbf{Stop}$ that do not end in ✓ or a refusal, followed by traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that do not start with a refusal. The traces from $a \longrightarrow \mathbf{Stop}$ are matched by traces of *tock* events from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ as before, but the traces after the b event are included as-is. In particular, this means that traces ending a refusal, such as $\langle evt\ b, ref\ \{a, b, \checkmark\}\rangle$, are included by set (32) since the refusals come after b.

Interrupt For the untimed version of interrupt, $P \triangle_U Q$, we do not need to match the *tock* events in P with those in Q. However, we do need to ensure that the refusals recorded before *tock* events in P are consistent with the initial events offered by Q, since an event can be refused only if it is refused by P and Q.

We thus define a function, *intersectRefusalTrace*, to intersect each of the refusals in a trace with a given refusal. This function is defined by the predicate shown below. As with *filterTocks*, *intersectRefusalTrace* is defined recursively. It takes as its first argument a refusal X. When its second argument is the empty trace, the result is also the empty trace. When the second argument begins with an event e, the result is that of applying *intersectRefusalTrace* to the rest of the trace, with e prepended. Finally, when the second argument begins with a refusal Y, the result is that of applying *intersectRefusalTrace* to the rest of the trace, with the intersection of X and Y prepended.

$$
\begin{array}{l}
\forall\, e : \Sigma^{\checkmark}_{tock};\ X, Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ s : \mathrm{seq}\ Obs \bullet \\
\quad intersectRefusalTrace\ X\ \langle\rangle = \langle\rangle \land \\
\quad intersectRefusalTrace\ X\ (\langle evt\ e\rangle \frown s) = \\
\qquad \langle evt\ e\rangle \frown intersectRefusalTrace\ X\ s \land \\
\quad intersectRefusalTrace\ X\ (\langle ref\ Y\rangle \frown s) = \\
\qquad \langle ref\ (X \cap Y)\rangle \frown intersectRefusalTrace\ X\ s
\end{array}
$$

We also define a boolean function *containsRefusal*, which takes a trace as its only argument and determines whether that trace contains a refusal. This function is used to determine when the intersection of refusals needs to be applied. We omit its definition here as it is relatively simple.

The semantics of interrupt is then as shown below. It is defined as the union of five sets, each accounting for a different form of trace in the semantics of P.

$$
\begin{array}{l}
tttraces[\![P \triangle_U Q]\!] = \\
\quad \{p : TickTockTrace;\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \mid \\
\qquad p \frown \langle evt\ \checkmark\rangle \in tttraces[\![P]\!] \land containsRefusal\ p \land
\end{array}
$$

$$
\begin{aligned}
&\langle ref\ X\rangle \in tttraces[\![Q]\!] \bullet \\
&intersectRefusalTrace\ X\ (p \frown \langle evt\ \checkmark\rangle)\} && (40)\\
\cup\ &\{p : TickTockTrace \mid \\
&p \frown \langle evt\ \checkmark\rangle \in tttraces[\![P]\!] \land \neg\ containsRefusal\ p \bullet \\
&p \frown \langle evt\ \checkmark\rangle\} && (41)\\
\cup\ &\{p, q : TickTockTrace;\ X, Y : \mathbb{P}\ \Sigma^{\checkmark}_{tock} \mid \\
&p \frown \langle ref\ X\rangle \in tttraces[\![P]\!] \land \langle ref\ Y\rangle \frown q \in tttraces[\![Q]\!] \bullet \\
&intersectRefusalTrace\ Y\ (p \frown \langle ref\ X\rangle) \frown q\} && (42)\\
\cup\ &\{p : tttraces[\![P]\!];\ q : tttraces[\![Q]\!];\ X : \mathbb{P}\ \Sigma^{\checkmark}_{tock} \mid \\
&(\neg\ \exists\, r : TickTockTrace \bullet p = r \frown \langle evt\ \checkmark\rangle) \land \\
&(\neg\ \exists\, r : TickTockTrace;\ Y : \mathbb{P}\ \Sigma^{\checkmark}_{tock} \bullet p = r \frown \langle ref\ Y\rangle) \land \\
&containsRefusal\ p \land \langle ref\ X\rangle \in tttraces[\![Q]\!] \land \\
&(\neg\ \exists\, r : TickTockTrace;\ Y : \mathbb{P}\ \Sigma^{\checkmark}_{tock} \bullet q = \langle ref\ Y\rangle \frown r) \bullet \\
&intersectRefusalTrace\ X\ p \frown q\} && (43)\\
\cup\ &\{p : tttraces[\![P]\!];\ q : tttraces[\![Q]\!] \mid \\
&(\neg\ \exists\, r : TickTockTrace \bullet p = r \frown \langle evt\ \checkmark\rangle) \land \\
&\neg\ containsRefusal\ p \land \\
&(\neg\ \exists\, r : TickTockTrace;\ Y : \mathbb{P}\ \Sigma^{\checkmark}_{tock} \bullet q = \langle ref\ Y\rangle \frown r) \bullet \\
&p \frown q\} && (44)
\end{aligned}
$$

The first set, (40), includes traces from P that end with ✓ and contain a refusal. For these traces to be included, there must be a refusal X in Q. This is due to the fact that a refusal indicates stability and, since the initial events of Q are offered at each point in P, there must be stability at the start of Q for stability to be observed. The refusals from P are intersected with X.

The second set, (41), includes the traces from P that end with ✓ and do not contain a refusal. These traces do not require a corresponding refusal in Q and are included as-is without intersection of refusals.

The third set, (42), handles traces in P that end with a refusal. For such traces there must be a corresponding trace in Q that starts with a refusal, which we append to the traces from P, intersecting their refusals. The corresponding refusal in P is required when appending such traces, since *tock* may only happen from a stable state, so instability in P prevents a *tock* in Q from interrupting. As in the first set, the refusal from the trace in Q is intersected with all refusals throughout the trace from P, using the function *intersectRefusalTrace*. The rest of the trace in Q is then appended to the intersected trace from P.

The fourth set, (43), handles the traces p in P that end with neither a ✓ nor a refusal, but do contain a refusal somewhere. For such traces, we require that there is a trace from Q that just contains a single refusal X, which is intersected with the refusals in p, as in the first set. This indicates that Q is stable, and so the refusals of p can be kept. In the set, we include the result of appending a

trace q from Q to the intersected p. This trace q must not start with a refusal, as appending such traces from Q is already handled by the third set.

Finally, the fifth set, (44), handles the traces p in P that do not end in $\checkmark$ and do not contain a refusal. Such traces do not require a corresponding refusal in Q because no stability has been observed in P. As with set (43), we allow for a trace q from Q that does not start with a refusal to be appended to p.

Example 9.

$$
\begin{aligned}
&\mathit{tttraces}[\![a \longrightarrow \mathbf{Stop} \mathbin{\triangle_U} b \longrightarrow c \longrightarrow \mathbf{Stop}]\!] = \{ \\
&\quad \langle \mathit{ref}\ \{c, \checkmark\}\rangle,\ \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{c, \checkmark\}\rangle,\ \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}\rangle,\ \ldots && (45)\\
&\quad \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (46)\\
&\quad \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{a, c, \checkmark\}\rangle,\ \ldots && (47)\\
&\quad \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ a\rangle,\ \ldots && (48)\\
&\quad \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (49)\\
&\quad \langle \mathit{evt}\ a, \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ b\rangle,\ \ldots && (50)\\
&\quad \langle\rangle,\ \langle \mathit{evt}\ a\rangle, && (51)\\
&\quad \langle \mathit{evt}\ b\rangle,\ \langle \mathit{evt}\ b, \mathit{ref}\ \{a, b, \checkmark\}\rangle,\ \langle \mathit{evt}\ b, \mathit{ref}\ \{a, b, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle,\ \ldots && (52)\\
&\quad \langle \mathit{evt}\ a, \mathit{evt}\ b\rangle,\ \langle \mathit{evt}\ a, \mathit{evt}\ b, \mathit{ref}\ \{a, b, \checkmark\}\rangle,\ \ldots && (53)\\
&\}
\end{aligned}
$$

For $a \longrightarrow \mathbf{Stop} \mathbin{\triangle_U} b \longrightarrow c \longrightarrow \mathbf{Stop}$, as with $a \longrightarrow \mathbf{Stop} \mathbin{\triangle} b \longrightarrow c \longrightarrow \mathbf{Stop}$, there are no traces in $a \longrightarrow \mathbf{Stop}$ that end with a $\checkmark$ event, so the first two sets, (40) and (41), do not contribute any traces. The third set, (42), contributes the traces shown on lines (45), (46) and (47). These consist of the combination of traces from $a \longrightarrow \mathbf{Stop}$ that end with a refusal and traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that start with a refusal. The refusal at the start of the trace from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ is intersected with all refusals in the trace from $a \longrightarrow \mathbf{Stop}$. Since $\langle \mathit{ref}\ \{a, c\checkmark\}\rangle$ is a trace of $b \longrightarrow c \longrightarrow \mathbf{Stop}$, the traces of $a \longrightarrow \mathbf{Stop}$ that end in a refusal are included, with b and *tock* excluded from all refusals. This yields the traces shown on line (45), which are the same as those shown on line (33) for Example 8.

The trace shown on line (46) derives from the traces $\langle \mathit{evt}\ a, \mathit{ref}\ \{a, b, c, \checkmark\}\rangle$ and $\langle \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}\rangle$. It is the same as the trace shown on line (36) for Example 8, but is included for a different reason, since the *tock* event comes from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ rather than $a \longrightarrow \mathbf{Stop}$. Indeed, the same trace is also included again in the semantics of $a \longrightarrow \mathbf{Stop} \mathbin{\triangle_U} b \longrightarrow c \longrightarrow \mathbf{Stop}$ (line (49)), for a different reason, as we explain below.

