Acta Informatica
https://doi.org/10.1007/s00236-020-00394-3

ORIGINAL ARTICLE

Sound reasoning in *tock*-CSP

James Baxter[1] · Pedro Ribeiro[1] · Ana Cavalcanti[1]

Received: 26 November 2019 / Accepted: 5 December 2020

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-time behaviours with powerful tool support. It uses an event *tock*, interpreted to mark passage of time. Analysis, however, has traditionally used the standard semantics of CSP, which is inadequate for reasoning about timed refinement. The most recent version of the model checker 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 the first sound tailored semantic model for *tock*-CSP that captures timewise refinement. It is fully mechanised in Isabelle/HOL and, to enable use of FDR4 to check refinement in this novel model, we use model shifting, which is a technique that explicitly encodes refusals in traces.

1 Introduction

In the realm of cyber-physical systems, time is a crucial concern. Such reactive systems can be modelled as cooperating with their environment via named events that correspond to atomic and instantaneous interactions of interest over their lifetime. In CCS [25] and CSP [38], 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 safety and liveness over time.

To encompass the notion of real time, several timed semantics have been proposed for CSP [31,32]. Early works on continuous Timed CSP include those of Reed and Roscoe [36], Davies [10] and Schneider [40]. The solid foundations of CSP with algebraic, denotational, and operational semantics gave rise to practical refinement checking, via model checking with FDR [13] and other tools [21,41]. As far as we know, however, no such effort has been made for Timed CSP, so that no tool has been developed specifically for Timed CSP.

This work is funded by 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.

✉ James Baxter
james.baxter@york.ac.uk

1 Department of Computer Science, University of York, York, UK

Developing a tool for reasoning about continuous timed models is a challenging task, as is using generic tools to reason about a Timed CSP model. Instead, Roscoe [37] has proposed *tock*-CSP, where the event *tock* encodes the passage of discrete time, allowing existing CSP tools like FDR to be used for reasoning about timed models. In addition, using *tock*-CSP, deadlines can be specified via timestops, that is, by refusing *tock*, and models can be decomposed into timed and untimed processes, facilitating abstraction and modularity. Extensive use of *tock*-CSP has been reported, including, for example, in the verification of security properties [11], the design of general-purpose I/O controllers [19], the study of railways [17], the verification of distributed adaptive systems [14], and more recently, in the verification of simulations for robotics [7].

The most recent version of FDR offers a syntactic environment called a timed section[1] that translates untimed processes into *tock*-CSP, to facilitate the specification of timed models, reusing the syntax of standard CSP and interpreting the operators in the context of a (discrete) timed semantics. Maximal progress, where time only advances after internal behaviour has stabilised, can be enforced by prioritising internal actions τ, and $\checkmark$, which signals termination, over *tock*.

Most case studies in the literature using *tock*-CSP focus on safety [11,14,19], rather than liveness [7,17]. Although useful, safety only is a weak notion of conformance. The standard traces, failures, and failures-divergences semantics of CSP, however, are inadequate for reasoning about timewise refinement [40], which ensures preservation of safety and liveness over time.

Although other semantic models are available, they either do not cater for the specification of deadlines [3,22,31] and termination [22], or for the *tock*-CSP view that, within each time unit, the standard (untimed) failures semantics of CSP holds [22,32]. Moreover, the use of deadlines may lead to Zeno behaviours, where an infinite number of events is required to take place in a finite amount of time. Just like divergence, this is undesirable; a useful model needs to be able to express Zeno behaviours, so that we can use it to prove their absence.

In this paper, we characterise *tock*-CSP as a language in its own right, with operators whose behaviour is as 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 novel semantic model for *tock*-CSP that allows the specification of deadlines, that caters for termination and Zeno behaviour, and whose refinement relation is timewise refinement.

The model and operators are specified in Isabelle/HOL [30]. Thus another contribution is an environment for mechanical theorem proving that paves the ground for the development of refinement tools for *tock*-CSP. We also illustrate how model shifting [23], a technique for reducing refinement over different CSP semantics to traces refinement, can be used for reasoning with FDR4 [13].

In Sect. 2 we review in detail the existing semantics studied in the context of *tock*-CSP, providing a more detailed account of the motivation and novelty of our model. In Sect. 3 we introduce the *tock*-CSP language. The denotational model is defined in Sect. 4 and the operators in Sect. 5. Section 6 illustrates the mechanisation in Isabelle/HOL, while Sect. 7 provides a didactic account of how processes, and refinement, can be encoded in FDR4. We conclude in Sect. 8 by summarising our contributions and discussing future work.

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

2 Related work

The literature is rich in timed models [1,2,6,8,12,20,26] for process calculi, with several timed variants of CCS [16] and ACP [4] reported, for example. In particular, [15,27,29] also employ special actions to mark the passage of discrete time, just like $tock$ is later considered for CSP in [37]. The principles of abstract time, maximal progress [15,43], instantaneous events, and the uniform passage of time [29], are also widely adopted. Some models do not assume uniform passage of time or allow the environment to arbitrarily delay visible events [15,43].

None of these models cater for termination (or sequential composition), with deadlock being the only basic process in most models. To cater for $tock$-CSP, our model allows the imposition of deadlines so that the environment may not be able to delay an event arbitrarily, ensures maximal progress, and allows the specification of termination, so that processes, not just events, may be sequentially composed.

The standard approach to giving meaning to processes in CCS is to define equivalence relations based on bisimulation. ACP, on the other hand, takes the axiomatic approach. CSP is an algebra for refinement, which embeds a notion of divergence (livelock), and does not distinguish processes based on the points in which internal choices are resolved. In the remainder of this section we focus our discussion on denotational discrete-time semantics for CSP that follow these principles. CSP also has well established algebraic and operational semantics as reported in [38], which are beyond the scope of this paper.

Several semantic models have been considered for reasoning for $tock$-CSP. However, they either do not contemplate timestops [3,22,31], required for the definition of deadlines, termination [22] and Zeno behaviour, or do not preserve the failures-based semantics of CSP within each time unit [22,32] as expected of $tock$-CSP processes. The latter is formally captured by the following property.

Property 1 $(\forall s : traces(P) \bullet tock \notin \operatorname{ran} s) \implies timed[\![P]\!] = [\![P]\!]_{\mathcal{F}}$

It ensures that for every process P where time is not advancing, characterised by requiring that $tock$ is not in the range (ran) of every sequence s in the $traces$ semantics of P, its timed semantics $timed[\![P]\!]$ is exactly the same as its failures semantics $[\![P]\!]_{\mathcal{F}}$. (We observe that the set of failures $[\![P]\!]_{\mathcal{F}}$ is a set of pairs, including a trace and a refusal. The timed models define different forms of observation, mostly sequences including refusals. The equality above is, therefore, strictly speaking an abuse of notation. In Property 1, we assume that either the traces or the failures are encoded in a way to allow a direct comparison between the models.)

If we use a model that does not allow timestops (and so cannot capture deadlines), there is no process P whose set $traces(P)$ of traces satisfies the antecedent of Property 1. So, it holds by vacuity. Such a model is not satisfactory either as a basis to reason with $tock$-CSP, and so we also consider the following property.

Property 2 *There is a process P for which* $\forall s : traces(P) \bullet tock \notin \operatorname{ran} s$.

In Sect. 5.15 we show that both properties hold for our $tock$-CSP semantics, named $\checkmark$-$tock$. In the remainder of this section we establish that existing $tock$-CSP models are unsatisfactory by failing to satisfy Property 1 or Property 2.

In Table 1 we provide a comparison of different semantics for $tock$-CSP. The first two rows indicate whether a semantic model captures termination, deadlines, and Zeno behaviour: only failures [38], refusal testing [28,34], discrete-time refusal traces [32] and $\checkmark$-$tock$ provide full support, satisfying Property 2. Subsequent rows record, for examples that we discuss next,

Table 1 Comparison of semantic models using *tock*

Semantic model	Stable failures [38]	Refusal testing [28,34]	Discrete-time refusal testing [31]	Discrete-time refusal traces [32]	Discrete-time failures [3]	Discrete-time refusals [22]	Timed testing [22]	✓-tock
Termination	✓	✓	✓	✓	✓	×	×	✓
Deadlines	✓	✓	×	✓	×	×	×	✓
Zeno	✓	✓	×	✓	×	×	×	✓
$R \not\sqsubseteq S$ (Example 1)	×	✓	✓	✓	✓	–	–	✓
$\mathcal{I}[R] \sqsubseteq \mathcal{I}[S]$ (Example 1)	✓	×	×	×	✓	–	–	✓

✓ indicates support (or that a refinement holds), and × otherwise. The symbol – indicates that a particular example cannot be specified

whether a refinement holds. $P \sqsubseteq Q$ indicates that P is refined by Q (and $P \not\sqsubseteq Q$ that it is not).

Semantic models capture the kinds of tests that an observer may perform on a process to probe its behaviour. In the stable-failures model it is possible to test, after each sequence of interactions, whether an event is refused. In a timed setting, each such test can be repeated as time progresses, with refusals recorded over time. This is required to ensure liveness over time, which is at the core of timewise refinement. We consider the following example to illustrate this point.

Example 1 $R = (a \rightarrow \mathbf{Skip} \mathrel{\square} b \rightarrow \mathbf{Skip}) \sqcap \mathbf{Stop}$
$S = a \rightarrow \mathbf{Skip} \sqcap \mathbf{Stop}$

Process R makes an internal choice ($\sqcap$). It may offer events a and b in an external choice ($\square$), followed by termination (**Skip**), or deadlock (**Stop**). The operator $\rightarrow$ is prefixing. Similarly, S may offer a or deadlock. In the failures model of CSP, R is refined by S. Although S never offers b, R could also refuse it because of **Stop**.

We consider, however, Example 1 in a timed setting, 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, and the event a is accepted, the observer can conclude that the experiment is with S, not R. If R refuses b early on, it behaves as **Stop** and does not later accept a. S presents a behaviour that is not possible for R, and so $R \sqsubseteq S$ should not hold. However, the failures model is not rich enough to disallow such a refinement, since 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.

To capture timewise refinement, Schneider [40, p. 457] suggested the refusal testing model [28,34], where refusals are recorded at the end of a trace, and also before each event. However, while in that model $R \not\sqsubseteq S$ as required, it is incompatible with a view where within each time unit we have a failures-based semantics, thus violating Property 1. To illustrate this we consider a context $\mathcal{I}$.

Example 2 $\mathcal{I}[P] = P \mathbin{\triangle_1} c \rightarrow \mathbf{Skip}$

Process $\mathcal{I}[P]$ behaves as P initially, and after exactly one time unit offers c, followed by termination. Here $\triangle_d$ is a strict timed interrupt that can be specified in *tock*-CSP in terms of

Table 2 The operators of *tock*-CSP

Operator	Name
div	Divergence
Skip	Termination
Wait n	Delay
Stop	Timed deadlock
$\mathbf{Stop}_U$	Timestop (untimed deadlock)
$g \mathbin{\&} P$	Guarding
$e \longrightarrow P$	Timed event prefixing
$P \sqcap Q$	Internal choice
$P \mathbin{\Box} Q$	Time-synchronising external choice
$P \mathbin{;} Q$	Sequential composition
$P \mathbin{\triangle} Q$	Time-synchronising interrupt
$P \mathbin{\triangle_d} Q$	Strict timed interrupt
$P \mathbin{[\![X]\!]} Q$	Time-synchronising parallel composition
$P \setminus X$	Hiding
$P[f]$	Renaming

basic operators. Since R is refined by S in the failures model, we expect $\mathcal{I}[R]$ to be refined by $\mathcal{I}[S]$ in a timed semantics as the behaviour during the first time unit is consistent with its failures semantics.

The semantics for *tock*-CSP in [40] and [31,32], based on refusal testing [28,34], however, are strictly more discriminating than failures: the refinement $\mathcal{I}[R] \sqsubseteq \mathcal{I}[S]$ does not hold. This is because refusal testing records what may be refused before each event, not only before *tock* events. Models with similar trace structures, such as discrete-time refusal testing [31] and discrete-time refusal traces [32], also fail to identify the refinement $\mathcal{I}[R] \sqsubseteq \mathcal{I}[S]$, and thus do not satisfy Property 1.

On the other hand, $\mathcal{I}[R] \sqsubseteq \mathcal{I}[S]$ holds in the discrete-time failures model [3] as required, because refusals are only recorded at the end of a trace and before *tock* events. However, in that model it is not possible to specify deadlines and Zeno behaviour, violating Property 2. In addition, processes R and S cannot be specified in the discrete-time refusals and timed testing models [22] because they may terminate, and those models do not consider termination.

In summary, semantics based on refusal testing are not compatible with the view of *tock*-CSP as a language with a failures semantics within each time unit (Property 1). The discrete-timed failures and the timed testing models are closest in identifying the required refinements. However, the former does not admit timestops, that is, it does not satisfy Property 2, or allow Zeno behaviour, while the latter, does not, in addition, handle termination.

To fully capture the expressive power of *tock*-CSP and endow it with a semantics compatible with the paradigm of stable failures in a time unit, we need a model that is as discriminating as discrete timed-failures, but which admits the specification of deadlines, termination, and Zeno behaviour. Despite the possibility to reuse the operational semantics of CSP to analyse processes in several of the models surveyed here using specialised timed operators and model-shifting [23] in FDR, for example, it remains, to the best of our knowledge, that no denotational semantics has been proposed that fully meets the identified criteria.