However, the trace shown on line (47) is not included in the semantics of $a \longrightarrow \mathbf{Stop} \mathbin{\triangle} b \longrightarrow c \longrightarrow \mathbf{Stop}$, but is unique to the untimed version. It follows from combining the trace $\langle \mathit{ref}\ \{b, c, \checkmark\}\rangle$ from $a \longrightarrow \mathbf{Stop}$ with the trace $\langle \mathit{ref}\ \{a, c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{ref}\ \{a, c, \checkmark\}\rangle$ from $b \longrightarrow c \longrightarrow \mathbf{Stop}$. As we discuss below, $a \longrightarrow \mathbf{Stop} \mathbin{\triangle_U} b \longrightarrow c \longrightarrow \mathbf{Stop}$ also has a trace $\langle \mathit{ref}\ \{c, \checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ a\rangle$ (shown

on line (48)). This means the process can both perform and refuse a at the same point, making it nondeterministic. This is due to the fact that both processes accept *tock*, and a *tock* from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ can interrupt $a \longrightarrow \mathbf{Stop}$, so occurrence of a *tock* can be from either process. Since *tock* events are synchronised in time-synchronising interrupt, it does not show this behaviour.

The traces on lines (48), (49) and (50) are contributed by set (43). These traces consist of combinations of traces from $a \longrightarrow \mathbf{Stop}$ that contain a refusal, but do not end in a refusal or a ✓, and traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that do not start with a refusal. These require stability at the start of $b \longrightarrow c \longrightarrow \mathbf{Stop}$, which is recorded in $\langle ref\ \{a, c, ✓\}\rangle$. The refusals in each of the traces from $a \longrightarrow \mathbf{Stop}$ are intersected with this refusal, meaning that b and *tock* are excluded. Taking the empty trace as the trace from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ yields the traces on lines (48) and (49). One of these, shown on line (49), is a trace shown previously on line (46). Here, *tock* comes from $a \longrightarrow \mathbf{Stop}$, whereas previously it came from $b \longrightarrow c \longrightarrow \mathbf{Stop}$.

Traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ beginning with b can be appended instead of the empty trace, since they are the only other traces in $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that do not start with a refusal. This yields the trace on line (50), where b can happen after the trace from line (49). As with the trace from line (46) this trace is also included by set (42), with b following the *tock* event in $b \longrightarrow c \longrightarrow \mathbf{Stop}$.

The traces on lines (51), (52) and (53) come from set (44). These are similar to those from set (43), but using traces from $a \longrightarrow \mathbf{Stop}$ that do not include refusals. The traces on line (51) are formed by appending the empty trace from the process $b \longrightarrow c \longrightarrow \mathbf{Stop}$ to these traces, and are, in fact, the only such traces from $a \longrightarrow \mathbf{Stop}$ that do not include refusals.

By appending traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that start with an occurrence of b to the empty trace from $a \longrightarrow \mathbf{Stop}$, we obtain the traces on line (52). These represent the traces where a b event interrupts at the start of the process. Similarly, the traces on line (53), consist of appending traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that start with an occurrence of b to the trace $\langle evt\ a\rangle$. These represent a b event interrupting after the occurrence of an a event.

Parallel Composition The semantics of parallel composition, $P \llbracket A \rrbracket Q$, is defined as shown below, by merging each pair of traces from the semantics of P and Q using a trace merge function $p \llbracket A \rrbracket^T q$. The trace merge function describes the set of traces of the parallel composition generated by each pair of traces and the semantics of parallel composition is given by the union of the results. Having the output of the function be a set allows for different interleavings of events to be enumerated and for subset closure to be ensured.

$$tttraces[\![P \llbracket A \rrbracket Q]\!] = \bigcup\{p : tttraces[\![P]\!];\ q : tttraces[\![Q]\!] \bullet p \llbracket A \rrbracket^T q\}$$

The predicate describing the trace merge function is shown in Figure 1. This function is defined recursively, considering each possible case for a well-formed trace: the empty trace, a trace with a single refusal, a trace with a single ✓ event, a trace starting with an event in Σ, and a trace starting with a refusal

$$
\begin{array}{l}
\forall\, A : \mathbb{P}\,\Sigma;\ X, Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ e, f : \Sigma;\ t, s : TickTockTrace \bullet \\
\quad \langle\rangle \,[\![\, A \,]\!]^T\, \langle\rangle = \{\langle\rangle\} \wedge \qquad (54) \\
\quad \langle\rangle \,[\![\, A \,]\!]^T\, \langle ref\ X\rangle = \{\langle\rangle\} \wedge \qquad (55) \\
\quad \langle\rangle \,[\![\, A \,]\!]^T\, \langle evt\ \checkmark\rangle = \{\langle\rangle\} \wedge \qquad (56) \\
\quad (e \notin A \Rightarrow \langle\rangle \,[\![\, A \,]\!]^T\, (\langle evt\ e\rangle \frown t) = \{s : \langle\rangle \,[\![\, A \,]\!]^T\, t \bullet \langle evt\ e\rangle \frown s\}) \wedge \qquad (57) \\
\quad (e \in A \Rightarrow \langle\rangle \,[\![\, A \,]\!]^T\, (\langle evt\ e\rangle \frown t) = \{\langle\rangle\}) \wedge \qquad (58) \\
\quad \langle\rangle \,[\![\, A \,]\!]^T\, (\langle ref\ X, evt\ tock\rangle \frown t) = \{\langle\rangle\} \wedge \qquad (59) \\
\\
\quad \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, \langle ref\ Y\rangle = \\
\qquad \{Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \mid Z \subseteq X \cup Y \wedge \\
\qquad\quad X \setminus (A \cup \{\checkmark, tock\}) = Y \setminus (A \cup \{\checkmark, tock\}) \bullet \langle ref\ Z\rangle\} \wedge \qquad (60) \\
\quad \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, \langle evt\ \checkmark\rangle = \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, \langle ref\ \Sigma\rangle \wedge \qquad (61) \\
\quad (e \notin A \Rightarrow \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, (\langle evt\ e\rangle \frown t) = \\
\qquad \{s : TickTockTrace \mid s \in \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, t \bullet \langle evt\ e\rangle \frown s\}) \wedge \qquad (62) \\
\quad (e \in A \Rightarrow \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, (\langle evt\ e\rangle \frown t) = \{\langle\rangle\}) \wedge \qquad (63) \\
\quad \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown t) = \{\langle\rangle\} \wedge \qquad (64) \\
\\
\quad \langle evt\ \checkmark\rangle \,[\![\, A \,]\!]^T\, \langle evt\ \checkmark\rangle = \{\langle evt\ \checkmark\rangle\} \wedge \qquad (65) \\
\quad (e \notin A \Rightarrow \langle evt\ \checkmark\rangle \,[\![\, A \,]\!]^T\, (\langle evt\ e\rangle \frown t) = \\
\qquad \{s : TickTockTrace \mid s \in \langle evt\ \checkmark\rangle \,[\![\, A \,]\!]^T\, t \bullet \langle evt\ e\rangle \frown s\}) \wedge \qquad (66) \\
\quad (e \in A \Rightarrow \langle evt\ \checkmark\rangle \,[\![\, A \,]\!]^T\, (\langle evt\ e\rangle \frown t) = \{\langle\rangle\}) \wedge \qquad (67) \\
\quad \langle evt\ \checkmark\rangle \,[\![\, A \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown t) = \\
\qquad \{Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ s : TickTockTrace \mid \langle ref\ Z\rangle \in \langle ref\ \Sigma\rangle \,[\![\, A \,]\!]^T\, \langle ref\ Y\rangle \wedge \\
\qquad\quad s \in \langle evt\ \checkmark\rangle \,[\![\, A \,]\!]^T\, t \bullet \langle ref\ Z, evt\ tock\rangle \frown s\} \wedge \qquad (68) \\
\\
\quad (e \notin A \wedge f \notin A \Rightarrow (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle evt\ f\rangle \frown s) = \\
\qquad \{r : TickTockTrace \mid r \in t \,[\![\, A \,]\!]^T\, (\langle evt\ f\rangle \frown s) \bullet \langle evt\ e\rangle \frown r\} \\
\qquad \cup \{r : TickTockTrace \mid r \in (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, s \bullet \langle evt\ f\rangle \frown r\}) \wedge \qquad (69) \\
\quad (e \notin A \wedge f \in A \Rightarrow (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle evt\ f\rangle \frown s) = \\
\qquad \{r : TickTockTrace \mid r \in t \,[\![\, A \,]\!]^T\, (\langle evt\ f\rangle \frown s) \bullet \langle evt\ e\rangle \frown r\}) \wedge \qquad (70) \\
\quad (e \in A \wedge f \in A \wedge e = f \Rightarrow (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle evt\ f\rangle \frown s) = \\
\qquad \{r : TickTockTrace \mid r \in t \,[\![\, A \,]\!]^T\, s \bullet \langle evt\ e\rangle \frown r\}) \wedge \qquad (71) \\
\quad (e \in A \wedge f \in A \wedge e \neq f \Rightarrow (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle evt\ f\rangle \frown s) = \{\langle\rangle\}) \wedge \qquad (72) \\
\quad (e \notin A \Rightarrow (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown s) = \\
\qquad \{r : TickTockTrace \mid r \in t \,[\![\, A \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown s) \bullet \\
\qquad \langle evt\ e\rangle \frown r\}) \wedge \qquad (73) \\
\quad (e \in A \Rightarrow (\langle evt\ e\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown s) = \{\langle\rangle\}) \wedge \qquad (74) \\
\\
\quad (\langle ref\ X, evt\ tock\rangle \frown t) \,[\![\, A \,]\!]^T\, (\langle ref\ Y, evt\ tock\rangle \frown s) = \\
\qquad \{Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ r : TickTockTrace \mid \\
\qquad\quad \langle ref\ Z\rangle \in \langle ref\ X\rangle \,[\![\, A \,]\!]^T\, \langle ref\ Y\rangle \wedge r \in t \,[\![\, A \,]\!]^T\, s \bullet \\
\qquad\quad \langle ref\ Z, evt\ tock\rangle \frown r\} \wedge \qquad (75) \\
\\
\quad s \,[\![\, A \,]\!]^T\, t = t \,[\![\, A \,]\!]^T\, s \qquad (76)
\end{array}
$$

Fig. 1. Definition of the parallel trace merge function

followed by a *tock* event. The trace merge function is defined to be commutative, so we only consider one ordering of each of the possible cases. In addition to the traces on each side, the trace merge function also takes a synchronisation set A, so that it can be determined which events require synchronisation.

Equations (54) to (59) define the cases in which one trace is empty. If both traces are empty (equation (54)), then the result is a set of traces containing only the empty trace, since there are no further observations to be merged from either of the traces. Similarly, when the empty trace is merged with a single refusal (equation (55)) the empty trace is also the only resulting trace, since the empty trace gives no guarantee of stability to allow the inclusion of a refusal.

If a trace starts with ✓, an event in the synchronisation set A, or a *tock* event (with its preceding refusal), then its initial event requires synchronisation. Since the empty trace cannot provide that synchronisation, the set containing the empty trace is also the result when it is merged with such traces (equations (56), (58) and (59)). When a trace starts with an event e that is not in A (i.e. that does not require synchronisation), then merging it with the empty trace yields all the traces formed by preprending e to the traces resulting from merging the empty trace with the rest of the trace (equation (57)). This ensures that events that do not require synchronisation can keep being performed until an event that requires synchronisation or the empty trace is encountered.

Example 10. When the empty trace is merged with the traces of $b \longrightarrow c \longrightarrow \mathbf{Stop}$ using the synchronisation set $\{c\}$, the results are as shown below.

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle\rangle = \{\langle\rangle\} \tag{77}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b\rangle = \{\langle evt\ b\rangle\} \tag{78}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle ref\ \{a, c, ✓\}\rangle = \{\langle\rangle\} \tag{79}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle ref\ \{a, c, ✓\}, evt\ tock\rangle = \{\langle\rangle\} \tag{80}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle ref\ \{a, c, ✓\}, evt\ tock, evt\ b\rangle = \{\langle\rangle\} \tag{81}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b, ref\ \{a, b, ✓\}\rangle = \{\langle evt\ b\rangle\} \tag{82}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b, ref\ \{a, b, ✓\}, evt\ tock\rangle = \{\langle evt\ b\rangle\} \tag{83}$$

$$\langle\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b, evt\ c\rangle = \{\langle evt\ b\rangle\} \tag{84}$$

The case with two empty traces (equation (77)) is a straightforward application of equation (54). In equation (78), the b event does not require synchronisation and so is included before the empty trace that comes from merging the rest of the trace (which is empty). Refusals and *tock* events merged with the empty trace result in the empty trace, since they are not matched, by corresponding observations, as defined in equations (79) and (80). Events after *tock* events are also removed by this, as specified equation (81). When the trace from the right-hand-side begins with a b event, as in equations (82), (83) and (84), the result is prepending b to the result of merging the rest of the trace. The rest of the trace is empty, since there is still nothing to match a refusal or *tock* event, and c requires synchronisation (since it is in the synchronisation set $\{c\}$).

Equations (60) to (64) define the cases where a trace has a single refusal (and the other is not empty). When both traces are refusals (equation (60)), $\langle ref\ X\rangle$ and $\langle ref\ Y\rangle$, the resulting set contains singleton traces containing refusals drawn from subsets of the union of X and Y. The refusal sets X and Y are required to be the same, except in $\checkmark$, *tock* and the events from A. This is because these events do not require synchronisation, and are refused only if they are refused in both traces.

When a singleton trace $\langle ref\ X\rangle$ is merged with a singleton trace containing just the $\checkmark$ event (equation (61)), the result is the same as that of merging *ref* X with a refusal of Σ. This is due to the fact that the presence of a refusal implies that $\checkmark$ is refused (by TT4). Therefore, the parallelism cannot terminate, but the process that is ready to terminate refuses every other event, with the exception of *tock*, since it does not block the passage of time.

When the other trace begins with a non-$\checkmark$ non-*tock* event (equations (62) and (63)), the result is similar to the corresponding case for the empty trace (equations (57) and (58)). Similarly, merging a refusal with a *tock* event (equation (64)) results in the set containing the empty trace, since *tock* also requires synchronisation. We could include a refusal formed by merging the refusal on its own with the refusal before the *tock* event, but that is already handled by prefix closure (equation (60)).

Example 11. Merging the traces $\langle ref\ \{b, c, \checkmark\}\rangle$ and $\langle ref\ \{c, \checkmark\}\rangle$ from the process $a \longrightarrow \mathbf{Stop}$ with traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$, again taking $\{c\}$ as the synchronisation set, yields the traces shown below.

$$\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle\rangle = \{\langle\rangle\} \tag{85}$$

$$\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b\rangle = \{\langle evt\ b\rangle\} \tag{86}$$

$$\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle ref\ \{a, c, \checkmark\}\rangle = \{\} \tag{87}$$

$$\langle ref\ \{c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle ref\ \{c, \checkmark\}\rangle = \begin{array}{l}\{\langle ref\ \{c, \checkmark\}\rangle,\\ \quad \langle ref\ \{c\}\rangle, \ldots\}\end{array} \tag{88}$$

$$\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle ref\ \{a, c, \checkmark\}, evt\ tock\rangle = \{\langle\rangle\} \tag{89}$$

$$\begin{array}{r}\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\\ \langle ref\ \{a, c, \checkmark\}, evt\ tock, evt\ b\rangle\end{array} = \{\langle\rangle\} \tag{90}$$

$$\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b, ref\ \{a, b, \checkmark\}\rangle = \{\} \tag{91}$$

$$\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\, \langle evt\ b, ref\ \{b, \checkmark\}\rangle = \begin{array}{l}\{\langle evt\ b, ref\ \{b, c, \checkmark\}\rangle,\\ \quad \langle evt\ b, ref\ \{b, c\}\rangle, \ldots\}\end{array} \tag{92}$$

$$\begin{array}{r}\langle ref\ \{b, c, \checkmark\}\rangle \,[\![\, \{c\} \,]\!]^T\\ \langle evt\ b, ref\ \{a, c, \checkmark\}, evt\ tock\rangle\end{array} = \{\langle evt\ b\rangle\} \tag{93}$$

Merging a refusal with an empty trace (equation (85)) yields just the empty trace, as in equation (79). When the refusal is merged with an event not requiring synchronisation, as in equation (86), the event is included at the start and the rest of the trace is empty, since the refusal has no corresponding refusal.

When both sides are traces with a single refusal, the refusals must agree in events not requiring synchronisation. Thus, in equation (87), there are no

resulting traces, since the refusals $\{b, c, \checkmark\}$ and $\{a, c, \checkmark\}$ do not agree in the events a and b. However, the subset refusal $\{c, \checkmark\}$ is present in both $a \longrightarrow$ **Stop** and $b \longrightarrow c \longrightarrow$ **Stop**, and the result is all the subsets of it, as shown in equation (88).

Merging a refusal with a trace starting with a *tock* event yields only the empty trace, as in equations (89) and (90).

When a refusal is merged with a trace consisting of b followed by a refusal, the event b is included first and the refusals are then merged as in equations (87) and (88). Thus, in equation (91), there are no traces, since $\{b, c, \checkmark\}$ and $\{a, b, \checkmark\}$ do not agree in the non-synchronised event a. The subset refusals $\{b, c, \checkmark\}$ and $\{b, \checkmark\}$ do agree and so the result, in equation (92) is all the traces with an occurrence of b followed by a refusal that is a subset of $\{b, c, \checkmark\}$. Note that the refusals do not have to agree in the event c, since it requires synchronisation. The union of the refusals from each trace is taken, so that c is included in the output refusals. A *tock* event after b does not match the refusal, so the result in that case is just the trace containing b, as shown in equation (93).

Equations (65) to (68) are the remaining cases for traces containing a single $\checkmark$ event. This $\checkmark$ event requires synchronisation, since processes in parallel composition must terminate together. Thus, in the case where both sides have a trace containing a single $\checkmark$ event (equation (65)), the result is the set containing a trace with a single $\checkmark$ event.

When the trace with a $\checkmark$ event is merged with a trace starting with non-$\checkmark$ non-*tock* event e (equations (66) and (67)), the $\checkmark$ event cannot occur because it does not have an event to synchronise with, but the result depends on whether e requires synchronisation.

When a $\checkmark$ event is merged with a trace starting with a *tock* event, with its refusal Y, there is no $\checkmark$ event to synchronise with, but we allow time to pass while waiting for termination, so the $\checkmark$ event behaves similarly to a *tock* event in parallel composition. The result is thus that of prepending a refusal Z and a *tock* event to the traces formed by merging the $\checkmark$ event with the trace after the input *tock* event. The refusal Z is drawn from the refusals formed by merging Σ with Y. The *tock* event thus follows on from the trace ending with a refusal that results from equation (61).

Equations (69) to (74) define the cases where a trace begins with a non-$\checkmark$ non-*tock* event e. The first four of these cover the cases where the other trace also begins with a non-$\checkmark$ non-*tock* event f. If neither e nor f require synchronisation (equation (69)), then the result is the union of the traces where the event e occurs and those where the event f occurs. The traces where the event e occurs are formed by prepending an occurrence of the event e to the result of merging the remainder of the trace with the other trace. The traces where f occurs are produced by the dual of this process. If e does not require synchronisation but f does (equation (70)), then the result is just the traces where e occurs, since f has nothing to synchronise with in this case.