3 *tock*-CSP

Effectively, *tock*-CSP is CSP with the special event *tock* that marks the passage of time. In CSP behaviours are specified by processes using operators. In Sect. 3.1, we provide an overview of the operators of *tock*-CSP. In Sect. 3.2, we show how deadlines and Zeno behaviour can be modelled.

3.1 Operators

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 2.

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.

The timed prefixing operator ($e \rightarrow P$) 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 events are instantaneous. If e is $tock$, this operator allows a nondeterministic but nonzero number of time units to pass before P starts.

The next operator we consider is the delay operator of timed CSP, **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 **Wait** 0 equivalent to **Skip**.

Timed deadlock (**Stop**), waits in a stable state, refusing all events except for $tock$. The timestop ($\mathbf{Stop}_U$) refuses all events and also timelocks, refusing $tock$. This is included since the ability to stably refuse the passage of time is an important feature of *tock*-CSP and can be used to specify deadlines.

We illustrate the specification of deadlines with the example shown below.

Example 3

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

We define 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. 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* followed by a timed deadlock. At any point this behaviour may be interrupted by the event *obs*; we specify this using the time-synchronising interrupt operator ($P \bigtriangleup Q$). 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.

Following *obs* in C, there is 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 on *halt* by allowing time to pass for up to s time units (**Wait** s), then behaving as $\mathbf{Stop}_U$ to prevent further time from passing while waiting for $halt$ to occur (making $halt$ urgent). The process C captures a bounded-response time property, typical of timed systems.

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.

In addition to the time-synchronising interrupt operator, we also include the strict timed interrupt operator used in Example 2. The process $P \triangle_d Q$ behaves as P until either P terminates, in which case the process as a whole terminates, or d time units have passed, in which case it immediately behaves as Q. The operator is strict because it does not allow further execution in P after the specified time has passed, including not allowing P to terminate.

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.

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 until the other process 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, that is, events e are renamed to $f(e)$. 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.

In addition to the operators described above, processes can also be defined recursively. For that, we use equations $P = F(P)$, where P is a process name used in the process expression F to refer to recursive calls. We also allow mutual recursion defined by sets of such equations as expected and just like in CSP.

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

3.2 Deadlines and Zeno behaviour

As illustrated by the previous example, $tock$-CSP allows the specification of deadlines using timestops, that is, $\mathbf{Stop}_U$. 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 algebraically ($P \blacktriangleright d$) as follows.

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

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

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, and so, it is important to be able to express such behaviour and prove its absence.

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

Z offers to perform the event a immediately, followed by b and then recurses. 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.

Next, we describe the semantic model $\checkmark$-$tock$, giving the healthiness conditions that processes are required to fulfil. In Sect. 5, we present the formal semantics of the operators described informally in this section.

4 Semantic model

We present a new denotational model for $tock$-CSP, which we call $\checkmark$-$tock$. We define it and describe its healthiness conditions. Afterwards, in Sect. 5, we present the semantics of the operators of $tock$-CSP. The mechanisation of the model and operators is discussed in Sect. 6 and can be found in full in [5].

We define the $\checkmark$-$tock$ semantics of $tock$-CSP in terms of a given set Σ of events specific to the model. To Σ we add two events that have a special role in the model: $\checkmark$, which signals termination of a process, and $tock$, which signals the passage of time. We refer to Σ with these special events added as $\Sigma^{\checkmark}_{tock}$. We make use of the mathematical notation of Z [42] in the definition below, and throughout this section and the next. We explain the more unusual aspects of the notation where they are first used. In particular, $==$ is used to introduce a definition.

Definition 2 $\Sigma^{\checkmark}_{tock} == \Sigma \cup \{\checkmark, tock\}$

The semantics of each $\checkmark$-$tock$ 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 $\Sigma^{\checkmark}_{tock}$. We define the type Obs using a Z algebraic datatype definition (free type), introduced with ::=. We write $\mathbb{P}\,A$ for the power set of a set A.

Definition 3 $Obs ::= evt\langle\!\langle \Sigma^{\checkmark}_{tock} \rangle\!\rangle \mid ref\langle\!\langle \mathbb{P}\,\Sigma^{\checkmark}_{tock} \rangle\!\rangle$

The functions evt and ref are the constructors of the type Obs.

We place constraints on the structure of the sequences of observations that form the semantics of a $\checkmark$-$tock$ process, defining traces of the $TickTockTrace$ type below. This definition is given using a Z notation set comprehension, which in general is of the form $\{Decls \mid Pred \bullet Expr\}$. $Decls$ is a list of declarations of variables used in the set comprehension, with their types. The optional $Pred$ is a predicate constraining the values of the variables introduced by $Decls$. The optional $Expr$ is an expression mapping the values of the variables into the values contained in the set. If $Expr$ is omitted, the set is taken to contain tuples made up of the values of the variables introduced by $Decls$. This is the case in the definition below for $TickTockTrace$ (the $\bullet$ here is part of the universal quantifier syntax). It is a set of sequences of Obs constrained by a predicate with three conjuncts.

Table 3 The healthiness conditions of $\checkmark$-*tock*

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

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

Constraint (1) is that a $\checkmark$ may occur only at the end of a trace, since $\checkmark$ signals termination. The expression $\#t$ denotes the size of the sequence t, so the index i refers to an element before the end of the sequence. Constraint (2) states that any refusal set that occurs before the end of a trace must be followed by a $tock$. The operator ran denotes the range of a function, so $\operatorname{ran} ref$ is the set of refusals in Obs and i is thus selected to be the index of a refusal in t. 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 that does not include $tock$. The function $ref^{\sim}$ denotes the inverse of ref, which extracts the refusal set from an element of Obs in the range of ref.

By contrast, refusals are optional at the end of a trace. We take the presence of a refusal at the end of a trace to indicate stability, and its absence to signify possible instability. Constraint (3) thus requires stability wherever $tock$ can occur, but recording stability via a final refusal in a trace does not require a $tock$ event.

Each $\checkmark$-*tock* process is represented by a subset of $TickTockTrace$. However, not all such sets characterise a valid process. We define four healthiness conditions that $\checkmark$-*tock* processes satisfy, shown in Table 3. 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 relation for sequences of observations, $\lesssim$, defined inductively below. The base case states that the empty trace ($\langle\rangle$) is a prefix of every trace t. The second case states that, when two traces start with the same event e, the first trace ($\langle evt\ e\rangle \frown s$) is a prefix of the second trace ($\langle evt\ e\rangle \frown t$) whenever the traces after e are prefix related ($s \lesssim t$). We use $\langle$ and $\rangle$ to delimit sequences, so that $\langle evt\ e\rangle$ is the singleton sequence with element $evt\ e$. The operator $\frown$ is sequence concatenation. Finally, the third case states that, for refusals X and Y, traces $\langle ref\ X\rangle \frown s$ and $\langle ref\ Y\rangle \frown t$ are prefix related when $s \lesssim t$ and X is a subset of Y (rather than requiring the refusals to be the same). We thus have that $t_1 \lesssim t_2$ if t_1 is obtained from a prefix of t_2 by possibly replacing some or all refusals with a subset.

Definition 4

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

The healthiness condition **TT1** thus imposes prefix and subset closure: given any trace ρ of a healthy process, any prefix of ρ is also a trace of that process. This corresponds to the prefix and subset closure conditions of the stable failures semantics for CSP, but accounts for the fact that refusals occur before $tock$ events. A consequence of **TT1** is that there must be a prefix of the trace of events before a $tock$ that ends with a refusal, indicating stability.

The remaining two healthiness conditions constrain the contents of refusal sets. 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 condition of the stable failures model that says that events that cannot be performed must be refused but, as with **TT1**, **TT2** handles the fact that refusals occur throughout the trace. **TT2** applies only where a refusal already occurs in a trace, so it does not require the inclusion of a refusal where instability occurs.

The final healthiness condition, **TT3**, states that anywhere a refusal occurs, the $\checkmark$ event must also be refused. This follows from the fact that termination occurs unstably, and so no refusal can be observed if $\checkmark$ is not refused. However, **TT3** does not exclude nondeterministic processes such as **Skip** $\sqcap\ a \rightarrow$ **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 $\checkmark$ is refused.

We observe that our treatment of termination is slightly different from, but consistent with, the failures semantics of CSP as presented in [38]. We consider termination to be unstable because it can happen without the agreement of the environment, and a process that is ready to terminate cannot delay termination (indefinitely). A crucial law of (untimed) CSP that captures the nature of termination is **Skip** $\Box\ P =$ (**Skip** $\Box\ P) \sqcap$ **Skip**. It states that a process **Skip** $\Box\ P$, which has the possibility to terminate, can do so without the control of the environment, as expressed by the internal choice. In the untimed model, there is no notion of instability, and so the uncontrollability of termination is captured by an internal choice. In $\checkmark$-$tock$, with a definition of **Skip** that does not record termination as unstable, this law does not hold. On the other hand, in $\checkmark$-$tock$ this law does not equate the (unstable) choice of termination with a (stable) internal choice, since termination, which is available in both processes in the internal choice, is unstable.

We define the semantics of a process P using a function $tt[\![P]\!]$, which gives the set of traces corresponding to P. This is always a healthy subset of $TickTockTrace$, irrespective of the particular process P under consideration. In the next section, we define $tt[\![P]\!]$ for every P, and show that our definitions indeed characterise healthy sets of $\checkmark$-$tock$ traces. Similarly to other semantic models of CSP, refinement in $\checkmark$-$tock$ is subset inclusion. It captures timewise refinement.

Definition 5 (*Refinement*) $P \sqsubseteq Q == tt[\![Q]\!] \subseteq tt[\![P]\!]$

A process Q refines P exactly when every trace of Q is also a trace of P. Following from **TT0-1** we have that the empty sequence is a trace of every process P, and so $\{\langle\rangle\}$ refines every $\checkmark$-$tock$ process. We also have that $TickTockTrace$ satisfies all the healthiness conditions.

It defines the semantics of **Chaos**, the process that may nondeterministically perform any event or deadlock, and which every process refines. Therefore, we have a complete lattice under the refinement order where $\top = \{\langle\rangle\}$, $\bot =$ **Chaos**, and the greatest lower bound $\sqcap$ is set union. The top ($\top$) is also the definition of **div**, and the greatest lower bound coincides with the semantics of internal choice, as shown in the next section.

Next, we give the semantics of the operators of $\checkmark$-$tock$, which satisfy the healthiness conditions described here, as proved in our mechanisation (see Sect. 6).

5 Operator semantics

Below, we give the semantics of the operators described in Sect. 3.

5.1 Divergence

The simplest $\checkmark$-$tock$ process 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.

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

We note that the process **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.

5.2 Termination

The process **Skip**, which terminates immediately, has the semantics shown below.

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

Similarly to **div**, it contains no refusals, since termination is unstable: 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.

5.3 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 $\checkmark$-$tock$.

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

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

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

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 $\Sigma^{\checkmark}_{tock} \setminus \{tock\}$.

5.4 Timestop

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 $\Sigma^{\checkmark}_{tock}$.

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

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

5.5 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 with at most n *tock* events, and with refusals drawn from $\Sigma^{\checkmark}$; they are specified using *tocks*. 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.

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

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 ✓, 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. We present the traces as a union of sets, corresponding to the union of sets presented in the definition above. Since ✓-*tock* processes are prefix and subset closed (healthiness condition **TT1**), we generally show only maximal traces within each set, except where we need to illustrate particular aspects of the definition. In this example, it is helpful to present traces with both one and two *tock* events, since the definition of **Wait** 2 treats them differently, but we show only maximal refusals within each trace and omit the corresponding traces with zero *tock* events. We indicate that (non-maximal) traces have been omitted in each set using ellipsis (...). Additionally, for brevity, we omit the constructors *evt* and *ref* in examples.

$$\begin{aligned} &tt[\![\mathbf{Wait}\ 2]\!] = \\ &\quad \{\langle\{a, b, c, \checkmark\}, tock\rangle, && (7) \end{aligned}$$

$$
\begin{aligned}
&\langle \{a, b, c, \checkmark\}, tock, \{a, b, c, \checkmark\}, tock\rangle,\ \ldots\} && (8)\\
&\cup \{\langle \{a, b, c, \checkmark\}, tock, \{a, b, c, \checkmark\}\rangle,\ \ldots\} && (9)\\
&\cup \{\langle \{a, b, c, \checkmark\}, tock, \{a, b, c, \checkmark\}, tock, \checkmark\rangle,\ \ldots\} && (10)
\end{aligned}
$$

The traces of **Wait** 2 in the set on lines (7) and (8) are traces of $tocks\ \Sigma^{\checkmark}$ that contain two or fewer *tock* events, contributed by set (4). Refusals are not included at the end of these traces, but set (5) does include corresponding traces with refusals that are subsets of $\Sigma^{\checkmark}$ for any traces with strictly fewer than two *tock* events. Set (5) thus contributes the trace on line (9), which corresponds to the trace on line (7). There is no trace in set (5) corresponding to the trace on line (8), since it contains exactly two *tock* events. The trace on line (9) represents the stability before a *tock*, 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 (that is, the trace shown on line (8)) with a ✓ event appended. We note that there is no trace with a refusal after 2 *tock* events, ensuring that stability cannot be observed before ✓.

5.6 Timed event prefixing

The semantics of a timed event prefixing, $e \rightarrow P$ is 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 their refusals do not include the event e.

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

The last two sets, (13) and (14), contain traces consisting of *tock* events followed by an occurrence of e, 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 P. The second, (14), represents the case where e is *tock*, and inserts a refusal set before e. We observe that when the e is *tock*, at least one *tock* event must occur, so it is included in the trace separately to the events from *tocks*.

Example 6 Assuming again $\Sigma = \{a, b, c\}$, the traces of $a \rightarrow$ **Stop** are below. In this and in the following examples, sequences of *tock* events from *tocks* are included. Since these sequences can be of any length, we just show a single *tock* event where such a sequence can occur, omitting longer sequences.

$$
\begin{aligned}
&tt[\![a \rightarrow \mathbf{Stop}]\!] = \\
&\quad \{\langle \{b, c, \checkmark\}, tock\rangle,\ \ldots\} && (15)\\
&\quad \cup \{\langle \{b, c, \checkmark\}, tock, \{b, c, \checkmark\}\rangle,\ \ldots\} && (16)\\
&\quad \cup \{\langle \{b, c, \checkmark\}, tock, a\rangle, && (17)\\
&\qquad \langle a, \{a, b, c, \checkmark\}, tock\rangle,\ \ldots\} && (18)\\
&\quad \cup \{\} && (19)
\end{aligned}
$$

The trace on line (15) is contributed by set (11). It consists of traces of $tock$ events, with refusal of every event except a and $tock$. Similarly, the trace on line (16) is contributed by set (12). It is the trace on line (15) with a refusal of every event except a and $tock$ appended. The third set (13) contributes the traces on lines (17) and (18). On line (17), the trace from line (15) is followed by a. After a, the traces of **Stop** are appended, as shown on line (18). Set (14) does not contribute any traces in this case [line (19)], since $a \neq tock$.

5.7 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.

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

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 $r \frown q$ of P and Q, constrained by several conditions. The common prefix r is from $tocks\ \Sigma^{\checkmark}_{tock}$ and 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 by conditions (21) and (22), since the $tock$ events before the choice is resolved must be synchronised.

$$\begin{aligned}
tt[\![P \,\Box\, Q]\!] = \{&r : tocks\ \Sigma^{\checkmark}_{tock};\ p, q, t : TickTockTrace \mid \\
&r \frown p \in tt[\![P]\!] \land r \frown q \in tt[\![Q]\!] \land && (20)\\
&(\forall r2 : tocks\ \Sigma^{\checkmark}_{tock} \bullet r2r \frown p \implies r2r) \land && (21)\\
&(\forall r2 : tocks\ \Sigma^{\checkmark}_{tock} \bullet r2r \frown q \implies r2r) \land && (22)\\
&(\forall X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet p = \langle ref\ X \rangle \implies \\
&\quad \exists Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet q = \langle ref\ Y \rangle \land X \setminus \{tock\} = Y \setminus \{tock\}) \land && (23)\\
&(\forall X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet q = \langle ref\ X \rangle \implies \\
&\quad \exists Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet p = \langle ref\ Y \rangle \land X \setminus \{tock\} = Y \setminus \{tock\}) \land \\
&(t = r \frown p \lor t = r \frown q) \bullet t\} && (24)
\end{aligned}$$

The refusals 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 (23) and (24). 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. In this case 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 are either empty or start with a non-$tock$ event, since $tock$ events at the start of p and q are ruled out by conditions (21) and (22). These traces are not constrained since a non-$tock$ event resolves the choice. We recall that $tock$ is absent from refusals before $tock$ events, so this handling of $tock$ in refusals does not need to be applied to r.

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

$$\begin{aligned}
&tt[\![a \to \mathbf{Stop} \,\Box\, b \to c \to \mathbf{Stop}]\!] = \{ \\
&\langle \{c, \checkmark\}, tock, \{c, \checkmark\} \rangle, && (25)
\end{aligned}$$

$$\langle \{c, \checkmark\}, tock, a\rangle,\ \langle a, \{a, b, c, \checkmark\}\rangle, \tag{26}$$
$$\langle \{c, \checkmark\}, tock, b\rangle,\ \langle b, \{a, b, \checkmark\}\rangle,\ \ldots$$
$$\} \tag{27}$$

The traces of $a \rightarrow \mathbf{Stop} \,\Box\, b \rightarrow c \rightarrow \mathbf{Stop}$ begin with the initial traces of $tock$ events common to both $a \rightarrow \mathbf{Stop}$ and $b \rightarrow c \rightarrow \mathbf{Stop}$. 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 each process. This can be seen in the trace on line (25), and in the first traces on lines (26) and (27). The conditions (21) and (22) specify the intersection. The second traces on lines (26) and (27) begin with an empty trace of $tock$ events, which is also permitted.

For those traces of one of the processes in the choice that have a refusal after an initial trace of $tock$ events, the conditions (23) and (24) 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 processes are defined using event prefixing and so both have traces of $tock$ events ending in refusals. These refusals are intersected, with the exception of $tock$ events, yielding the trace on line (25).

Any traces beyond the initial sequence of $tock$ events and a single refusal are included without restriction. In our example, traces of $a \rightarrow \mathbf{Stop}$ are shown on line (26), and of $b \rightarrow c \rightarrow \mathbf{Stop}$ on line (27).

Example 8 This example considers a process similar to that presented in Example 7 but where one of the processes in the choice is a timestop ($\mathbf{Stop}_U$), to illustrate how external choice behaves when one of the operands does not allow time to pass.

$$tt[\![\mathbf{Stop}_U \,\Box\, b \rightarrow c \rightarrow \mathbf{Stop}]\!] = \{$$
$$\langle \{a, c, \checkmark, tock\}\rangle, \tag{28}$$
$$\langle b, c, \{a, b, c, \checkmark\}\rangle, \tag{29}$$
$$\langle b, \{a, b, \checkmark\}, tock, c, \{a, b, c, \checkmark\}\rangle, \tag{30}$$
$$\langle b, c, \{a, b, c, \checkmark\}, tock, \{a, b, c, \checkmark\}\rangle,\ \ldots$$
$$\} \tag{31}$$

Timestop contributes an initial maximal refusal that refuses everything, which is matched by refusals from the prefix of b that refuse everything except b and $tock$. Since lines (23) and (24) of the definition of external choice require the events of the refusal to be the same except for $tock$, the maximal initial refusal of $\mathbf{Stop}_U \,\Box\, b \rightarrow c \rightarrow \mathbf{Stop}$ thus contains everything except b, as shown on line (28).

Since timestop does not contain any initial sequences of $tock$ events, none of the traces of $\mathbf{Stop}_U \,\Box\, b \rightarrow c \rightarrow \mathbf{Stop}$ can begin with a $tock$ event. Traces of $b \rightarrow c \rightarrow \mathbf{Stop}$ that do not begin with a $tock$ event are included, some of which are shown on lines (29), (30) and (31). In particular, $tock$ is permitted after an occurrence of b [line (30)] and after an occurrence of c [line (31)].

An external choice with $\mathbf{Stop}_U$ thus has the effect of removing initial $tock$ events and adding $tock$ to initial refusals, making the choice urgent.

5.8 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 of P that do not end with $\checkmark$. These traces represent the behaviour of P before termination.

$$
\begin{aligned}
& tt[\![P\ ;\ Q]\!] = \\
& \quad \{p : tt[\![P]\!] | \neg(\exists t : TickTockTrace \bullet p = t \frown \langle evt\ \checkmark\rangle)\} \\
& \quad \cup \{p, q : TickTockTrace | p \frown \langle evt\ \checkmark\rangle \in tt[\![P]\!] \land q \in tt[\![Q]\!] \bullet p \frown q\}
\end{aligned}
$$

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

5.9 Time-synchronising interrupt

To define the semantics of time-synchronising interrupt, $P \triangle Q$, we need a function to project the $tock$ events and their associated refusals 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 below.

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

For the empty trace $\langle\rangle$, $filterTocks$ results in $\langle\rangle$. For a trace with a single refusal, we also get $\langle\rangle$, since such a refusal does not have an associated $tock$ event. For 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 $tock$, the refusal and $tock$ event are retained, and followed by the result of applying $filterTocks$ to the rest of the trace.

The semantics of time-synchronising interrupt is defined, using $filterTocks$, as the union of three sets, as shown below. The first set, (32), contains the traces $p \frown \langle evt\ \checkmark\rangle$ from P that end in $\checkmark$. In this case, 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, (33), 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 and ending 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.

$$
\begin{aligned}
& tt[\![P \triangle Q]\!] = \\
& \quad \{p : TickTockTrace; q : tt[\![Q]\!] | \\
& \quad\quad p \frown \langle evt\ \checkmark\rangle \in tt[\![P]\!] \land filterTocks\ p = q \bullet p \frown \langle evt\ \checkmark\rangle\} \qquad (32) \\
& \quad \cup \{p, q : TickTockTrace; X, Y, Z : \mathbb{P}\, \Sigma^{\checkmark}_{tock} | \\
& \quad\quad p \frown \langle ref\ X\rangle \in tt[\![P]\!] \land q \frown \langle ref\ Y\rangle \in tt[\![Q]\!] \land
\end{aligned}
$$

$$\begin{aligned}&filterTocks\ p = q \wedge Z \subseteq X \cup Y \wedge X\setminus\{tock\} = Y\setminus\{tock\} \bullet \\ &p \frown \langle ref\ Z\rangle\} \qquad (33)\\ \cup\, &\{p : tt[\![P]\!]; q1, q2 : TickTockTrace | \\ &(\neg\exists r : \mathrm{seq}\ Obs \bullet p = r \frown \langle evt\ \checkmark\rangle) \wedge \\ &(\neg\exists r : \mathrm{seq}\ Obs; X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet p = r \frown \langle ref\ X\rangle) \wedge \\ &filterTocks\ p = q1 \wedge q1 \frown q2 \in tt[\![Q]\!] \wedge \\ &(\neg\exists r : \mathrm{seq}\ Obs; X : \mathbb{P}\,\Sigma^{\checkmark}_{tock} \bullet q2 = \langle ref\ X\rangle \frown r) \bullet p \frown q2\} \qquad (34)\end{aligned}$$

Finally, the third set, (34), considers the traces p of P that end in neither ✓ nor a refusal. These traces are required to be matched by traces $q1 \frown q2$ from Q, where $q1$ is the trace of *tock* events corresponding to p. It is required that $q2$ does not start with a refusal, since refusals at the end of a trace are considered in (33) and refusals before a *tock* must be synchronised (and hence should occur in $q1$). The traces in (34) are made up of the concatenation of p and $q2$, representing the behaviour of P before interruption followed by the behaviour of Q after interruption. Since $q2$ can be empty, (34) can include traces just from P.

Example 9 We revisit Example 7, but use an interrupt rather than a choice.

$$\begin{aligned}&tt[\![a \rightarrow \mathbf{Stop} \vartriangle b \rightarrow c \rightarrow \mathbf{Stop}]\!] = \\ &\quad \{\} \qquad (35)\\ &\quad \cup \{\langle\{c, \checkmark\}\rangle,\ \langle a, \{a, c, \checkmark\}\rangle, \qquad (36)\\ &\qquad \langle\{c, \checkmark\}, tock, \{c, \checkmark\}\rangle,\ \ldots\} \qquad (37)\\ &\quad \cup \{\langle\{c, \checkmark\}, tock, a\rangle, \qquad (38)\\ &\qquad \langle a, \{a, c, \checkmark\}, tock\rangle, \qquad (39)\\ &\qquad \langle b, \{a, b, \checkmark\}, tock, c\rangle,\ \langle b, c, \{a, b, \checkmark\}, tock\rangle, \qquad (40)\\ &\qquad \langle a, b, \{a, b, \checkmark\}, tock, c\rangle, \qquad (41)\\ &\qquad \langle a, \{a, c, \checkmark\}, tock, b, c\rangle,\ \ldots\} \qquad (42)\end{aligned}$$

There are no traces contributed by the first set, (32), as shown on line (35), since $a \rightarrow$ **Stop** does not have any traces ending with a ✓ event.

The second set, (33), contributes the traces on lines (36) and (37). These are traces of $a \rightarrow$ **Stop** that end with a refusal and have a corresponding trace of *tock* events ending with a refusal in $tt[\![b \rightarrow c \rightarrow \mathbf{Stop}]\!]$. Examples of relevant traces of $a \rightarrow$ **Stop** are those consisting of a single refusal (such as $\langle\{b, c, \checkmark\}\rangle$), those consisting of a *tock* followed by a refusal (such as $\langle\{b, c, \checkmark\}, tock, \{b, c, \checkmark\}\rangle$, and those that have an event a followed by a refusal (such as $\langle a, \{a, b, c, \checkmark\}\rangle$).

As explained, the sequence of *tock* events for each trace of $a \rightarrow$ **Stop** is obtained by applying $filterTocks$ to the trace before the final refusal. For those traces consisting of a single refusal or a followed by a refusal, the sequence of *tock* events is empty. Those are matched by traces of $b \rightarrow c \rightarrow$ **Stop** that consist of a single refusal (such as $\langle\{a, c, \checkmark\}\rangle$). The traces of $a \rightarrow$ **Stop** whose final refusals agree with the refusals of these traces of $b \rightarrow c \rightarrow$ **Stop** (in all but *tock* events) are those indicated on line (36). Since refusals are prefix closed, this has the effect of intersecting the refusals (with the exception of *tock*).

For the traces of $a \rightarrow$ **Stop** with a single *tock* event followed by a refusal, the corresponding sequences of *tock* events are those consisting of the single *tock* event with the refusal preceding it. We thus consider the traces in $b \rightarrow c \rightarrow$ **Stop** with a single *tock* event followed by a refusal: $\langle\{a, c, \checkmark\}, tock, \{a, c, \checkmark\}\rangle, \langle\{a, c, \checkmark\}, tock, \{c, \checkmark\}\rangle, \langle\{c, \checkmark\}, tock, \{c, \checkmark\}\rangle, \ldots$.