When both e and f require synchronisation, then we must consider whether or not they are the same event. If they are the same event (equation (71)),

then they can synchronise with one another. The result in this case is that of prepending the event to the traces formed by merging the traces after e and f. If e and f are not equal (equation (72)), then the empty trace is the only possible trace, since they cannot synchronise with each other.

When the trace starting with e is merged with a trace starting with a *tock* event (and its associated refusal), then the *tock* event requires synchronisation, which e cannot provide. The result then depends on whether e requires synchronisation. If e does not require synchronisation (equation (73)), then the result is similar to that of merging it with an event f requiring synchronisation (equation (70)). If e requires synchronisation (equation (74)), then both sides require synchronisation so the only result is the empty trace.

Example 12. Merging the trace $\langle evt\ a \rangle$ from $a \longrightarrow \mathbf{Stop}$ with the traces from $b \longrightarrow c \longrightarrow \mathbf{Stop}$ yields the traces shown below.

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle\rangle = \{\langle evt\ a \rangle\} \tag{94}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle evt\ b \rangle = \begin{array}{r}\{\langle evt\ a, evt\ b \rangle, \\ \langle evt\ b, evt\ a \rangle\}\end{array} \tag{95}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle ref\ \{a, c, \checkmark\} \rangle = \{\langle evt\ a \rangle\} \tag{96}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle ref\ \{a, c, \checkmark\}, evt\ tock \rangle = \{\langle evt\ a \rangle\} \tag{97}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle ref\ \{a, c, \checkmark\}, evt\ tock, evt\ b \rangle = \{\langle evt\ a \rangle\} \tag{98}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle evt\ b, ref\ \{a, c, \checkmark\} \rangle = \begin{array}{r}\{\langle evt\ a, evt\ b \rangle, \\ \langle evt\ b, evt\ a \rangle\}\end{array} \tag{99}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle evt\ b, ref\ \{a, c, \checkmark\}, evt\ tock \rangle = \begin{array}{r}\{\langle evt\ a, evt\ b \rangle, \\ \langle evt\ b, evt\ a \rangle\}\end{array} \tag{100}$$

$$\langle evt\ a \rangle \llbracket \{c\} \rrbracket^T \langle evt\ b, evt\ c \rangle = \begin{array}{r}\{\langle evt\ a, evt\ b \rangle, \\ \langle evt\ b, evt\ a \rangle\}\end{array} \tag{101}$$

Similarly to the case shown in equation (78), merging $\langle evt\ a \rangle$ with the empty trace (equation (94)), yields the set containing a trace with just an occurrence of a. When there are events on both sides (equation (95)), either event can occur first and is followed by the result of merging the rest of the trace (defined as in equations (78) and (94)). Both orderings of the events a and b can thus be seen in equation (95).

When there is a refusal or *tock* event at the start (equations (96) and (97)), the event a can still occur, but there is nothing to match the refusal or *tock* event. This can also be seen in equation (98), where both *tock* and b are excluded from the resulting traces, since *tock* is not matched. However, if b occurs before a refusal, *tock* or another event that requires synchronisation (equations (99), (100) and (101)), the a and b events can occur in either order (like in equation (95)). No further observations are possible after a and b have been performed, since the observation after b in these cases is not matched by a corresponding observation after a.

Finally, if both sides begin with *tock* events, preceded by refusals X and Y (equation (75)), then the *tock* events synchronise with each other. The result

is all the traces formed by prepending a refusal Z and the *tock* event to the traces resulting from merging the rest of the trace on each side. The refusal Z comes from merging the refusals X and Y from each side as if they were refusals refusals on their own (deferring to equation (60).

Example 13. Merging traces from $a \longrightarrow \mathbf{Stop}$ and $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that contain one *tock* event yields the traces shown below.

$$\begin{array}{l} \langle ref\ \{b, c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle ref\ \{a, c, \checkmark\}, evt\ tock\rangle \end{array} = \{\} \tag{102}$$

$$\begin{array}{l} \langle ref\ \{b, c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle ref\ \{a, c, \checkmark\}, evt\ tock, evt\ b\rangle \end{array} = \{\} \tag{103}$$

$$\begin{array}{l} \langle ref\ \{c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle ref\ \{c, \checkmark\}, evt\ tock\rangle \end{array} = \begin{array}{l} \{\langle ref\ \{c, \checkmark\}, evt\ tock\rangle, \\ \qquad \langle ref\ \{c\}, evt\ tock\rangle, \ldots\} \end{array} \tag{104}$$

$$\begin{array}{l} \langle ref\ \{c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle ref\ \{c, \checkmark\}, evt\ tock, evt\ b\rangle \end{array} = \begin{array}{l} \{\langle ref\ \{c, \checkmark\}, evt\ tock, evt\ b\rangle, \\ \qquad \langle ref\ \{c\}, evt\ tock, evt\ b\rangle, \ldots\} \end{array} \tag{105}$$

$$\begin{array}{l} \langle ref\ \{c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle evt\ b, ref\ \{\checkmark\}, evt\ tock\rangle \end{array} = \begin{array}{l} \{\langle evt\ b, ref\ \{c, \checkmark\}, evt\ tock\rangle, \\ \qquad \langle evt\ b, ref\ \{c\}, evt\ tock\rangle, \ldots\} \end{array} \tag{106}$$

$$\begin{array}{l} \langle evt\ a, ref\ \{a, b, c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle evt\ b, ref\ \{a, b, \checkmark\}, evt\ tock\rangle \end{array} = \begin{array}{l} \{\langle evt\ a, evt\ b, ref\ \{a, b, c, \checkmark\}, evt\ tock\rangle, \\ \qquad \langle evt\ b, evt\ a, \\ \qquad\quad ref\ \{a, b, c, \checkmark\}, evt\ tock\rangle, \\ \qquad \langle evt\ a, evt\ b, \\ \qquad\quad ref\ \{a, b, c\}, evt\ tock\rangle, \ldots\} \end{array} \tag{107}$$

$$\begin{array}{l} \langle evt\ a, ref\ \{a, c, \checkmark\}, evt\ tock\rangle \\ \qquad [\![\, \{c\} \,]\!]^T \\ \langle ref\ \{a, c, \checkmark\}, evt\ tock, evt\ b\rangle \end{array} = \begin{array}{l} \{\langle evt\ a, ref\ \{a, c, \checkmark\}, evt\ tock, \\ \qquad\quad evt\ b\rangle, \\ \qquad \langle evt\ a, ref\ \{a, c\}, evt\ tock, \\ \qquad\quad evt\ b\rangle, \ldots\} \end{array} \tag{108}$$

When two *tock* events are merged, the refusals before them are merged in a similar way to single refusals (Example 11). In particular, refusals that do not contain the same non-synchronised events yield no output traces, as shown in equations (102) and (103). When the refusals before the *tock* events match, the result is the traces with *tock* events preceded by refusals drawn from the subsets of the union of the refusals on each side, as in equation (104). Any observations after a *tock* event are then merged, as can be seen in equation (105), where a b event follows the *tock* event.

Since a *tock* event requires synchronisation, any events that do not require synchronisation occurring before *tock* in an input trace are included before it the output traces. Thus, in equation (106) a b event occurs before the *tock* event. When there are non-synchronised events at the start of the traces on both sides, as in equation (107), the non-synchronised events can occur in either order before the *tock* event. If an event on one side occurs before *tock* while the event on the other side occurs after *tock*, the ordering with respect to *tock* is maintained. This can be seen in equation (108), where a occurs before *tock* and b occurs after *tock*.

Hiding The semantics of hiding, $P \setminus X$, is defined as shown below. It is specified as the distributed union of the traces defined by applying to each trace of P a function *hideTrace*, which elides the events in X. A set is output by *hideTrace* in order to properly ensure subset closure, as we discuss below as we present the definition of *hideTrace*.

$$tttraces[\![P \setminus X]\!] = \bigcup \{p : tttraces[\![P]\!] \bullet hideTrace\ X\ p\}$$

The *hideTrace* function takes a trace and the set X of events to hide, and outputs the set of traces corresponding to the input trace, with the events in X hidden. The *hideTrace* function is defined recursively, considering the different cases for the traces, as specified by the predicate below.

$$\begin{aligned}
&\forall X, Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark}_{tock};\ s : TickTockTrace \bullet \\
&\quad hideTrace\ X\ \langle\rangle = \{\langle\rangle\} \land && (109)\\
&\quad hideTrace\ X\ \langle ref\ Y\rangle = \{Z : \mathbb{P}\ Y \mid X \subseteq Y \bullet \langle ref\ Z\rangle\} \land && (110)\\
&\quad (e \in X \Rightarrow hideTrace\ X\ (\langle evt\ e\rangle \frown s) = hideTrace\ X\ s) \land && (111)\\
&\quad (e \notin X \Rightarrow hideTrace\ X\ (\langle evt\ e\rangle \frown s) = \\
&\qquad \{t : hideTrace\ X\ s \bullet \langle evt\ e\rangle \frown t\}) \land && (112)\\
&\quad (tock \in X \Rightarrow hideTrace\ X\ (\langle ref\ Y, evt\ tock\rangle \frown s) = \\
&\qquad hideTrace\ X\ s) && (113)\\
&\quad (tock \notin X \Rightarrow hideTrace\ X\ (\langle ref\ Y, evt\ tock\rangle \frown s) = \\
&\qquad \{Z : \mathbb{P}\ Y;\ t : hideTrace\ X\ s \mid X \subseteq Y \bullet \langle ref\ Z, evt\ tock\rangle \frown t\}) && (114)
\end{aligned}$$

The first equation, (109), specifies that applying *hideTrace* to the empty trace just returns the set containing the empty trace. This is because there are no events to hide, so the trace is simply returned as-is.