The corresponding traces from $a \rightarrow \mathbf{Stop}$ are required to agree with these in the refusals preceding the $tock$ event, and the refusals at the end are required to agree in all but $tock$ events. Since the refusals in the traces from $b \rightarrow c \rightarrow \mathbf{Stop}$ do not include b, we cannot include the trace $\langle\{b, c, \checkmark\}, tock, \{b, c, \checkmark\}\rangle$, but the trace shown on line (37) is included since it is present in the traces of both processes.

The traces contributed by the third set, (34), are those on lines (38) to (42). Those on lines (38) and (39) are from $a \rightarrow \mathbf{Stop}$ and do not end in a $\checkmark$ or a refusal. As with the traces from (33)), their $tock$ events and corresponding refusals are matched by a trace of $tock$ events from $b \rightarrow c \rightarrow \mathbf{Stop}$. The filtering of $tock$ events means this applies to $tock$ events before [line (38)] and after a [line (39)]. The result is that b is removed from each refusal observed before a $tock$, since it is offered as the initial event of $b \rightarrow c \rightarrow \mathbf{Stop}$.

The traces on lines (40), (41) and (42) consist of traces from $a \rightarrow \mathbf{Stop}$ that do not end in $\checkmark$ or a refusal, followed by traces from $b \rightarrow c \rightarrow \mathbf{Stop}$ that do not start with a refusal. The traces from $a \rightarrow \mathbf{Stop}$ are matched by traces of $tock$ events from $b \rightarrow c \rightarrow \mathbf{Stop}$, but the traces after b are included as-is.

Example 10 This example considers the traces of the process obtained by replacing the left-hand process in Example 9 with a timestop ($\mathbf{Stop}_U$).

$$\begin{aligned}
&tt[\![\mathbf{Stop}_U \vartriangle b \rightarrow c \rightarrow \mathbf{Stop}]\!] = \\
&\quad \{\} && (43)\\
&\quad \cup \{\langle\{a, c, \checkmark, tock\}\rangle,\ \ldots\}, && (44)\\
&\quad \cup \{\langle b, \{a, b, \checkmark\}, tock, c\rangle,\ \langle b, c, \{a, b, \checkmark\}, tock\rangle,\ \ldots\} && (45)
\end{aligned}$$

As in Example 9, there are no traces from set (32), as shown on line (43).

For set (33), which contains traces from the left-hand process ending in refusals, the only traces are those consisting of a single refusal. Refusals that include b are excluded, since the refusals must agree on non-$tock$ events with $b \rightarrow c \rightarrow \mathbf{Stop}$, which does not refuse b initially. The refusals can include $tock$, since $tock$ is refused by one side, so everything except b is refused, as shown on line (44).

The traces contributed by set (34) are on line (45). Since $\mathbf{Stop}_U$ has no $tock$ events, this just consists of traces of traces of $b \rightarrow c \rightarrow \mathbf{Stop}$ without initial events. Time-synchronising interrupt with $\mathbf{Stop}_U$ on the left thus has the effect of removing initial $tock$ events, similarly to external choice.

Example 11 When the left and right-hand processes of Example 10 are swapped, time-synchronising interrupt yields different timed behaviour, as shown below.

$$\begin{aligned}
&tt[\![b \rightarrow c \rightarrow \mathbf{Stop} \vartriangle \mathbf{Stop}_U]\!] = \\
&\quad \{\} && (46)\\
&\quad \cup \{\langle\{a, c, \checkmark, tock\}\rangle,\ \langle b, \{a, b, \checkmark, tock\}\rangle,\ \langle b, c, \{a, b, c, \checkmark, tock\}\rangle,\ \ldots\}, && (47)\\
&\quad \cup \{\langle b, c\rangle,\ \ldots\} && (48)
\end{aligned}$$

As in Example 10, there are no traces from set (32), as shown on line (46).

Set (33) contributes the traces on line (47): traces from $b \rightarrow c \rightarrow \mathbf{Stop}$ that end with a refusal, with $tock$ added to the final refusal because $\mathbf{Stop}_U$ refuses $tock$. Since $tock$ events in the traces of $b \rightarrow c \rightarrow \mathbf{Stop}$ must be matched by a sequence of $tock$ events in a trace of $\mathbf{Stop}_U$ (which does not engage in any $tock$ events), no $tock$ events can occur in the traces contributed by set (33).

The traces contributed by set (34) are shown on line (48). Since $\mathbf{Stop}_U$ has no $tock$ events to match $tock$ events in $b \rightarrow c \rightarrow \mathbf{Stop}$, this just consists of traces of $b \rightarrow c \rightarrow \mathbf{Stop}$ that contain no $tock$ events and do not end with a refusal. Adding a time-synchronising interrupt with $\mathbf{Stop}_U$ on the right thus has the effect of removing all $tock$ events in the left-hand process.

5.10 Strict timed interrupt

The semantics of strict timed interrupt, $P \triangle_d Q$, is defined, as shown below, by the union of two sets. The first set, on line (49), contains the traces that record only events that occur before the deadline d. These are the traces p of P that contain fewer than d $tock$ events, specified by filtering the $tock$ events from p and restricting the length of the resulting sequence.

$$\begin{aligned}
&tt[\![P \triangle_d Q]\!] = \\
&\quad \{p : tt[\![P]\!] | \#(p \upharpoonright \{evt\ tock\}) < d\} && (49)\\
&\quad \cup \{p : tt[\![P]\!];\ q : tt[\![Q]\!];\ r : \operatorname{seq} \Sigma^{\checkmark}_{tock} | \#(p \upharpoonright \{evt\ tock\}) = d \\
&\qquad \wedge ((d = 0 \wedge p = \langle\rangle) \vee (d > 0 \wedge p = r \frown \langle evt\ tock\rangle)) \bullet p \frown q\} && (50)
\end{aligned}$$

The second set, defined on line (50), contains traces with events occurring after the deadline. In this case a trace p of P, with events up to the deadline, is concatenated with a trace q of Q, with events after the deadline. The trace p must contain exactly d $tock$ events, since it is the trace up to the deadline. Since the interrupt is immediate, no further events can occur in p after the deadline. If d is zero, p must be the empty trace. If d is greater than zero, the last event of p must be $tock$, with an arbitrary trace r before the final $tock$.

Example 12

$$\begin{aligned}
&tt[\![a \rightarrow \mathbf{Stop} \triangle_1 b \rightarrow c \rightarrow \mathbf{Stop}]\!] = \\
&\quad \{\langle\{b, c, \checkmark\}\rangle,\ \langle a, \{a, b, c, \checkmark\}\rangle,\ \ldots\} && (51)\\
&\quad \cup \{\langle\{b, c, \checkmark\}, tock, \{a, c, \checkmark\}, tock\rangle, && (52)\\
&\qquad \langle a, \{a, b, c, \checkmark\}, tock, \{a, c, \checkmark\}, tock\rangle,\ \ldots\} && (53)
\end{aligned}$$

For $a \rightarrow \mathbf{Stop} \triangle_1 b \rightarrow c \rightarrow \mathbf{Stop}$, the first set, (49), contributes the traces shown on line (51). These are the traces of $a \rightarrow \mathbf{Stop}$ that do not include a $tock$, since they occur before the deadline of one $tock$.

The second set, (50), contributes the traces on lines (52) and (53). These are made up of a trace from $a \rightarrow \mathbf{Stop}$, with exactly one $tock$, concatenated with a trace from $b \rightarrow c \rightarrow \mathbf{Stop}$. The trace from $a \rightarrow \mathbf{Stop}$ may contain just the $tock$ event and its preceding refusal (line (52)), or may have an a before the $tock$ event (line (53)). We note that a cannot occur after the first $tock$, since $tock$ must be the last element of every trace. The trace from $b \rightarrow c \rightarrow \mathbf{Stop}$ may contain additional $tock$ events, but the refusals present before them may differ from those in $a \rightarrow \mathbf{Stop}$, as illustrated by the trace on line (52).

5.11 Parallel composition

The semantics of a parallel composition $P [\![A]\!] Q$ is defined below, by merging each pair of traces from P and Q using a function $p [\![A]\!]^T q$. This 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.

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

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 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 (55)–(60) define the cases in which one trace is empty. If both traces are empty (Eq. 55), 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 (Eq. 56) 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.

Merging the empty trace with the trace consisting of a single ✓ event also yields just the empty trace (Eq. 57), since ✓ requires synchronisation, which the empty trace cannot provide. Since, however, termination of a process does not block the other process running in parallel, the merge results in the set containing the empty trace, rather than an empty set of traces.

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, no additional traces result when it is merged with such traces (Eqs. 59 and 60)).

When a trace starts with an event e that is not in A (that is, that does not require synchronisation), then merging it with the empty trace yields all the traces formed by prepending e to the traces resulting from merging the empty trace with the rest of the trace (Eq. 58). This ensures that events that do not require synchronisation can keep being performed.

Example 13 When the empty trace is merged with the traces of $b \rightarrow c \rightarrow$ **Stop** using the synchronisation set $\{c\}$, the results are as shown below. In line with the presentation of previous examples, we generally show merging with maximal traces. Where that generates no traces, we also show the result of merging with prefixes, to illustrate the variety of the merging function's behaviour.

$$\langle\rangle \;[\![\{c\}]\!]^{T}\; \langle\{a, c, \checkmark\}, tock, b\rangle = \{\} \tag{79}$$

$$\langle\rangle \;[\![\{c\}]\!]^{T}\; \langle b, \{a, b, \checkmark\}, tock\rangle = \{\} \tag{80}$$

$$\langle\rangle \;[\![\{c\}]\!]^{T}\; \langle b\rangle = \{\langle b\rangle\} \tag{81}$$

$$\langle\rangle \;[\![\{c\}]\!]^{T}\; \langle\{a, c, \checkmark\}\rangle = \{\langle\rangle\} \tag{82}$$

$$\langle\rangle \;[\![\{c\}]\!]^{T}\; \langle b, c\rangle = \{\} \tag{83}$$

Since *tock* events require synchronisation, traces containing *tock* events do not produce any traces when merged with the empty trace, as shown in equations (79) and (80). This results from the empty set in Eq. (60).

However, prefixes not containing *tock* do produce traces. This can be seen in Eq. (81), where b does not require synchronisation and so $\langle b\rangle$ is included. In Eq. (82), an empty trace is