The other base case of the definition is specified by equation (110), which describes the result of applying *hideTrace* to a trace $\langle ref\ Y\rangle$ consisting of a single refusal. In this case, we check whether the set X of events to hide is a subset of the refusal Y. If it is not, then some of the events in X are not refused and so may be performed. The hiding of these events turns them into internal events, which are unstable, so the refusal is removed and the output of *hideTrace* is the empty set. When X is a subset of Y, all the traces with refusals that are subsets of Y are included. All the subsets must be included, since some subsets may not

include X and so are removed. We must reinclude such refusals where there is a refusal including X in order to maintain subset closure.

Equations (111) and (112) define the result of *hideTrace* when it is applied to a trace starting with a non-*tock* event e. There are two cases to consider, depending on whether e is in the hiding set X. Equation (111) specifies the case where e is in X. The result is that of applying *hideTrace* to the remainder of the trace, since e is hidden and removed. In the case where e is not in X, specified by equation (112), the result is that of prepending e to each of the traces resulting from applying *hideTrace* to the rest of the trace.

Finally, equations (113) and (114) define the result when *hideTrace* is applied to a trace starting with a refusal Y followed by a *tock* event. This case can be viewed as a combination of the cases for a non-*tock* event (equations (111) and (112)) and for a single refusal (equation (110)). As for a non-*tock* event, there are two cases depending on whether *tock* is in X or not. If *tock* is in X (equation (113)), then it is hidden and the result is that of applying *hideTrace* to the rest of the trace, as in equation (111). If *tock* is not in X (equation (114)), then it is prepended to the result of applying *hideTrace* to the rest of the trace, as in equation (112), but the refusal Y before the *tock* event is handled as in equation (110). If the hiding set X is a subset of Y, then refusals drawn from all the possible subsets are prepended before the *tock*. If X is not a subset of Y, then no traces are included, since at least one hidden event is not refused, so its hiding creates instability, but *tock* can only occur from a stable state.

Example 14. Considering the traces of $b \longrightarrow c \longrightarrow \mathbf{Stop}$, with the hiding set $\{a, b\}$, the results of applying *hideTrace* are those shown below.

$$\mathit{hideTrace}\,\{a,b\}\,\langle\rangle = \{\langle\rangle\} \tag{115}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{ref}\,\{a,c,\checkmark\}\rangle = \{\} \tag{116}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{ref}\,\{a,c,\checkmark\}, \mathit{evt}\ \mathit{tock}\rangle = \{\} \tag{117}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{ref}\,\{a,c,\checkmark\}, \mathit{evt}\ \mathit{tock}, \mathit{evt}\ b\rangle = \{\} \tag{118}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{evt}\ b\rangle = \{\langle\rangle\} \tag{119}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{evt}\ b, \mathit{evt}\ c\rangle = \{\langle \mathit{evt}\ c\rangle\} \tag{120}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{evt}\ b, \mathit{ref}\,\{a,b,\checkmark\}\rangle = \{\langle \mathit{ref}\,\{a,b,\checkmark\}\rangle, \langle \mathit{ref}\,\{a,b\}\rangle, \ldots\} \tag{121}$$

$$\mathit{hideTrace}\,\{a,b\}\,\langle \mathit{evt}\ b, \mathit{ref}\,\{a,b,\checkmark\}, \mathit{evt}\ \mathit{tock}\rangle = \{\langle \mathit{ref}\,\{a,b,\checkmark\}, \mathit{evt}\ \mathit{tock}\rangle, \langle \mathit{ref}\,\{a,b\}, \mathit{evt}\ \mathit{tock}\rangle, \ldots\} \tag{122}$$

Applying *hideTrace* to the empty trace yields just the set containing the empty trace (equation (115)). When *hideTrace* is applied to a trace with a refusal that does not include the hiding set, as in equation (116), no traces are output, since there is no stability if the hidden event can be performed. This applies even if some of the hidden events are refused. In equation (116) a is refused, but b is not, and an ocurrence of b is still hidden and so introduces instability. When *tock* is not hidden (as in this example), a similar rule applies to the refusal present before a *tock* event, as can be seen in equations (117) and (118).

When a b event occurs at the start, as in equation (119), since b is hidden, the only resulting trace is the empty trace. Observations after b are also subject to hiding. The event c is included on its own in the sole output trace in equation (120), since c is not hidden. The maximal refusal after the hidden b includes both events in the hiding set, so it is included (with its subsets), as can be seen in equation (121). This also applies to refusals before *tock* events, since *tock* is not hidden, as shown in equation (122).

Renaming The semantics of renaming, $P[f]$, is defined as shown below. The definition is similar to that of hiding in that it consists of a union of sets generated by applying to the traces of P a function *renameTrace*. This function takes the renaming function f as one of its inputs in addition to a trace p of P. We recall that the renaming function maps elements of $\Sigma^{\checkmark}_{tock}$ to elements of $\Sigma^{\checkmark}_{tock}$, but is required to identify $\checkmark$ and *tock*, since they cannot be renamed.

$$tttraces[\![P[f]]\!] = \bigcup \{p : tttraces[\![P]\!] \bullet renameTrace\, f\, p\}$$

The predicate defining *renameTrace* is shown below. For the empty trace it returns the set containing the empty trace (equation (123)). When *renameTrace* is applied to a trace beginning with event e (equation (124)), the result is the traces resulting from applying *renameTrace* to the rest of the trace, with the event formed from applying f to e prepended.

$$\forall f : \Sigma^{\checkmark}_{tock} \to \Sigma^{\checkmark}_{tock};\ e : \Sigma^{\checkmark}_{tock};\ X : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\ s : \mathrm{seq}\, Obs \bullet$$

$$renameTrace\, f\, \langle\rangle = \{\langle\rangle\} \land \tag{123}$$

$$renameTrace\, f\, (\langle evt\ e\rangle \frown s) =$$

$$\{t : renameTrace\, f\, s \bullet \langle evt\, (f\ e)\rangle \frown t\} \land \tag{124}$$

$$renameTrace\, f\, (\langle ref\ X\rangle \frown s) =$$

$$\{t : renameTrace\, f\, s;\ Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} \mid X = (f^{\sim})\, (\!|\ Y\ |\!) \bullet \langle ref\ Y\rangle \frown t\} \tag{125}$$

For a trace starting with a refusal X (equation (125)), the result is that of prepending a corresponding refusal Y to the traces from applying *renameTrace* the rest of the trace. The refusal Y is one whose image under the inverse of f is equal to X. This means that for the events refused in X, the corresponding events under f are refused in Y. It also allows for events not in the range of f to be included in Y, since such events cannot be performed in any trace that results from renaming and so must be refused (by TT2).

Example 15. We consider f to be $id \oplus \{a \mapsto b\}$, that is, the function that maps a to b and maps every other event to itself. Applying $renameTrace\, f$ to the traces in the semantics of $a \longrightarrow \mathbf{Stop}$ yields the results shown below.

$$renameTrace\, f\, \langle\rangle = \{\langle\rangle\} \tag{126}$$

$$renameTrace\, f\, \langle evt\ a\rangle = \{\langle evt\ b\rangle\} \tag{127}$$

$$renameTrace\, f\, \langle ref\ \{b, c, \checkmark\}\rangle = \{\} \tag{128}$$

$$renameTrace\, f\, \langle ref\, \{c, \checkmark\}\rangle = \{\langle ref\, \{a, c, \checkmark\}\rangle, \langle ref\, \{c, \checkmark\}\rangle\} \quad (129)$$

$$renameTrace\, f\, \langle ref\, \{c, \checkmark\}, evt\ tock\rangle = \begin{array}{l}\{\langle ref\, \{a, c, \checkmark\}, evt\ tock\rangle, \\ \quad \langle ref\, \{c, \checkmark\}, evt\ tock\rangle\}\end{array} \quad (130)$$

$$renameTrace\, f\, \langle evt\ a, ref\, \{a, b, c, \checkmark\}\rangle = \begin{array}{l}\{\langle evt\ b, ref\, \{a, b, c, \checkmark\}\rangle, \\ \quad \langle evt\ b, ref\, \{b, c, \checkmark\}\rangle\}\end{array} \quad (131)$$

Renaming the empty trace results in just the empty trace (equation (126)). Applying *renameTrace* to a trace with just a, results in a being renamed by f to b (equation (127)).

For a refusal, we must find corresponding refusals that map to under the inverse image of f. The maximal initial refusal $\{b, c, \checkmark\}$ thus yields no traces, as shown in equation (128), since any refusal that includes b has an inverse image under f including both a and b. Since our input trace does not refuse a, we cannot, therefore, output any refusal that refuses b, but without refusing b we cannot satisfy the refusal of b in the input trace, so there are no corresponding refusals for $\{b, c, \checkmark\}$. However, the subset refusal $\{c, \checkmark\}$ is the inverse image under f of both $\{a, c, \checkmark\}$ and $\{c, \checkmark\}$, since nothing maps to a under f. The result of applying *renameTrace* to a trace consisting of the refusal $\{c, \checkmark\}$ is thus the set containing the traces with refusals $\{a, c, \checkmark\}$ and $\{c, \checkmark\}$ (equation (129)).

For a trace consisting of a refusal followed by a *tock* event (equation (130)), the refusal is handled as in equations (128) and (129) and the *tock* event is included as-is, since f maps *tock* to itself (as all renaming functions should). Refusals occuring after renamed events are also handled in the same way as in equations (128) and (129), as can be seen in equation (131).

This concludes our explanation of the ✓-*tock* model. In the next section we show how to use FDR to reason about *tock*-CSP processes using ✓-*tock*.