$$
\begin{aligned}
&\forall A : \mathbb{P}\,\Sigma;\ X, Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ e, f : \Sigma;\ t, s : \mathit{TickTockTrace} \bullet \\
&\quad \langle\rangle \llbracket A \rrbracket^T \langle\rangle = \{\langle\rangle\} \land && (55)\\
&\quad \langle\rangle \llbracket A \rrbracket^T \langle \mathit{ref}\ X\rangle = \{\langle\rangle\} \land && (56)\\
&\quad \langle\rangle \llbracket A \rrbracket^T \langle \mathit{evt}\checkmark\rangle = \{\langle\rangle\} \land && (57)\\
&\quad (e \notin A \Rightarrow \langle\rangle \llbracket A \rrbracket^T (\langle \mathit{evt}\ e\rangle \frown t) = \{s : \langle\rangle \llbracket A \rrbracket^T t \bullet \langle \mathit{evt}\ e\rangle \frown s\}) \land && (58)\\
&\quad (e \in A \Rightarrow \langle\rangle \llbracket A \rrbracket^T (\langle \mathit{evt}\ e\rangle \frown t) = \{\}) \land && (59)\\
&\quad \langle\rangle \llbracket A \rrbracket^T (\langle \mathit{ref}\ X, \mathit{evt}\ \mathit{tock}\rangle \frown t) = \{\} \land && (60)\\
&\quad (X \setminus (A \cup \{\checkmark, \mathit{tock}\}) = Y \setminus (A \cup \{\checkmark, \mathit{tock}\}) \Rightarrow \\
&\qquad \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T \langle \mathit{ref}\ Y\rangle = \{\mathit{ref}\,(X \cup Y)\}) \land && (61)\\
&\quad (X \setminus (A \cup \{\checkmark, \mathit{tock}\}) \neq Y \setminus (A \cup \{\checkmark, \mathit{tock}\}) \Rightarrow \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T \langle \mathit{ref}\ Y\rangle = \{\}) \land && (62)\\
&\quad \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T \langle \mathit{evt}\checkmark\rangle = \{B : \Sigma^{\checkmark}_{tock} \mid B \subseteq A \bullet \langle \mathit{ref}\,(X \cup B)\rangle\} \land && (63)\\
&\quad (e \notin A \Rightarrow \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T (\langle \mathit{evt}\ e\rangle \frown t) = \\
&\qquad \{s : \mathit{TickTockTrace} \mid s \in \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T t \bullet \langle \mathit{evt}\ e\rangle \frown s\}) \land && (64)\\
&\quad (e \in A \Rightarrow \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T (\langle \mathit{evt}\ e\rangle \frown t) = \{\}) \land && (65)\\
&\quad \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T (\langle \mathit{ref}\ Y, \mathit{evt}\ \mathit{tock}\rangle \frown t) = \{\} \land && (66)\\
&\quad \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T \langle \mathit{evt}\checkmark\rangle = \{\langle \mathit{evt}\checkmark\rangle\} \land && (67)\\
&\quad (e \notin A \Rightarrow \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T (\langle \mathit{evt}\ e\rangle \frown t) = \\
&\qquad \{s : \mathit{TickTockTrace} \mid s \in \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T t \bullet \langle \mathit{evt}\ e\rangle \frown s\}) \land && (68)\\
&\quad (e \in A \Rightarrow \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T (\langle \mathit{evt}\ e\rangle \frown t) = \{\}) \land && (69)\\
&\quad \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T (\langle \mathit{ref}\ Y, \mathit{evt}\ \mathit{tock}\rangle \frown t) = \\
&\qquad \{Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ s : \mathit{TickTockTrace} \mid \langle \mathit{ref}\ Z\rangle \in \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T \langle \mathit{ref}\ Y\rangle \land \\
&\qquad\quad s \in \langle \mathit{evt}\checkmark\rangle \llbracket A \rrbracket^T t \bullet \langle \mathit{ref}\ Z, \mathit{evt}\ \mathit{tock}\rangle \frown s\} \land && (70)\\
&\quad (e \notin A \land f \notin A \Rightarrow (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{evt} f\rangle \frown s) = \\
&\qquad \{r : \mathit{TickTockTrace} \mid r \in t \llbracket A \rrbracket^T (\langle \mathit{evt} f\rangle \frown s) \bullet \langle \mathit{evt}\ e\rangle \frown r\} \\
&\qquad \cup \{r : \mathit{TickTockTrace} \mid r \in (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T s \bullet \langle \mathit{evt} f\rangle \frown r\}) \land && (71)\\
&\quad (e \notin A \land f \in A \Rightarrow (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{evt} f\rangle \frown s) = \\
&\qquad \{r : \mathit{TickTockTrace} \mid r \in t \llbracket A \rrbracket^T (\langle \mathit{evt} f\rangle \frown s) \bullet \langle \mathit{evt}\ e\rangle \frown r\}) \land && (72)\\
&\quad (e \in A \land f \in A \land e = f \Rightarrow (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{evt} f\rangle \frown s) = \\
&\qquad \{r : \mathit{TickTockTrace} \mid r \in t \llbracket A \rrbracket^T s \bullet \langle \mathit{evt}\ e\rangle \frown r\}) \land && (73)\\
&\quad (e \in A \land f \in A \land e \neq f \Rightarrow (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{evt} f\rangle \frown s) = \{\}) \land && (74)\\
&\quad (e \notin A \Rightarrow (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{ref}\ Y, \mathit{evt}\ \mathit{tock}\rangle \frown s) = \\
&\qquad \{r : \mathit{TickTockTrace} \mid r \in t \llbracket A \rrbracket^T (\langle \mathit{ref}\ Y, \mathit{evt}\ \mathit{tock}\rangle \frown s) \bullet \langle \mathit{evt}\ e\rangle \frown r\}) \land && (75)\\
&\quad (e \in A \Rightarrow (\langle \mathit{evt}\ e\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{ref}\ Y, \mathit{evt}\ \mathit{tock}\rangle \frown s) = \{\}) \land && (76)\\
&\quad ((\langle \mathit{ref}\ X, \mathit{evt}\ \mathit{tock}\rangle \frown t) \llbracket A \rrbracket^T (\langle \mathit{ref}\ Y, \mathit{evt}\ \mathit{tock}\rangle \frown s) = \\
&\qquad \{Z : \mathbb{P}\,\Sigma^{\checkmark}_{tock};\ r : \mathit{TickTockTrace} \mid \\
&\qquad\quad \langle \mathit{ref}\ Z\rangle \in \langle \mathit{ref}\ X\rangle \llbracket A \rrbracket^T \langle \mathit{ref}\ Y\rangle \land r \in t \llbracket A \rrbracket^T s \bullet \\
&\qquad\quad \langle \mathit{ref}\ Z, \mathit{evt}\ \mathit{tock}\rangle \frown r\} \land && (77)\\
&\quad s \llbracket A \rrbracket^T t = t \llbracket A \rrbracket^T s && (78)
\end{aligned}
$$

Fig. 1 Definition of the parallel trace merge function

produced from merging an empty trace with a refusal, since the empty trace cannot guarantee stability.

Similarly to the cases with $tock$, b followed by an occurrence of c yields no traces (Eq. 83), since c also requires synchronisation.

Equations (61)–(66) define the cases where a trace has a single refusal (and the other is not empty). When both traces are refusals, $\langle ref\ X\rangle$ and $\langle ref\ Y\rangle$, the result depends on whether X and Y agree in the events that do not require synchronisation (that is, events other than ✓, $tock$, and the events from A). Such events are refused only if they are refused by both the processes in the parallelism. If X and Y do agree in events not requiring synchronisation, then the result is a refusal formed from their union (Eq. 61), so those events that do require synchronisation are refused if they are refused by one side. When X and Y do not agree, there are no resulting traces (Eq. 62).

When a singleton trace $\langle ref\ X\rangle$ is merged with a singleton trace containing just the ✓ event (Eq. 63), the resulting traces are the singleton traces containing refusals obtained by adding subsets of the synchronisation set A to X. This is due to the fact that the presence of a refusal implies that ✓ is refused (see **TT3**). 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. The non-synchronised events of X are thus refused, since they are refused by both sides, and all the events in A are refused (including those in X), since they require the cooperation of both sides. The process that is ready to terminate allows ✓ and $tock$, so a refusal of those events is only added if they are in X. Thus, all the events in X are refused, with the addition of the events in A, of which we take subsets to ensure subset closure.

When the other trace begins with a non-✓ non-$tock$ event (Eqs. 64 and 65), the result is similar to that for the empty trace (Eqs. 58 and 59). Similarly, when we have a $tock$ (Eq. 66) the result is the set containing the empty trace, since $tock$ also requires synchronisation.

Example 14 Merging the traces $\langle\{b, c, \checkmark\}\rangle$ and $\langle\{c, \checkmark\}\rangle$ from the process $a \rightarrow$ **Stop** with traces from $b \rightarrow c \rightarrow$ **Stop**, again taking $\{c\}$ as the synchronisation set, yields the traces shown below.

$$\langle\{b, c, \checkmark\}\rangle \llbracket \{c\} \rrbracket^T \langle\{a, c, \checkmark\}, tock, b\rangle = \{\} \tag{84}$$

$$\langle\{b, c, \checkmark\}\rangle \llbracket \{c\} \rrbracket^T \langle b, \{a, b, \checkmark\}, tock\rangle = \{\} \tag{85}$$

$$\langle\{b, c, \checkmark\}\rangle \llbracket \{c\} \rrbracket^T \langle\{a, c, \checkmark\}\rangle = \{\} \tag{86}$$

$$\langle\{c, \checkmark\}\rangle \llbracket \{c\} \rrbracket^T \langle\{c, \checkmark\}\rangle = \{\langle\{c, \checkmark\}\rangle\} \tag{87}$$

$$\langle\{b, c, \checkmark\}\rangle \llbracket \{c\} \rrbracket^T \langle b, \{a, b, \checkmark\}\rangle = \{\} \tag{88}$$

$$\langle\{b, c, \checkmark\}\rangle \llbracket \{c\} \rrbracket^T \langle b, \{b, \checkmark\}\rangle = \{\langle b, \{b, c, \checkmark\}\rangle\} \tag{89}$$

Merging a refusal with a trace containing a $tock$ event yields no traces (Eqs. 84 and 85). This is because $tock$ requires synchronisation; it matches the cases for merging the empty trace (Eqs. 79 and 80).

When both sides are traces with a single refusal, the refusals must agree in events not requiring synchronisation. Thus, in Eq. (86), 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 $a \rightarrow$ **Stop** and $b \rightarrow c \rightarrow$ **Stop**, and so is given as the resulting trace, as shown in Eq. (87).

When a refusal is merged with a trace consisting of b followed by a refusal, b is included first and the refusals are merged. Thus, in Eq. (88), 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 Eq. (89), is the trace with an occurrence of b followed by a refusal of $\{b, c, \checkmark\}$. The refusals do not have to agree in c, since it requires synchronisation. The union of the refusals from each trace is taken, so that c is included in the output refusals.

Equations (67)–(70) are the remaining cases for traces containing a single $\checkmark$ event. Since $\checkmark$ requires synchronisation, when both traces contain a single $\checkmark$ event (Eq. 67), the result is the set with that trace.

When the trace with a $\checkmark$ event is merged with a trace starting with non-$\checkmark$ non-$tock$ event e (Eqs. 68 and 69), the $\checkmark$ event cannot occur because it does not have an event to synchronise with, but the result depends on whether e requires synchronisation, with results similar to Eq. (64) and (65).

When the trace with a $\checkmark$ event is merged with a trace starting with a $tock$ event, with its refusal Y (Eq. 70), there is no $\checkmark$ event to synchronise with, but we allow time to pass while waiting for termination. The result is thus that of prepending a refusal Z and a $tock$ event to the traces formed by merging the trace with the $\checkmark$ event with the trace after the input $tock$ event. The refusal Z is drawn from the refusals formed by merging $\langle evt\ \checkmark\rangle$ with $\langle ref\ Y\rangle$.

Equations (71)–(76) define the cases where a trace begins with a non-$\checkmark$ non-$tock$ event e. The first four cover the cases where the other trace also begins with a non-$\checkmark$ non-$tock$ event f. If neither e nor f require synchronisation (Eq. 71), then the result is the union of the traces where e occurs and those where f occurs. The traces where e occurs are formed by prepending e to the result of merging the remainder of the trace with the other trace. The traces where f occurs are defined similarly. If e does not require synchronisation but f does (Eq. 72), then the result is just the traces where e occurs, followed by the result of merging the remainder of the trace with the trace that starts with f.

When both e and f require synchronisation, then we must consider whether or not they are the same event. If they are the same (Eq. 73), then they can synchronise. The result 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 (Eq. 74), then there are no possible traces, since they cannot synchronise.

When the trace starting with e is merged with a trace starting with a $tock$ (and its associated refusal), then the $tock$ requires synchronisation, which e cannot provide. The result then depends on whether e requires synchronisation. If it does not (Eq. 75), then the result is similar to that for an event f requiring synchronisation (Eq. 72). If e requires synchronisation (Eq. 76), then both sides require synchronisation, and so there are no resulting traces.

Example 15 Merging the trace $\langle a\rangle$ from $a \rightarrow$ **Stop** with the traces from the process $b \rightarrow c \rightarrow$ **Stop** yields the traces shown below.

$$\langle a\rangle \llbracket \{c\} \rrbracket^T \langle\{a, c, \checkmark\}, tock, b\rangle = \{\} \tag{90}$$

$$\langle a\rangle \llbracket \{c\} \rrbracket^T \langle b, \{a, c, \checkmark\}, tock\rangle = \{\} \tag{91}$$

$$\langle a\rangle \llbracket \{c\} \rrbracket^T \langle b, c\rangle = \{\} \tag{92}$$

$$\langle a\rangle \llbracket \{c\} \rrbracket^T \langle\{a, c, \checkmark\}\rangle = \{\langle a\rangle\} \tag{93}$$

$$\langle a\rangle \llbracket \{c\} \rrbracket^T \langle b, \{a, c, \checkmark\}\rangle = \{\langle a, b\rangle, \langle b, a\rangle\} \tag{94}$$

As in previous examples, merging with a trace containing a $tock$ or another event requiring synchronisation (Eqs. 90, 91, and 92) yields no traces, since there is no matching event in $\langle a\rangle$. When $\langle a\rangle$ is merged with a trace containing a refusal (Eq. 93), a can still occur, but there is nothing to match the refusal.

If b occurs before a refusal (Eq. 94), the a and b events can occur in either order. No further observations are possible after a and b, since the refusal after b is not matched by a corresponding refusal after a.

Finally, if both traces to be merged begin with $tock$ events, preceded by refusals X and Y (Eq. 77), 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 on their own (deferring to Eqs. 61 and 62).

Example 16 Merging traces from $a \longrightarrow \mathbf{Stop}$ and $b \longrightarrow c \longrightarrow \mathbf{Stop}$ that contain one $tock$ event yields the traces shown below.

$$\begin{array}{l}\langle\{b, c, \checkmark\}, tock\rangle \\ \quad [\![\{c\}]\!]^T \\ \langle\{a, c, \checkmark\}, tock, b\rangle\end{array} = \{\} \tag{95}$$

$$\begin{array}{l}\langle\{c, \checkmark\}, tock\rangle \\ \quad [\![\{c\}]\!]^T \\ \langle\{c, \checkmark\}, tock, b\rangle\end{array} = \{\langle\{c, \checkmark\}, tock, b\rangle\} \tag{96}$$

$$\begin{array}{l}\langle\{c, \checkmark\}, tock\rangle \\ \quad [\![\{c\}]\!]^T \\ \langle b, \{\checkmark\}, tock\rangle\end{array} = \{\langle b, \{c, \checkmark\}, tock\rangle\} \tag{97}$$

$$\begin{array}{l}\langle a, \{a, b, c, \checkmark\}, tock\rangle \\ \quad [\![\{c\}]\!]^T \\ \langle b, \{a, b, \checkmark\}, tock\rangle\end{array} = \begin{array}{l}\{\langle a, b, \{a, b, c, \checkmark\}, tock\rangle, \\ \quad \langle b, a, \{a, b, c, \checkmark\}, tock\rangle\}\end{array} \tag{98}$$

$$\begin{array}{l}\langle a, \{a, c, \checkmark\}, tock\rangle \\ \quad [\![\{c\}]\!]^T \\ \langle\{a, c, \checkmark\}, tock, b\rangle\end{array} = \{\langle a, \{a, c, \checkmark\}, tock, b\rangle\} \tag{99}$$

When two $tock$ events are merged, the refusals before them are merged in a similar way to single refusals (Example 14). In particular, refusals that do not contain the same non-synchronised events yield no output traces, as shown in Eq. (95). When the refusals before the $tock$ events match, the result is the traces with $tock$ events preceded by a refusal that is the union of the refusals on each side. Any observations after a $tock$ event are then merged, as can be seen in Eq. (96), where a b event follows the $tock$ event.

Since $tock$ requires synchronisation, any events that do not require synchronisation occurring before $tock$ in a trace to be merged are included before it in the resulting traces. Thus, in Eq. (97) a b occurs before the $tock$. When there are non-synchronised events at the start of both traces to be merged, as in Eq. (98), the non-synchronised events can occur in either order before $tock$. 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 Eq. (99), where a occurs before $tock$ and b occurs after.

Overall, Examples 13–16 characterise the traces of the parallel composition $a \longrightarrow \mathbf{Stop} \; [\![\{c\}]\!]^T \; b \longrightarrow c \longrightarrow \mathbf{Stop}$, which are generated by merging each of the traces of $a \longrightarrow \mathbf{Stop}$ with each of the traces of $b \longrightarrow c \longrightarrow \mathbf{Stop}$.

5.12 Hiding

The semantics of hiding, $P \setminus X$, is 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 defined by $hideTrace$ to ensure subset closure.

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

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.

$$\begin{aligned}
&\forall X, Y : \mathbb{P}\,\Sigma^{\checkmark}_{tock}; e : \Sigma^{\checkmark}_{tock}; s : TickTockTrace \bullet \\
&\quad hideTrace\ X\ \langle\rangle = \{\langle\rangle\} \wedge && (100)\\
&\quad hideTrace\ X\ \langle ref\ Y\rangle = \{Z : \mathbb{P}\,Y | X \subseteq Y \bullet \langle ref\ Z\rangle\} \wedge && (101)\\
&\quad (e \in X \implies hideTrace\ X\ (\langle evt\ e\rangle \frown s) = hideTrace\ X\ s) \wedge && (102)\\
&\quad (e \notin X \implies hideTrace\ X\ (\langle evt\ e\rangle \frown s) = \\
&\quad\quad \{t : hideTrace\ X\ s \bullet \langle evt\ e\rangle \frown t\}) \wedge && (103)\\
&\quad (tock \in X \implies hideTrace\ X\ (\langle ref\ Y, evt\ tock\rangle \frown s) = \\
&\quad\quad hideTrace\ X\ s) && (104)\\
&\quad (tock \notin X \implies hideTrace\ X\ (\langle ref\ Y, evt\ tock\rangle \frown s) = \\
&\quad\quad \{Z : \mathbb{P}\,Y; t : hideTrace\ X\ s | X \subseteq Y \bullet \langle ref\ Z, evt\ tock\rangle \frown t\}) && (105)
\end{aligned}$$

The first equation, (100), 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 in the empty trace, so the trace is simply returned as-is.

The other base case of the definition is specified by Eq. (101), 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 (re)include such refusals where there is a refusal including X in order to maintain subset closure.

Equations (102) and (103) define $hideTrace$ when applied to a trace starting with a non-$tock$ event e. Equation (102) specifies the case where e is in X; the result is that of applying $hideTrace$ to the remaining trace; e is hidden and removed. If e is not in X, as specified by Eq. (103), the result is that of prepending e to the traces resulting from applying $hideTrace$ to the rest of the trace.

Finally, Eqs. (104) and (105) define the result when $hideTrace$ is applied to a trace starting with a refusal Y followed by a $tock$. This case can be viewed as a combination of the cases for a non-$tock$ event and for a single refusal. As for a non-$tock$ event, there are two cases depending on whether $tock$ is in X or not. If $tock$ is in X (Eq. 104), then it is hidden and the result is that of applying $hideTrace$ to the rest of the trace, as in Eq. (102). If $tock$ is not in X (Eq. 105), then it is prepended to the result of applying $hideTrace$ to the rest of the trace, as in Eq. (103), but the refusal Y is handled as in Eq. (101). 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 17 Considering the traces of $b \rightarrow c \rightarrow$ **Stop**, with the hiding set $\{a, b\}$, the results of applying $hideTrace$ are those shown below.

$$hideTrace\ \{a, b\}\ \langle\{a, c, \checkmark\}, tock, b\rangle = \{\} \tag{106}$$

$$hideTrace\ \{a, b\}\ \langle b, c\rangle = \{\langle c\rangle\} \tag{107}$$

$$hideTrace\ \{a, b\}\ \langle b, \{a, b, \checkmark\}, tock, c\rangle = \{\langle\{a, b, \checkmark\}, tock, c\rangle,\ \ldots\} \tag{108}$$

Applying $hideTrace$ to a trace containing a refusal that does not include the hiding set (including refusals before $tock$ events when $tock$ is not hidden) results in the empty set of traces (Eq. 106). This is because there is no stability if a hidden event can be performed. This applies even if some of the hidden events are refused, since there are still hidden events that could occur.

Where b occurs in a trace, it is removed, since b is in the hiding set, as can be seen in Eqs. (107) and (108). The $hideTrace$ function is then applied to the rest of the trace after b. The event c is included on its own in the sole resulting trace in (107), since c is not hidden. For refusals that include all events in the hiding set, including those before non-hidden $tock$ events, the refusal is included with all its subsets, as can be seen in Eq. (108).

5.13 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.

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

The definition of $renameTrace$ is below. The result of applying $renameTrace$ to the empty set is the set containing the empty trace (Eq. 109). When $renameTrace$ is applied to a trace beginning with event e (Eq. 110), the result is the set of traces resulting from applying $renameTrace$ to the rest of the trace, with the event formed from applying f to e prepended.

$$\begin{aligned}
&\forall f : \ \Sigma^{\checkmark}_{tock} \rightarrow \Sigma^{\checkmark}_{tock};\, e : \Sigma^{\checkmark}_{tock};\, X : \mathbb{P}\, \Sigma^{\checkmark}_{tock};\, s : \mathrm{seq}\ Obs \bullet \\
&\quad renameTrace\ f\ \langle\rangle = \{\langle\rangle\} \wedge && (109)\\
&\quad renameTrace\ f\ (\langle evt\ e\rangle \frown s) = \\
&\qquad \{t : renameTrace\ f\ s \bullet \langle evt\ (f\ e)\rangle \frown t\} \wedge && (110)\\
&\quad renameTrace\ f\ (\langle ref\ X\rangle \frown s) = \\
&\qquad \{t : renameTrace\ f\ s;\, Y : \mathbb{P}\, \Sigma^{\checkmark}_{tock} | X = (f^{\sim})\ (\!|\ Y\ |\!) \bullet \langle ref\ Y\rangle \frown t\} && (111)
\end{aligned}$$

For a trace starting with a refusal X (Eq. 111), the result is that of prepending a corresponding refusal Y to the traces obtained by applying $renameTrace$ to 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, they must be refused (by healthiness condition **TT2**).

Example 18 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.

$$renameTrace\ f\ \langle \{b, c, \checkmark\}, tock, a\rangle = \{\} \tag{112}$$

$$renameTrace\ f\ \langle \{c, \checkmark\}, tock, a\rangle = \{\langle \{a, c, \checkmark\}, tock, b\rangle, \langle \{c, \checkmark\}, tock, b\rangle\} \tag{113}$$

$$renameTrace\ f\ \langle a, \{a, b, c, \checkmark\}, tock\rangle = \{\langle b, \{a, b, c, \checkmark\}, tock\rangle, \langle b, \{b, c, \checkmark\}, tock\rangle\} \tag{114}$$

For a refusal, there must be refusals that map to it under the inverse image of f. The maximal initial refusal $\{b, c, \checkmark\}$ thus yields no traces, as shown in Eq. (112), since any refusal that includes b has an inverse image under f including both a and b. Since $\{b, c, \checkmark\}$ does not refuse a, we cannot, therefore, include b, but without refusing b we cannot match the refusal of b. So there are no corresponding refusals for $\{b, c, \checkmark\}$, and, therefore, no traces.

The subset refusal $\{c, \checkmark\}$, however, 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 starting with $\{c, \checkmark\}$ is thus the set of traces starting with refusals $\{a, c, \checkmark\}$ and $\{c, \checkmark\}$ (Eq. 113). We show all the refusals to emphasise that only those are included, rather than the full subset closure. **TT1** healthiness of the process to which renaming is applied ensures that the renaming fulfils **TT1**. Proof of healthiness of renaming and all other operator definitions in this section is discussed in Sect. 5.15. After the initial refusal, $renameTrace$ is applied to the rest of the trace, leaving $tock$ unaffected and replacing a with b.

Refusals occurring after renamed events are handled in the same way as in Eq. (112) and (113), as is illustrated in Eq. (114). We note that both a and b occur in the refusal in Eq. (114), so b can be included in the refusals in the resulting traces, in contrast to the situation illustrated by Eq. (112) where the lack of a refusal of a eliminates traces.

5.14 Recursion

If P is a recursive process defined by $P = F(P)$, then the semantics of P is defined, as shown below, by the union of all iterations of F applied to a divergence. Here, we take F as a function from processes to processes, and use F^n to refer to repeated application of F n times: $F^0(X) = X$, $F^1(X) = F(X)$, $F^{n+1}(X) = F(F^n(X))$.

$$tt[\![P]\!] = \bigcup\{n : \mathbb{N} \bullet F^n(tt[\![\mathbf{div}]\!])\}$$

The application of the iteration to a divergence means an unguarded recursion results in a divergence. Since divergence refines every process in $\checkmark$-$tock$, this means recursion is defined by the greatest fixed point of F. All the operators we have defined distribute through arbitrary unions, so for F made up of those operators, the defining equation $P = F(P)$ holds with this semantics for recursion.

Mutual recursion is, as usual, the fixed point for a vector of functions.

5.15 Key results

For validation of our definitions, we prove that the set of traces of all processes defined using our operators satisfy the healthiness conditions described in Sect. 4. We, therefore, say that $\checkmark$-$tock$ processes are healthy.

The healthiness conditions are not directly enforced, but rather arise from the definitions, which have been constructed to ensure that these properties hold. To establish the healthiness of the definitions, first of all, we prove that all traces in all sets characterised by our definitions are well formed. This is proved by first showing that each of the traces of the basic processes (**div**, **Skip**, **Stop**, $\mathbf{Stop}_U$, **Wait** n) is well formed. We then consider each of the remaining operators: we assume the sets of $\checkmark$-*tock* traces of its operands are well formed and show that the each of traces of the process formed by applying the operator to those operands is well formed. We thus have that the traces of all processes are well formed by induction. For example, we show the following for external choice.

$$\begin{aligned} & tt[\![P]\!] \subseteq TickTockTrace \wedge tt[\![Q]\!] \subseteq TickTockTrace \\ & \quad \implies tt[\![P \mathbin{\Box} Q]\!] \subseteq TickTockTrace \end{aligned}$$

This follows from the fact that $tt[\![P \mathbin{\Box} Q]\!]$ is made up of traces in $tt[\![P]\!]$ or $tt[\![Q]\!]$ satisfying certain constraints, so that $tt[\![P \mathbin{\Box} Q]\!] \subseteq tt[\![P]\!] \cup tt[\![Q]\!]$. Since, by assumption, the traces in $tt[\![P]\!]$ and $tt[\![Q]\!]$ are well formed, so are those in $tt[\![P \mathbin{\Box} Q]\!]$.

To establish healthiness of the sets of traces, we also consider first each of the basic processes P, and prove that $tt[\![P]\!]$ satisfies all the conditions. In addition, we prove that all operators, when applied to healthy processes, characterise healthy processes. For example, we prove the following theorem for external choice.

$$\begin{aligned} & \mathbf{TT0}(P) \wedge \mathbf{TT1}(P) \wedge \mathbf{TT2}(P) \wedge \mathbf{TT3}(P) \\ & \quad \wedge \mathbf{TT0}(Q) \wedge \mathbf{TT1}(Q) \wedge \mathbf{TT2}(Q) \wedge \mathbf{TT3}(Q) \\ & \quad\quad \implies \mathbf{TT0}(P \mathbin{\Box} Q) \wedge \mathbf{TT1}(P \mathbin{\Box} Q) \wedge \mathbf{TT2}(P \mathbin{\Box} Q) \wedge \mathbf{TT3}(P \mathbin{\Box} Q) \end{aligned}$$

Additionally, since CSP is a language for refinement, it is important that the operators are monotonic: refinement of a process to which an operator is applied produces a refinement of the process as a whole. For example, the statement of monotonicity for the left operand of external choice is as follows.

$$P \sqsubseteq Q \implies P \mathbin{\Box} R \sqsubseteq Q \mathbin{\Box} R$$

This property is important because it allows processes to be considered and reasoned about in a compositional way, independently of their context. We have thus proved that monotonicity holds for all the operators of $\checkmark$-*tock* CSP.

A distinguishing feature of CSP over other process algebras is the distributivity of the operators, except recursion, over internal choice. This is stronger than monotonicity, and holds for $\checkmark$-*tock* operators. For example, we have the following property for external choice, which implies the monotonicity property above.

$$(P \sqcap Q) \mathbin{\Box} R = (P \mathbin{\Box} R) \sqcap (Q \mathbin{\Box} R)$$

We have shown that, as expected, distributivity over internal choice holds for all $\checkmark$-*tock* operators except recursion. This is because, for example, recursions such as $P = F(P \sqcap Q)$ and $Q = G(P \sqcap Q)$ can resolve the choices differently at each iteration, so neither P nor Q can be expressed as the choice of two recursions.