5 Model-checking with FDR

We describe how timed sections can be used to specify processes in Section 5.1, and then in Section 5.2 we show how refinement in ✓-*tock* can be checked.

5.1 Processes

In FDR, processes are defined using $\texttt{CSP}_\texttt{M}$, the machine-readable version of CSP. For *tock*-CSP this means that we use the standard $\texttt{CSP}_\texttt{M}$ syntax within a timed section, with the exception of the untimed operators $\mathbf{Stop}_U$ and $\triangle_U$ that need to be declared outside of a timed section. For example, for the untimed interrupt operator this means declaring a parametrised definition `UInt`, for example, that uses the standard $\texttt{CSP}_\texttt{M}$ syntax `/\` for interrupt as `UInt(P,Q) = P /\ Q` and then using `UInt` within a timed section as required.

A timed section is written `Timed(et) {...}`, where `...` is a sequence of declarations, and `et` is a function from events to time, specifying how many *tock* events follow an event. Because events in our model are instantaneous `et` is defined as zero for every event. Below we show Example 3's $\texttt{CSP}_\texttt{M}$ encoding.

Example 16.

```
channel move, obs, halt, tock
USTOP = STOP
et(_) = 0
Timed(et) { C(s) = timed_priority((move -> STOP) /\
           (obs -> (halt -> SKIP [] (WAIT(s);USTOP)))) }
```

Before the timed section we declare: events `move`, `obs`, `halt` and `tock`; process `USTOP` that corresponds to $\mathbf{Stop}_U$; and the function `et`. (Events and functions can be equally declared inside a timed section.) Within the timed section `C` is specified using the CSP_M syntax with `timed_priority`[2] applied, which enforces maximal progress by giving ✓ and τ priority over `tock`. Here `C` is parametric, so that `s` can be instantiated to a particular value for model-checking.

Processes in a timed section are implicitly translated to ensure `tock` is offered whenever time is allowed to pass. For example, a prefixing `e -> P` is rewritten as `X = e -> P [] tock -> X`, where `X` is a process that offers event `e` and `tock` in an external choice (`[]`) so that time can pass before accepting `e`. This means that, as explained, the behaviour of operators in a timed section is that of the *tock*-CSP operators we define in ✓-*tock* as described in Section 4.

Parallel composition operators are translated to include `tock` in their synchronisation sets to ensure time is uniform. For example, the parallel composition `P [| s |] Q`, of processes `P` and `Q` synchronising on a set of events `s`, is translated to include `tock` in `s` as `P [| union(s,{tock}) |] Q`. As termination of a parallel composition requires termination of both operands, for `SKIP` not timestop a parallel composition, `SKIP` is translated to `TSKIP = SKIP [] tock -> TSKIP`. Prioritisation of `TSKIP` ensures that it matches the behaviour of **Skip** of ✓-*tock*.

We note that `timed_priority` has to be applied at the outer level of a process because it does not distribute through FDR's parallel composition operators. For example, `timed_priority(SKIP) ||| timed_priority(WAIT(1))`, where `|||` is a parallel composition where only `tock` is included as part of the synchronisation set, timestops as the application to `SKIP` removes the possibility to perform `tock`. On the other hand `timed_priority(SKIP ||| WAIT(1))` behaves in the expected way, that is, it can perform a single `tock` followed by termination.

5.2 Refinement

To encode the refinement relation for ✓-*tock* we tailor Mestel and Roscoe [18]'s approach to encoding refusals using traces, and extend it to cater for termination, so that ✓-*tock* refinement is reduced to traces refinement. We begin this section by explaining how refusals are encoded, followed by the enconding of ✓-*tock* traces, and termination. Finally we illustrate the technique with examples.

Refusals Given a set Σ_{tock} we define Σ'_{tock}, where each $e \in \Sigma_{tock}$ is replaced by a dashed counterpart e', used to indicate that e is refused. For a process P,

[2] cs.ox.ac.uk/projects/fdr/manual/cspm/prelude.html#function_timed_priority

a context $\mathcal{C}_1$ is defined below to define a process $\mathcal{C}_1[P]$ whose traces encode the refusals of P. We note that $\mathcal{C}_1$, and other definitions to follow, are specified in FDR outside of a timed section as they are not *tock*-CSP processes.

Definition 2. $\mathcal{C}_1[P] \mathrel{\widehat{=}} \mathbf{Pri}_{\leq_1}(P \mid\mid\mid RUN(\Sigma'_{tock} \cup \{stab\}))$

Process P is composed in interleaving ($|||$), a form of parallel composition where, outside a timed section, no synchronisation is required, with the process $RUN(\Sigma'_{tock} \cup \{stab\})$ that offers events in Σ'_{tock}, including a dashed version of *tock*, and the event *stab* that encodes an empty refusal, in an external choice forever. This composition is followed by the application of $\mathbf{Pri}_{\leq_1}$ to prioritise each event e over e', so that e' is only available whenever e is refused stably. Event *stab* is prioritised lower than τ and $\checkmark$, so an empty refusal can be observed whenever a process is not divergent or terminated, that is, for example, not the case for **div**.

The operator $\mathbf{Pri}_{\leq}(P)$ [20], implemented in FDR as `prioritisepo`, is a more general version of `timed_priority`, whereby events can be prioritised according to a partial order $\leq$. The behaviour is that of P, but changed so that whenever events a and b are available, then if b is of strictly higher priority than a, that is, $a < b$, then a, and the following behaviour from a, is pruned. For example prioritising the process $a \longrightarrow P \,\Box\, b \longrightarrow Q$ with $a < b$ would yield $b \longrightarrow \mathbf{Pri}_{\leq}(Q)$. In the above definition, $\leq$ is $\leq_1$, defined by $e <_1 e'$.

To illustrate the operational effect of $\mathcal{C}_1$ we consider the following example, assuming that a is the only event in Σ.

Example 17. $F \mathrel{\widehat{=}} (a \longrightarrow \mathbf{Stop}) \,\Box\, \mathbf{Stop}_U$

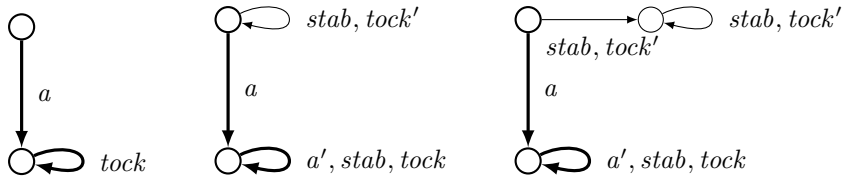
$$\mathit{tttraces}[\![F]\!] = \{\langle\rangle, \langle \mathit{ref}\ \{\mathit{tock}, \checkmark\}\rangle, \langle \mathit{evt}\ a\rangle, \langle \mathit{evt}\ a, \mathit{ref}\ \Sigma^{\checkmark}, \mathit{evt}\ \mathit{tock}\rangle, \ldots\}$$
$$\mathit{traces}(\mathcal{C}_1[F]) = \{\langle\rangle, \langle \mathit{stab}, \mathit{tock}'\rangle, \langle a\rangle, \langle a, a', \mathit{stab}, \mathit{tock}\rangle, \langle \mathit{stab}, \mathit{tock}', a\rangle, \ldots\}$$

Process F offers to do a immediately, because of the timestop $\mathbf{Stop}_U$, and then deadlocks. Its traces in $\checkmark$-*tock* with maximal refusals are: the empty sequence; the sequence with the only refusal containing both *tock* and $\checkmark$; and the sequence with event a, possibly concatenated with $\langle \mathit{ref}\ \Sigma^{\checkmark}, \mathit{evt}\ \mathit{tock}\rangle$ any number of times, corresponding to the behaviour of $\mathbf{Stop}_U$ after event a has happened.

The Labelled Transition System (LTS) resulting from the application of the operational semantics of CSP, which can be calculated using FDR, is shown in Fig. 2a. The application of $\mathcal{C}_1$ to F introduces additional transitions corresponding to the events being refused at each state. Thus in Fig. 2a, we have that in the initial state *tock* is refused, and so events *stab* and *tock*′ become available in Fig. 2b, and similarly for the next state. We note that, for now, we are not considering termination, which we discuss later on.

The trace $\langle \mathit{stab}, \mathit{tock}', a\rangle$ encoding the trace $\langle \mathit{ref}\ \{\mathit{tock}\}, \mathit{evt}\ a\rangle$, however, is undesirable because, in a $\checkmark$-*tock* trace, after a refusal the only possible event is *tock*. Next we introduce another context $\mathcal{C}_2$ to eliminate such undesirable traces.

Semantics Having encoded refusal events using $\mathcal{C}_1[P]$, it is then necessary to ensure they can only occur as permitted by the $\checkmark$-*tock* model. Thus, we define



(a) LTS for F. (b) Application of $\mathcal{C}_1$. (c) Application of $\mathcal{C}_2$.

Fig. 2. LTS calculated from F and stepwise application of contexts $\mathcal{C}_1$ and $\mathcal{C}_2$.

another context $\mathcal{C}_2[P]$, where $\mathcal{C}_1[P]$ is composed in parallel with a process *Sem* synchronising on events on the union of Σ_{tock}, Σ'_{tock} and $\{stab\}$. The role of *Sem* is to eliminate traces of $\mathcal{C}_1[P]$ that are not valid in ✓-*tock*.

Definition 3. $\mathcal{C}_2[P] \mathrel{\widehat{=}} \mathcal{C}_1[P] \llbracket \Sigma_{tock} \cup \Sigma'_{tock} \cup \{stab\} \rrbracket Sem$

The process *Sem* is defined below, where we use the standard (untimed) operators of CSP for external choice and prefixing, and $\Sigma'_{tock,stab} = \Sigma'_{tock} \cup \{stab\}$.

Definition 4.

$$Sem \mathrel{\widehat{=}} \left(\Box\, e : \Sigma_{tock} \bullet e \longrightarrow Sem\right) \Box \left(\Box\, r : \Sigma'_{tock,stab} \bullet r \longrightarrow Ref\right)$$

$$Ref \mathrel{\widehat{=}} \left(\Box\, r : \Sigma'_{tock,stab} \bullet r \longrightarrow Ref\right) \Box\, tock \longrightarrow Sem$$

Sem offers every event e from Σ_{tock} in an external choice followed by a recursion, and every event r encoding refusals from $\Sigma'_{tock,stab}$ also in an external choice, but followed by the behaviour of *Ref*. That process also offers events encoding refusals from $\Sigma'_{tock,stab}$ followed by a recursion, but *tock* is also offered followed by the behaviour of *Sem*. So, a trace of *Sem* includes any number of original events from Σ_{tock}, until a refusal event r from $\Sigma'_{tock,stab}$ occurs, when we then have any number of such events, before a *tock*, and we can again have original events. This encodes the possibility to observe events from a refusal set at the end of a trace of original events, and before *tock* events.

To illustrate the application of $\mathcal{C}_2$ to process F from Example 17 we consider the LTS in Fig. 2c. The self transition on the initial node obtained from the application of $\mathcal{C}_1$ is replaced by a transition on the same events, *stab* and *tock*′, to a node that accepts these events indefinitely, but not a. This is because initially *tock* can be refused, and so a refusal event *tock*′, encoding a refusal set where *tock* is refused, cannot be followed by any regular event.

Example 18.

$$traces(\mathcal{C}_2[F]) = \{\langle\rangle, \langle stab, tock'\rangle, \langle a\rangle, \langle a, a', stab, tock\rangle, \langle stab, tock', stab, \ldots\rangle, \ldots\}$$

The ✓-*tock* traces of F before observing event a are encoded by traces $\langle\rangle$, $\langle stab, \ldots\rangle$, $\langle tock', \ldots\rangle$, where *tock*′ and *stab* are offered continuously. This is

effectively an encoding of the set $\{tock\}$ via traces. Traces of F after a are similarly encoded by $\langle a \rangle$ concatenated with $\langle a', \ldots \rangle$ or $\langle stab, \ldots \rangle$ any number of times, with *tock* being possible after each event a' or *stab* in the traces. More importantly, subset inclusion of refusal sets corresponds to subset inclusion over the set of encoding traces, which is key to reducing refinement of traces with refusal information, that is, ✓-*tock* traces, to traces refinement.

Termination The original encoding in [18] does not account for termination. For example, we have that $\mathcal{C}_1[\mathbf{Skip}] = \mathcal{C}_1[\mathbf{Stop}]$. Because in $\mathcal{C}_1$ there is an interleaving, termination of **Skip** does not lead to termination of $\mathcal{C}_1[\mathbf{Skip}]$, and instead refusal information is added exactly as in the case of **Stop**. To cater for termination, we extend the encoding by defining a third context $\mathcal{C}_3$ and extending Σ_{tock} with a fresh event *tick* that explicitly encodes ✓.

Definition 5. $\mathcal{C}_3[P] \mathrel{\widehat{=}} \mathcal{C}_2[P \,;\; tick \longrightarrow \mathbf{Skip}]$

Thus we sequentially compose P with the prefixing on event *tick* before applying context $\mathcal{C}_2$, so that actual termination is not masked by the interleaving in $\mathcal{C}_1$. This enables us to establish the following key result whereby P is refined by Q in the ✓-*tock* model if, and only if, its encoding using $\mathcal{C}_3[P]$ is refined by $\mathcal{C}_3[Q]$ in the traces model ($\mathcal{T}$) of CSP. A script with the complete encoding is available[3].

Theorem 1. $P \sqsubseteq Q \Leftrightarrow \mathcal{C}_3[P] \sqsubseteq_{\mathcal{T}} \mathcal{C}_3[Q]$.

Proof. Similarly to that outlined in [18] by following the above construction.

Full mechanisation in $\texttt{CSP}_\texttt{M}$ The above construction is mechanised in $\texttt{CSP}_\texttt{M}$, as shown in Figure 3, using a parametric module `MS(Sigma)` of name `MS` and whose only parameter `Sigma` corresponds to Σ. A $\texttt{CSP}_\texttt{M}$ module consists of three parts: an optional sequence of parameters that allows modules to be instantiated with different values, a sequence of zero or more declarations internal to a module, and a sequence of declarations to be made visible outside a module.

In our case, we have a single parameter `Sigma`. Within the internal declarations of `MS(Sigma)` we have channels `stab` and `tick`, corresponding to *stab* and *tick*, and a channel `ref` whose type includes events in `Sigma`, and the events `tick` and `tock`. This parametric declaration is a practical way of specifying events encoding refusals, where e' events are encoded as `ref.e`, and so `ref` corresponds to $\Sigma'_{tock,tick} = \Sigma'_{tock} \cup \{tick'\}$. The partial `order` is a relation specified by pairs of events `(x,ref.x)`, when `x` has priority over `ref.x`, thus corresponding to $\leq_{\mathbf{1}}$.

Process `C1(P)` is the mechanised version of $\mathcal{C}_1[P]$, here specified as the prioritisation of the interleaving of processes `P` and `RUN({|ref,stab|})`. The function `prioritisepo` takes four parameters: a process to be prioritised, a set of events that are affected by prioritisation, a partial order, and a set of events whose priority is the same as ✓ and τ. In our case, we have that: the set of events affected by prioritisation is the union of `Sigma`, the channel set `ref`, and events

[3] github.com/robo-star/tick-tock-CSP/tree/master/fdr

```
module MS(Sigma)

channel stab, tick
channel ref:union(Sigma,{tock,tick})

order = {(x,ref.x) | x:union(Sigma,{tock,tick})}

C1(P) = prioritisepo(P ||| RUN({|ref,stab|}),
                     union(Sigma,{|ref,tock,tick|}), order,
                     union(Sigma,{tock,tick}))
C2(P) = C1(P) [| union(Sigma,{|ref,stab,tock|}) |] Sem

Sem = ([] x : union(Sigma,{tock,tick}) @ x -> Sem)
      [] (ref?x -> Ref)
      [] (stab -> Ref)

Ref = (ref?x -> Ref) [] (stab -> Ref) [] tock -> Sem

exports

C3(P) = C2(P ; tick -> SKIP)

endmodule
```

Fig. 3. Mechanisation in $\mathtt{CSP_M}$

`tock` and `tick`; the order is that specified by the ordered pairs in `order`; and events `tick`, `tock` and those in `Sigma` are specified as having the same priority as ✓ and τ. Maximal progress is not affected assuming that `P` itself has already been prioritised using `timed_priority` as required in *tock*-CSP.

The mechanised version of context $\mathcal{C}_2[P]$ is `C2(P)`. The definition is very similar. The synchronisation set is that obtained as the union of `Sigma`, the channel set `ref`, and the events `stab`, `tick` and `tock`, corresponding to the union $\Sigma_{tock,tick} \cup \Sigma'_{tock,tick} \cup \{stab\}$ in the definition of $\mathcal{C}_2$.

Finally, $\mathcal{C}_3[P]$ is defined by `C3(P)` within `MS(Sigma)` as being exported, that is, it can be used from outside the module. The module `MS` can be instantiated, for example, as `M` for a set of events a and b using `instance M = MS({a,b})`, and thus `M::C3(P)` can be used to access process `C3` within module `MS` instantiated with `Sigma` as the set of events `{a,b}`.

Examples To illustrate the application of our refinement technique to use of FDR to check for refinement in *tock*-CSP we reconsider Examples 1 and 2.

Example 1 We consider processes R and S of Example 1, defined outside a timed section to illustrate the flexibility in defining *tock*-CSP processes in FDR,

show their equivalent ✓-*tock* definitions in a timed section, by way of refinement checking, and compare the result to failures refinement.

```
R = (a -> SKIP [] b -> SKIP [] tock -> R) |~| RUN({tock})
S = (a -> SKIP [] tock -> S) |~| RUN({tock})
```

Process `R` makes an internal choice (`|~|`). It may offer events `a`, `b` and `tock` in an external choice (`[]`), where `tock` is followed by a recursion on `R`, and `a` and `b` lead to termination (`SKIP`), or offer `tock` indefinitely. `RUN({tock})` is effectively a timed deadlock, offering only the event `tock` indefinitely. Similarly, process `S` may also offer `tock` indefinitely, or decide to offer `a` and `tock` in an external choice, where `a` leads to termination, and `tock` to a recursion on `S`.

Processes `R` and `S` can be restated as `R1` and `S1` below in a timed section, using a time-synchronising external choice and **Stop**.

```
Timed(et) {
  R1 = (a -> SKIP [] b -> SKIP) |~| STOP
  S1 = (a -> SKIP) |~| STOP

  assert M::C3(timed_priority(R1))  [T= M::C3(R)
  assert M::C3(R)                   [T= M::C3(timed_priority(R1))

  assert M::C3(timed_priority(S1))  [T= M::C3(S)
  assert M::C3(S)                   [T= M::C3(timed_priority(S1))
}
```

To check that `R1` is equivalent to `R`, and that `S1` is equivalent to `S`, we instantiate `MS` as `instance M = MS({a,b})` and then check that the refinement over traces (`[T=`) holds in both directions for each pair of processes. Namely, in the case of `R1` we check that, having applied `timed_priority` and put it into context `M::C3`, it refines and is refined by `M::C3(R)`. Similarly for the case of `S1` and `S`. As expected FDR finds no problem with these assertions.