Our semantics also satisfies the properties discussed in Sect. 2 that ensure that processes behave as captured by their failure semantics within a time unit. We recall that Property 1 requires that any process whose traces do not include *tock* has the same failures and $\checkmark$-*tock* semantics. As mentioned, comparing $\checkmark$-*tock* semantics to failures semantics requires a common representation for failures and $\checkmark$-*tock* traces, so that they can be compared. To prove Property 1 we thus define a function $tt2F$ for converting a set of $\checkmark$-$tock$ traces, to a set of failures.

$$\begin{array}{l} tt2F(P) = \{t : \mathrm{seq}\,\Sigma^{\checkmark};\, X : \mathbb{P}\,\Sigma^{\checkmark} \mid \\ \quad ((evt \circ t) \frown \langle ref\ X\rangle \in P) \\ \quad \lor ((evt \circ t) \frown \langle evt\ \checkmark\rangle \in P \land \checkmark \notin X) \\ \quad \lor ((evt \circ t) \in P \land (\exists s \bullet t = s \frown \langle \checkmark \rangle)) \\ \} \end{array}$$

This function takes a set P of $\checkmark$-$tock$ traces, and defines a corresponding set of failures containing traces of non-$tock$ events t paired with a refusal X. The pairs (t, X) are those that correspond to one of three kinds of $\checkmark$-$tock$ traces in P: (1) a trace consisting of the events of t followed by X; (2) a trace consisting of the events of t followed by $\checkmark$, where X does not contain $\checkmark$; or (3) a trace consisting of the events of t, where t ends in $\checkmark$ and X is arbitrary.

Sequences are functions from indices to the elements of the sequence, so the composition $evt \circ t$ is the sequence containing the result of applying the constructor evt to the events of t. We disregard $tock$ events in $tt2F$, since Property 1 considers processes with traces that do not include $tock$, and we are mapping to a failures semantics for a set of events that does not include $tock$.

We then show Property 1 by showing that the $\checkmark$-$tock$ semantics of each operator, mapped under $tt2F$, is equal to the failures semantics of that operator. For example, we have shown the following for external choice.

$$tt2F(tt[\![P]\!]) = [\![P]\!]_{\mathcal{F}} \land tt2F(tt[\![P]\!]) = [\![P]\!]_{\mathcal{F}} \implies tt2F(tt[\![P \,\Box\, Q]\!]) = [\![P \,\Box\, Q]\!]_{\mathcal{F}}$$

We then have that $tt2F(tt[\![P]\!]) = [\![P]\!]_{\mathcal{F}}$ for any process P by induction, and we have proved that Property 1 follows from this.

For Property 2, it is easy to see that there are simple processes such as **Skip** and $\mathbf{Stop}_U$ that do not include $tock$ events. A more complex example of such a process is given above in Example 11, where adding a time-synchronising interrupt with $\mathbf{Stop}_U$ removes $tock$ events from a process.

It is outside the scope of this work to prove a complete set of algebraic laws for *tock*-CSP, but we have proved some laws to validate the more complex operator definitions. For example, we have shown that parallel composition is associative and that **Stop** is a unit of external choice. In total, we have proved 16 laws.

All the results discussed in this paper have been established using Isabelle/HOL. The examples in Table 1 have been checked using model checking (see Sect. 7). Our support for $\checkmark$-$tock$ proofs using Isabelle is described next.

6 Mechanisation in Isabelle

In this section we describe a mechanisation of the $\checkmark$-$tock$ model in Isabelle/HOL, via recursive data types, and of its healthiness conditions and operators. In Sect. 6.1 we describe the encoding of $\checkmark$-$tock$ traces, and of the healthiness conditions. In Sect. 6.2, we discuss the definitions of the semantics of sequential composition, as an example, and the proof of healthiness of the sets of traces that it defines. As said, all theories and proofs can be found in [5].

6.1 Model

Our mechanisation of $\checkmark$-*tock* in Isabelle is very close to our presentation of this model in the previous section. This provides confidence in our encoding and valuable validation of our definitions. It is constructed by defining a parametric **datatype** for $\Sigma^{\checkmark}_{tock}$ as `'e ttevent` below, where `'e` is an arbitrary HOL type.

```
datatype 'e ttevent = Event 'e  | Tock | Tick
```

The `Event 'e` constructor is used to represent an event in Σ, while `Tock` and `Tick` correspond to *tock* and $\checkmark$, respectively. The advantage of parametric type definitions is that results are independent of a particular Σ.

The HOL type for Obs is defined using another **datatype** `'e ttobs` that has two constructors: `ObsEvent` and `Ref`, corresponding to $evt\langle\langle\_\rangle\rangle$ and $ref\langle\langle\_\rangle\rangle$, respectively. The more concise syntax `[_]`$_E$, and `[_]`$_R$ can also be used.

```
datatype 'e ttobs = ObsEvent "'e ttevent" ("[_]E") |
                        Ref "'e ttevent set" ("[_]R")
```

The type seq Obs is specified using `'e ttobs list`, the type of finite lists parametrised by `'e ttobs`, which we abbreviate in Isabelle by defining a **type_synonym**.

```
type_synonym 'e tttrace = "'e ttobs list"
```

A valid trace ($TickTockTrace$) is identified by requiring that the Boolean function `ttWF`, defined below, gives $true$ when applied to it.

```
fun ttWF :: "'e tttrace ⇒ bool" where
  "ttWF [] = True" |
  "ttWF [[X]R] = True" |
  "ttWF [[Tick]E] = True" |
  "ttWF ([Event e]E # σ) = ttWF σ" |
  "ttWF ([X]R # [Tock]E # σ) = (ttWF σ ∧ Tock ∉ X)" |
  "ttWF σ = False"
```

There are three base cases: the empty list `[]`; the list whose only element is a refusal, or a `Tick` event. The recursive cases are those where an event is followed by a list σ, and where `Tock` is preceded by a refusal followed by a list σ. Here, `#` is the constructor for lists. Since in HOL functions are total, there is a default case defined last that matches any construction not covered by the previous cases.

The healthiness conditions, and operators, are defined by HOL functions on sets of `'e ttrace`, that is, models of $\checkmark$-*tock* processes. We introduce a corresponding **type_synonym** that identifies processes with such sets.

```
type_synonym 'e ttprocess = "'e tttrace set"
```

The healthiness conditions are defined exactly as presented earlier, taking into account the minor differences in the mathematical syntax adopted by Isabelle. For example, the empty set is typed as `{}` instead of $\emptyset$.

```
definition TT0 :: "'e ttprocess ⇒ bool" where
  "TT0 P = (P ≠ {})"
definition TT1 :: "'e ttprocess ⇒ bool" where
  "TT1 P = (∀ ρ σ. (ρ ≲C σ ∧ σ ∈ P) ⟶ ρ ∈ P)"
```

Furthermore, in the case of **TT2**, for example, the set comprehension does not need to be explicitly typed as Isabelle can automatically infer the correct types. The definition is, otherwise, identical to **TT2**.

```
definition TT2 :: "'e ttprocess ⇒ bool" where
  "TT2 P = (∀ ϱ σ X Y. (ϱ @ [[X]R] @ σ ∈ P ∧ (Y ∩ {e. (e ≠ Tock ∧ ϱ
@ [[e]E] ∈ P) ∨ (e = Tock ∧ ϱ @ [[X]R, [e]E] ∈ P) } = {}))
     ⟶ ϱ @ [[X ∪ Y]R] @ σ ∈ P)"
```

In the case of **TT3**, we have encoded it in a similar form to that presented earlier.

```
definition TT3 :: "'e ttprocess ⇒ bool" where
  "TT3 P = (∀ ϱ σ X. ϱ @ [[X]R] @ σ ∈ P ⟶ ϱ @ [[X ∪ {Tick}]R] @ σ
∈ P)"
```

To make proofs by induction easier, we have also provided an alternative definition `TT3w`, shown below, that uses a recursively defined function. We use `TT3w` in proofs; we have proved lemmas connecting it to the definition above.

```
definition TT3w :: "'e ttprocess ⇒ bool" where
  "TT3w P = (∀ ϱ. ϱ ∈ P ⟶ add_Tick_refusal_trace ϱ ∈ P)"
```

It takes a ✓-*tock* process `P` and requires that, for every trace `ϱ` in `P`, the trace generated by the function `add_Tick_refusal_trace`, defined next, is also in `P`.

```
fun add_Tick_refusal_trace :: "'e tttrace ⇒ 'e tttrace" where
  "add_Tick_refusal_trace [] = []" |
  "add_Tick_refusal_trace ([e]E # t) = [e]E # add_Tick_refusal_trace
t" |
  "add_Tick_refusal_trace ([X]R # t) = [X ∪ {Tick}]R #
add_Tick_refusal_trace t"
```

For the empty trace `[]`, `add_Tick_refusal_trace` returns the empty trace. When a trace begins with an event `[e]`$_E$, the result is the trace with `[e]`$_E$ followed by the result of applying `add_Tick_refusal_trace` to the rest of the trace. Finally, when a trace begins with a refusal `[X]`$_R$, the result is the trace that begins with a refusal of `X` with `Tick` added to it (`[X ∪ {Tick}]`$_R$), followed by the result of applying `add_Tick_refusal_trace` to the rest of the trace. The function `add_Tick_refusal_trace` thus has the effect of adding `Tick` to every refusal in a trace.

The difference between `TT3w` and `TT3`, is that `TT3` considers part of a trace, where a refusal is preceded by ρ and followed by σ (the trace $\rho \frown \langle ref\ X\rangle \frown \sigma$ in the antecedent), whereas `add_Tick_refusal_trace` is defined recursively, applying to every refusal in a trace. Taken together with the subset closure in **TT1**, the definitions are equivalent, a fact we have proved in our mechanisation.

6.2 Operators

To illustrate the use of Isabelle in establishing key results of ✓-*tock*, we show the proof of well-formedness for the traces of sequential composition. We show, first of all, the definition for sequential composition below.

```
definition SeqCompTT :: "'e ttprocess ⇒ 'e ttprocess ⇒ 'e ttprocess"
(infixl ";C" 60) where
  "P ;C Q = {ϱ∈P. ∄ s. ϱ = s @ [[Tick]E]}
             ∪ {ϱ. ∃ s t. s @ [[Tick]E] ∈ P ∧ t ∈ Q ∧ ϱ = s @ t}"
```

Besides its type signature, it follows very closely the definition given in Sect. 5. The proof of closure is shown next, using the Isar dialect of Isabelle that can be used to write proofs in a deductive style, with major steps justified by the application of a relatively small number of proof tactics.

```
lemma SeqComp_wf:
  assumes "∀t∈P. ttWF t" "∀t∈Q. ttWF t"
  shows "∀ t ∈ P ;C Q. ttWF t"
  unfolding SeqCompTT_def
proof auto
  fix t
  assume "t ∈ P" "∀s. t ≠ s @ [[Tick]E]"
  then show "ttWF t"
    using assms(1) by auto
next
  fix s ta
  assume "s @ [[Tick]E] ∈ P"
  then have 1: "ttWF (s @ [[Tick]E])"
    using assms(1) by auto
  assume "ta ∈ Q"
  then have 2: "ttWF ta"
    using assms(2) by auto
  from 1 2 show "ttWF (s @ ta)"
    by (induct s rule:ttWF.induct, auto)
qed
```

The **lemma** `SeqComp_wf` **assumes** that all the traces of both `P` and `Q` are well-formed, and **shows** that all traces of the sequential composition are well-formed (`ttWF`). The proof starts by **unfolding** the definition of $;_C$ (`SeqCompTT_def`). The following application of the tactic `auto` in a **proof** environment leads to two proof goals.

The first goal requires showing that every trace `t` in `P` not ending in `[Tick]`$_E$ is well-formed according to `ttWF`, corresponding to the case captured by the first set comprehension in the definition of `SeqCompTT`. It can be discharged using the assumption that every trace of `P` satisfies `ttWF` (`assms(1)`).

The second goal requires showing that assuming `s @ [[Tick]`$_E$`]` is in `P` and `ta` is in `Q`, then the concatenation (`@`) of `s` and `ta` satisfies `ttWF`, corresponding to the case captured by the second set comprehension in the definition of `SeqCompTT`. It can be discharged by showing that: 1) `s @ [[Tick]`$_E$`]` satisfies `ttWF`, using the assumption that every trace of `P` satisfies `ttWF` (`assms(1)`); and 2) `ta` satisfies `ttWF`, discharged using the assumption that every trace of `Q` satisfies `ttWF` (`assms(2)`); and finally establishes that the concatenation of the traces satisfies `ttWF` by `induct`ing over the possible traces of `s` based on the definition of `ttWF` (via the rule `ttWF.induct`).

Proofs for other key results of ✓-*tock*, namely that all operators are monotonic and closed under the healthiness conditions, have also largely been done using the Isar dialect. Despite the high degree of automation provided by Isabelle, it is reassuring that the proof steps in Isar are relatively easy to follow.

7 Mechanisation in FDR

To encode the refinement for ✓-*tock* we tailor Mestel and Roscoe [23,24]'s model shifting technique to encoding refusals using traces, so that ✓-*tock* refinement is reduced to traces refinement. We begin this section by explaining how refusals are encoded, followed by the encoding of ✓-*tock* traces, and termination. Finally we revisit Example 1.

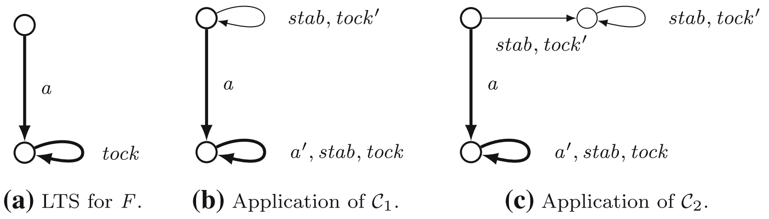

Fig. 2 LTS calculated from F and stepwise application of contexts $\mathcal{C}_1$ and $\mathcal{C}_2$

7.1 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, 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 6 $\mathcal{C}_1[P] \mathrel{\widehat=} \mathbf{Pri}_{\leq_1}(P|||RUN(\Sigma'_{tock} \cup \{stab\}))$

Process P is composed in interleaving ($|||$), a form of parallel composition where 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 $\mathbf{Pri}_{\leq}(P)$ [35], 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 idea of priorities on actions originates with [9]). 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 $a \rightarrow P \,\Box\, b \rightarrow Q$ with $a < b$ would yield $b \rightarrow \mathbf{Pri}_{\leq}(Q)$. 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.

Example 19 With $\Sigma = \{a\}$, we have for $F \mathrel{\widehat=} (a \rightarrow \mathbf{Stop}) \,\Box\, \mathbf{Stop}_U$ that:

$$tt[\![F]\!] = \{\langle\rangle, \langle\{tock, \checkmark\}\rangle, \langle a\rangle, \langle a, ref\ \Sigma^{\checkmark}, tock\rangle, \ldots\}$$

$$traces(\mathcal{C}_1[F]) = \{\langle\rangle, \langle stab, tock'\rangle, \langle a\rangle, \langle a, a', stab, tock\rangle, \langle stab, tock', a\rangle, \ldots\}$$

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

For reasoning based on $\checkmark$-$tock$ semantics, however, the trace $\langle stab, tock', a\rangle$, encoding $\langle\{tock\}, a\rangle$ is undesirable because, in a $\checkmark$-$tock$ trace, after a refusal the only possible event is $tock$. Next we introduce a context $\mathcal{C}_2$ to eliminate such undesirable traces and support checks for refinement based on a $\checkmark$-$tock$ semantics.

7.2 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 another context $\mathcal{C}_2[P]$, where $\mathcal{C}_1[P]$ is composed in parallel with a process Sem synchronising on events in 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 $\checkmark$-$tock$.

Definition 7 $\mathcal{C}_2[P] \mathrel{\widehat=} \mathcal{C}_1[P] \llbracket\, \Sigma_{tock} \cup \Sigma'_{tock} \cup \{stab\} \,\rrbracket\, Sem$

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 8 $Sem \mathrel{\widehat=} (\Box\, e : \Sigma_{tock} \bullet e \rightarrow Sem) \,\Box\, (\Box\, r : \Sigma'_{tock,stab} \bullet r \rightarrow Ref)$
$Ref \mathrel{\widehat=} (\Box\, r : \Sigma'_{tock,stab} \bullet r \rightarrow Ref) \,\Box\, tock \rightarrow Sem$

Sem offers every event e from Σ_{tock} in an external choice followed by a recursion, and every event r from $\Sigma'_{tock,stab}$ encoding refusals also in choice, but followed by Ref. That process also offers events from $\Sigma'_{tock,stab}$ followed by a recursion, but $tock$ is also offered followed by 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 19 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 20 $traces(\mathcal{C}_2[F]) =$
$\{\langle\rangle, \langle stab, tock'\rangle, \langle a\rangle, \langle a, a', stab, tock\rangle, \langle stab, tock', stab, \ldots\rangle, \ldots\}$

The $\checkmark$-$tock$ traces of F before observing event a are encoded by $\langle\rangle$, $\langle stab, \ldots\rangle$, $\langle tock', \ldots\rangle$, 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. Subset inclusion of refusal sets corresponds to subset inclusion over the set of encoding traces, which is key to reducing refinement of $\checkmark$-$tock$ traces to traces refinement.

7.3 Termination

The original technique in [23] did not account for termination. For example, we have that $\mathcal{C}_1[\mathbf{Skip}] = \mathcal{C}_1[\mathbf{Stop}_U]$. 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 for $\mathbf{Stop}_U$. To cater for termination, we define a third context $\mathcal{C}_3$ and extend Σ_{tock} with a fresh event $tick$ that encodes $\checkmark$, similarly to the approach in [24]. Unlike [24, p. 411], however, we do not hide $tick$, as this allows **Skip** to be erroneously refined by $\mathbf{Stop}_U$.

Definition 9 $\mathcal{C}_3[P] \mathrel{\widehat{=}} \mathcal{C}_2[P \mathbin{;} tick \rightarrow \mathbf{Skip}]$

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: P is refined by Q in the $\checkmark$-*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.

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

Proof *Similarly to that outlined in [23, Lemma 3.1] by following the above construction, where the regulator process is Sem. In addition to the refusals of the stable-failures model as described in [23], encoded by dashed events x' and $stab$, the event $tock$ can happen, with the subsequent observable events in a trace being those of Sem or P, as permitted by the synchronisation of the context with P.* □

A script with the complete encoding is available.[2]

To illustrate the refinement technique we reconsider Example 1. Recall that although R is refined by S when considering the failures semantics of CSP, in a timed setting this refinement should not hold. Here we focus on the result of the check $R \sqsubseteq S$. Using Theorem 1, this amounts to checking whether $\mathcal{C}_3[R] \sqsubseteq_{\mathcal{T}} \mathcal{C}_3[S]$ holds. This yields a counter-example where after the trace $\langle b', tock \rangle$ process S can perform event a but R cannot. That is, having refused b, followed by a $tock$, process R then behaves as **Stop**, whereas S can perform a. This is exactly the scenario we previously described in Sect. 2.

On the other hand, when R and S are considered in context $\mathcal{I}$, the refinement $\mathcal{I}[R] \sqsubseteq \mathcal{I}[S]$ should hold in $\checkmark$-*tock* as discussed in Sect. 2 and summarized in Table 1. Using our encoding in FDR we have checked that $\mathcal{C}_3[\mathcal{I}[R]] \sqsubseteq_{\mathcal{T}} \mathcal{C}_3[\mathcal{I}[S]]$ holds.

8 Conclusions

The inclusion of $tock$ in CSP enables a rich and flexible approach to modelling time. However, despite several models accounting for the use of $tock$, none have, so far, adequately catered for deadlines, termination, 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.

Most case studies in the *tock*-CSP literature using refinement focus on safety only [11,19]. Evans and Schneider [11] consider an embedding of *tock*-CSP in PVS [33] using the traces model for analysis of time-dependent security properties. An embedding in Isabelle/HOL of the failures model of CSP has also been considered by Isobe and Roggenbach [18]. However, to reason about liveness 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 [37]. Despite using timestops, Roscoe later describes these undesirably as “breaching the laws of nature by preventing time from progressing” [38]. Similarly, Ouaknine’s discrete-time refusal testing model [31] does not admit timestops. Like Timed CSP, there is no explicit control of time, thus time can pass arbitrarily between events, and Zeno behaviour is forbidden.

[2] https://github.com/robo-star/tick-tock-CSP/tree/master/fdr/tick-tock.csp.

Timestops are useful to model deadlines as shown here and by other authors [32]. More recently, Lowe and Ouaknine [22] revisited the discrete-time refusal testing model by considering traces where refusals are recorded only before $tock$, but do not admit timestops or termination. They have also proposed a similar model, called timed testing, that does not record instability.

Armstrong et al. [3] explore refinement checking 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 [39] using the concept of slow-abstraction, and more recently Mestel [24] employed model-shifting, an approach that is similar to our encoding of $\checkmark$-$tock$ in FDR (described in Sect. 7), but whose treatment of termination is not adequate.

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.

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 [7]. It is clear that $\mathbf{Pri}_{\leq}$ endows CSP with extra expressive power [35], allowing, for example, regular events to be made urgent by prioritising them over $tock$. In future work, we also plan to provide a prioritisation operator for $\checkmark$-$tock$.

References

1. Aceto, L., Murphy, D.: On the ill-timed but well-caused. In: Best, E. (ed.) CONCUR'93, pp. 97–111. Springer, Berlin (1993)
2. Andersen, H.R., Mendler, M.: An asynchronous process algebra with multiple clocks. In: Sannella, D. (ed.) Programming Languages and Systems—ESOP'94, pp. 58–73. Springer, Berlin (1994)
3. Armstrong, P., Lowe, G., Ouaknine, J., Roscoe, A.: Model checking Timed CSP. In: Proceedings of HOWARD (Festschrift for Howard Barringer) (2012)
4. Baeten, J.C.M., Bergstra, J.A.: Discrete time process algebra. Formal Aspects Comput. **8**(2), 188–208 (1996)
5. Baxter, J., Ribeiro, P.: Tick-tock-CSP in Isabelle/HOL (2019). https://github.com/robo-star/tick-tock-CSP/
6. Berry, G., Cosserat, L.: The ESTEREL synchronous programming language and its mathematical semantics. In: International Conference on Concurrency, pp. 389–448. Springer, Berlin (1984)
7. Cavalcanti, A., Sampaio, A., Miyazawa, A., Ribeiro, P., Conserva Filho, M., Didier, A., Li, W., Timmis, J.: Verified simulation for robotics. Sci. Comput. Program. (2019)
8. Chen, L.: An interleaving model for real-time systems. In: Nerode, A., Taitslin, M. (eds.) Logical Foundations of Computer Science—Tver'92, pp. 81–86. Springer, Berlin (1992)
9. Cleaveland, R., Hennessy, M.: Priorities in process algebras. Inf. Comput. **87**(1), 58–77 (1990). https://doi.org/10.1016/0890-5401(90)90059-Q. Special Issue: Selections from 1988 IEEE Symposium on Logic in Computer Science

10. Davies, J.: Specification and Proof in Real Time CSP, vol. 6. Cambridge University Press, Cambridge (1993)
11. Evans, N., Schneider, S.: Analysing time dependent security properties in CSP using PVS. In: European Symposium on Research in Computer Security, pp. 222–237. Springer, Berlin (2000)
12. Gerth, R., Boucher, A.: A timed failures model for extended communicating processes. In: Ottmann, T. (ed.) Automata, Languages and Programming, pp. 95–114. Springer, Berlin (1987)
13. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3—A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS, Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)
14. 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, pp. 3–12. Springer, Cham (2015)
15. Groote, J.F.: Specification and verification of real time systems in ACP. In: Proceedings of the IFIP WG6.1 10th International Symposium on Protocol Specification, Testing and Verification X, pp. 261–274 (1990)
16. Hennessy, M., Regan, T.: A process algebra for timed systems. Inf. Comput. **117**(2), 221–239 (1995)
17. 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, pp. 54–68. Springer, London (2012)
18. 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, pp. 108–123. Springer, Berlin (2005)
19. 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, pp. 335–351. Springer, Berlin (2011)
20. Klusener, A.S.: Abstraction in real time process algebra. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) Real-Time: Theory in Practice, pp. 325–352. Springer, Berlin (1992)
21. Leuschel, M., Butler, M.: ProB: a model checker for B. In: International Symposium of Formal Methods Europe, pp. 855–874. Springer, Berlin (2003)
22. Lowe, G., Ouaknine, J.: On timed models and full abstraction. Electron. Notes Theor. Comput. Sci. **155**, 497–519 (2006)
23. Mestel, D., Roscoe, A.: Reducing complex CSP models to traces via priority. Electron. Notes Theor. Comput. Sci. **325**, 237–252 (2016)
24. Mestel, D., Roscoe, A.W.: Translating between models of concurrency. Acta Inf. **57**(3), 403–438 (2020)
25. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1982)
26. Milner, R.: Calculi for synchrony and asynchrony. Theoret. Comput. Sci. **25**(3), 267–310 (1983)
27. Moller, F., Tofts, C.: A temporal calculus of communicating systems. In: International Conference on Concurrency Theory, pp. 401–415. Springer, Berlin (1990)
28. Mukarram, A.: A refusal testing model for CSP. Ph.D. Thesis, University of Oxford (1993)
29. Nicollin, X., Sifakis, J.: The algebra of timed processes, ATP: theory and application. Inf. Comput. **114**(1), 131–178 (1994)
30. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Berlin (2002)
31. Ouaknine, J.: Discrete analysis of continuous behaviour in real-time concurrent systems. Ph.D. Thesis, University of Oxford (2000)
32. Ouaknine, J., Worrell, J.: Timed CSP = closed timed automata. Electron. Notes Theor. Comput. Sci. **68**(2), 142–159 (2002)
33. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: International Conference on Automated Deduction, pp. 748–752. Springer, Berlin (1992)
34. Phillips, I.: Refusal testing. Theoret. Comput. Sci. **50**(3), 241–284 (1987)
35. Roscoe, A.: The expressiveness of CSP with priority. Electron. Notes Theor. Comput. Sci. **319**, 387–401 (2015)
36. Roscoe, A., Reed, G.: A timed model for communicating sequential processes. Theoret. Comput. Sci. **58**, (1988)
37. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1998)
38. Roscoe, A.W.: Understanding Concurrent Systems. Springer, Berlin (2010)
39. Roscoe, A.W.: The automated verification of timewise refinement. In: First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering (2013)
40. Schneider, S.: Concurrent and Real-Time Systems. Wiley, London (2000)

41. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26–July 2, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5643, pp. 709–714. Springer, Berlin (2009)
42. Woodcock, J.C.P., Davies, J.: Using Z-Specification, Refinement, and Proof. Prentice-Hall, Upper Saddle River (1996)
43. Yi, W.: CCS + time = an interleaving model for real time systems. In: Albert, J.L., Monien, B., Artalejo, M.R. (eds.) Automata, Languages and Programming, pp. 217–228. Springer, Berlin (1991)