More importantly, however, in the failures and failures-divergences model, `R` is refined by `S`, because, even though `S` does not offer `b`, refusal of `b` is a possible behaviour of `R`. In a timed setting, however, this should not be the case as we previously observed in Section 1. To check that indeed this refinement does not hold in ✓-*tock* we state it as a negated assertion as follows.

```
assert not M::C3(timed_priority(R1)) [T= M::C3(timed_priority(S1))
```

Above, we observe the use of `not` as the assertion is expected to fail. FDR yields a counter-example where after the trace `<M::ref.b,tock>` process `S1` can perform event `a` but `R1` cannot. That is, having refused b, followed by a *tock*, process `R1` then behaves as **Stop**, whereas `S1` can perform a.

Example 2 In the case of Example 2, we observe that the behaviour of T and U differed when considering a refusal testing semantics, that is, the behaviour within a time unit is not as expected in the failures model. We first restate processes T and U in a timed section as follows.

Semantic Model	Termination	Deadlines	Liveness
Stable failures [1]	✓	✓	current
Refusal testing [2]	✓	✓	full-history
Discrete-time failures [21]	✓	×	full-history
Discrete-time refusals [22]	×	×	timed-history
Timed testing [22]	×	×	timed-history
✓-tock	✓	✓	timed-history

Table 3. Comparison of semantic models using *tock*.

```
Timed(et) {
  T = (a -> STOP [] b -> STOP) |~| c -> STOP
  U = (a -> STOP |~| c -> STOP) [] (b -> STOP |~| c -> STOP)

  T1 = timed_priority(T [] USTOP)
  U1 = timed_priority(U [] USTOP)
}
```

Furthermore, to analyse their behaviour within a single time unit, in the context of ✓-*tock*, we define $T_1 = T \,\Box\, \mathbf{Stop}_U$ and $U_1 = U \,\Box\, \mathbf{Stop}_U$, that is, in both U_1 and T_1, the prefixing on a, b or c must be immediate. Instantiating `MS` as `instance M = MS({a,b,c})` we can then state the following assertions.

```
  assert M::C3(timed_priority(T1))  [T= M::C3(timed_priority(U1))
  assert M::C3(timed_priority(U1))  [T= M::C3(timed_priority(T1))
```

FDR confirms that they both hold, and so $T_1 = U_1$ as required, that is, refinement holds within zero time, as previously discussed in Section 1.

6 Related work

In Table 3 we give a comparison of the different semantic models for *tock*-CSP according to: whether they account for termination; whether they support deadlines; and how liveness information is recorded. We classify liveness as: *current* if it is relative to the sequence of events only; *full-history* if, in addition, it takes into account liveness across the whole history of interactions, including before each event; *timed-history* if the history is only in relation to liveness of previous time units, but not before each event. For timed refinement of *tock*-CSP, where the laws of failures semantics of CSP hold within each time unit, it is *timed-history* liveness that is required as we have shown.

Most case studies in the *tock*-CSP literature using refinement focus on safety only[11,10], for which the trace semantics of CSP is adequate. For example, Evans and Schneider [10] consider an embedding of *tock*-CSP in PVS [23] using the traces model for analysis of time-dependent security properties. An embedding in Isabelle/HOL of the stable-failures model of CSP has also been considered by Isobe and Roggenbach [24]. However, as already discussed, to reason about timed refinement we need a richer model encompassing refusals over time.

The earliest introduction to *tock*-CSP appears in Chapter 14 of Theory and Practice of Concurrency [25]. Despite using timestops, Roscoe later describes these undesirably as "breaching the laws of nature by preventing time from progressing" [1]. Similarly, Ouaknine's discrete-time refusal testing model [2] does not admit timestops. Like Timed CSP, there is no explicit control of time, thus time can pass arbitrarily between events, but Zeno behaviour is forbidden.

More recently, Lowe and Ouaknine [22] revisited the discrete-time refusal testing model by considering traces where refusals are only recorded before *tock*, but which also do not admit timestops. Termination is also not considered in that work. On the other hand, they have also proposed a timed testing model whereby the null refusal is dropped. That model is similar to ours, in that we can also record refusals before a *tock* and do not record null refusals.

The discrete-time failures model is perhaps closest to ours, but unlike ours, does not allow *tock* to be refused and so cannot model deadlines. Armstrong et al. [21] explore refinement checking in that model by using the refusal testing model in FDR2. Because refusals before events other than *tock* need to be "ignored" to yield the right refinement relation, and not that of refusal testing, the construction is not trivial. A different encoding has also been considered by Roscoe [26] using the concept of slow-abstraction. Our model, on the other hand, allows the refusal of *tock*, as required for the specification of deadlines, is fully specified in Isabelle/HOL, and is amenable to model checking with FDR using an intuitive encoding via traces that accounts for termination.

7 Conclusions

The inclusion of the event *tock* in CSP enables a rich and flexible approach to modelling time, which makes reusing theories and tools feasible. However, as discussed, despite several models employing *tock*, none have, so far, adequately catered for deadlines, termination, capturing erroneous Zeno behaviour and timed refinement in a way that is compatible with a view of *tock*-CSP as a language with a failures-based semantics within each time unit.

In this work we have considered *tock*-CSP as a language on its own right by defining its operators, consistently with their use in FDR's timed sections, and a semantic model adequate for timed refinement. The model, and its operators, have been mechanised in Isabelle/HOL for the purpose of establishing key results. It is an environment for mechanical proving of laws and paves the way for the development of symbolic refinement tools for *tock*-CSP.

Despite the denotational setting of our study, an intuitive approach [18] using priorities can be used to perform analysis using FDR. This is at the expense of introducing additional processes and events, for encoding via traces. However, because traces refinement checking can be parallelised by FDR, in our experience the approach remains tractable for models of modest complexity.

It is in our plans to prove laws of $\checkmark$-*tock*, using our mechanisation, in support of a refinement strategy for semi-automatic generation of sound simulations for robotics [14]. It is clear that $\mathbf{Pri}_{\leq}$ endows CSP with extra expressive power [20],

allowing, for example, regular events to be made urgent by prioritising them over *tock*. It remains to be seen how it can play a direct role in our semantics beyond its use in FDR to ensure maximal progress.

Acknowledgements This work is funded by the EPSRC grants EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engineering. No new primary data was created as part of the study reported here.

References

1. Roscoe, A.W.: Understanding concurrent systems. Springer (2010)
2. Ouaknine, J.: Discrete analysis of continuous behaviour in real-time concurrent systems. PhD thesis, University of Oxford (2000)
3. Davies, J.: Specification and proof in real time CSP. Number 6. Cambridge University Press (1993)
4. Ouaknine, J., Worrell, J.: Timed CSP = closed timed automata. Electronic Notes in Theoretical Computer Science **68**(2) (2002) 142–159
5. Roscoe, A., Reed, G.: A timed model for communicating sequential processes. Theoretical Computer Science **58** (1988)
6. Schneider, S.: Concurrent and Real-time systems. Wiley (2000)
7. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In brahm, E., Havelund, K., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 8413 of Lecture Notes in Computer Science. (2014) 187–201
8. Leuschel, M., Butler, M.: ProB: A model checker for B. In: International Symposium of Formal Methods Europe, Springer (2003) 855–874
9. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness. Volume 5643 of Lecture Notes in Computer Science., Springer (2009) 709–714
10. Evans, N., Schneider, S.: Analysing time dependent security properties in CSP using PVS. In: European Symposium on Research in Computer Security, Springer (2000) 222–237
11. Kharmeh, S.A., Eder, K., May, D.: A design-for-verification framework for a configurable performance-critical communication interface. In: International Conference on Formal Modeling and Analysis of Timed Systems, Springer (2011) 335–351
12. Isobe, Y., Moller, F., Nguyen, H.N., Roggenbach, M.: Safety and line capacity in railways–an approach in Timed CSP. In: International Conference on Integrated Formal Methods, Springer (2012) 54–68
13. Göthel, T., Bartels, B.: Modular design and verification of distributed adaptive real-time systems. In Vinh, P.C., Vassev, E., Hinchey, M., eds.: Nature of Computation and Communication, Cham, Springer International Publishing (2015) 3–12
14. Cavalcanti, A., Sampaio, A., Miyazawa, A., Ribeiro, P., Conserva Filho, M., Didier, A., Li, W., Timmis, J.: Verified simulation for robotics. Science of Computer Programming (2019)
15. Phillips, I.: Refusal testing. Theoretical Computer Science **50**(3) (1987) 241–284
16. Mukarram, A.: A refusal testing model for CSP. PhD thesis, University of Oxford (1993)
17. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)

18. Mestel, D., Roscoe, A.: Reducing complex CSP models to traces via priority. Electronic Notes in Theoretical Computer Science **325** (2016) 237–252
19. Baxter, J., Ribeiro, P.: tick-tock-CSP in Isabelle/HOL. `https://github.com/robo-star/tick-tock-CSP/` (April 2019)
20. Roscoe, A.: The expressiveness of CSP with priority. Electronic Notes in Theoretical Computer Science **319** (2015) 387–401
21. Armstrong, P., Lowe, G., Ouaknine, J., Roscoe, A.: Model checking Timed CSP. In Proceedings of HOWARD (Festschrift for Howard Barringer) (2012)
22. Lowe, G., Ouaknine, J.: On timed models and full abstraction. Electronic Notes in Theoretical Computer Science **155** (2006) 497–519
23. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: International Conference on Automated Deduction, Springer (1992) 748–752
24. Isobe, Y., Roggenbach, M.: A generic theorem prover of CSP refinement. In Halbwachs, N., Zuck, L.D., eds.: Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 108–123
25. Roscoe, A.W.: The theory and practice of concurrency. (1998)
26. Roscoe, A.W.: The automated verification of timewise refinement. (2013)