Progress-preserving Refinements of CTA

Massimo Bartoletti
Università degli Studi di Cagliari
Cagliari, Italy

Laura Bocchi
University of Kent
Canterbury, UK

Maurizio Murgia
University of Kent
Canterbury, UK

Abstract

We develop a theory of refinement for timed asynchronous systems, in the setting of Communicating Timed Automata (CTA). Our refinement applies point-wise to the components of a system of CTA, and only affecting their time constraints — in this way, we achieve compositionality and decidability. We then establish a decidable condition under which our refinement preserves behavioural properties of systems, such as their global and local progress. Our theory provides guidelines on how to implement timed protocols using the real-time primitives of programming languages. We validate our theory through a series of experiments, supported by an open-source tool which implements our verification techniques.

2012 ACM Subject Classification Theory of computation → Timed and hybrid models

Keywords and phrases protocol implementation, communicating timed automata, message passing

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.40

Funding This work has been partially supported by EPSRC EP/N035372/1 "Time-sensitive protocol design and implementation", and by Aut. Reg. of Sardinia project P.I.A. 2013 "NOMAD".

1 Introduction

Formal reasoning of real-time computing systems is supported by established theories and frameworks based on e.g., timed automata [4, 32, 44]. In the standard theory of timed automata, communication between components is *synchronous*: a component can send a message only when its counterpart is ready to receive it. However, in many concrete scenarios, such as web-based systems, communications are *asynchronous* and often implemented through middlewares supporting FIFO messaging [5, 42]. These systems can be modelled as Communicating Timed Automata (CTA) [29], an extension of timed automata with asynchronous communication. Asynchrony comes at the price of an increased complexity: interesting behavioural properties, starting from reachability, become undecidable in the general case, both in the timed [1, 22] and in the untimed [14] setting. Several works propose restrictions of the general model, or sound approximate techniques for the verification of CTA [11, 22]. These works leave one important problem largely unexplored: *the link between asynchronous timed models and their implementations*.

Relations between models at different levels of abstraction are usually expressed as *refinements*. These have been used, e.g., to create abstract models which enhance effectiveness of verification techniques (e.g., abstraction refinement [25, 43], time-wise refinement [40]), or to concretize abstract models into implementations [21, 23]. Existing notions of refinement between timed models are based on *synchronous* communications [7, 17, 26, 33]. Asynchronous refinement has been investigated in the *untimed* setting, under the name of subtyping between session types [8, 20, 24, 34–36].

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:1–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To our knowledge, no notion of refinement has been yet investigated in the *asynchronous timed* setting. The only work that studies a notion close to that of refinement is [12], which focusses on the relation between timed multiparty session types and their implementations (processes in an extended π-calculus). The work in [12] has two main limitations. First, their model is not as general as CTA: in particular, it does not allow states with both sending and receiving outgoing transitions (so-called *mixed states*). Mixed states are crucial to capture common programming patterns like *timeouts* [38] (e.g. a server waiting for a message that sends a timeout notification after a deadline). Some programming languages provide specific primitives to express timeouts, e.g. the receive/after construct of Erlang [6]. The second limitation of [12] is that its calculus is very simple (actions are statically set to happen at precise points in time), and cannot express common real-world blocking receive primitives (with or without timeout) that listen on a channel until a message is available.

To be usable in practice, a theory of refinements should support real-world programming patterns (e.g., timeouts *à la* Erlang) and primitives, and feature *decidable* notions of refinement. Further, refinement should be *compositional* (i.e. a system can be refined by refining its single components, independently), and preserve desirable properties (e.g., progress) of the system being refined. These goals contrast with the fact that, in general (e.g. when refinements may arbitrarily alter the interaction structures) establishing if an *asynchronous* FIFO-based communication model is a refinement of another is *undecidable*, even in the untimed setting [15,30]. Therefore, when defining an asynchronous refinement, a loss of generality is necessary to preserve decidability.

Contributions

We develop a theory of asynchronous timed refinement for CTA. Our main purpose is to study preservation of behavioural properties under refinement, focussing on two aspects: *timed behaviour* and *progress*. The former kind of preservation, akin timed similarity [18], ensures that the observable behaviour of the concrete system can be simulated by the abstract system. The latter requires that refinement does not introduce deadlocks, either *globally* (i.e., the whole system gets stuck), or *locally* (i.e., a single CTA gets stuck, although the whole system may still proceed).

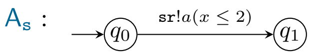
Refinement We introduce a new refinement relation, which is decidable and compositional, so enabling modular development of systems of CTA. Our refinement is *structure preserving*, i.e. it may only affect time constraints: refinements can only restrict them; further, for receive actions, refinements must preserve the deadline of the original constraint (i.e., the receiving component must be ready to receive until the very last moment allowed of the original constraint). This way of refining receive actions, and structure preservation, are key to obtain decidability and other positive results. Furthermore, structure preservation reflects the common practice of implementing a model: starting from a specification (represented as a system of CTA), one derives an implementation by following the interaction structure of the CTA, and by adjusting the timings of actions as needed, depending on implementation-related time constraints, and on the programming primitives one wants to use for each action (e.g., blocking/unblocking, with/without timeout). We illustrate in Section 6 how to exploit our theory in practice, to implement progress-preserving timed protocols in Go.

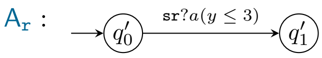
Positive and negative results Our main positive result (Theorem 26) is a decidable condition called *Locally Latest-Enabled Send Preservation* (LLESP) ensuring preservation of timed behaviour, global and local progress under our refinement. Our refinement and the LLESP condition naturally apply to most of the case studies found in literature (Section 4) In Section 6 we show how our tool and results can be used to guide the implementation of timed protocols with the Go programming language. We also considered other refinement strategies: (i) arbitrary restriction of constraints of send and receive actions (similarly to [12]), and (ii) asymmetric restriction where constraints of send actions may be restricted, and those of receive actions may be relaxed (this is the natural timed

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:2–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

extension of the subtyping relation in [24]). Besides being relevant in literature, (i) and (ii) reflect common programming practices: (i) caters for e.g. non-blocking receive with constraint reduced to an arbitrary point in the model's guard, and (ii) caters e.g. for blocking receive without timeouts. For (i) and (ii) we only have negative results, even when LLESP holds, and if mixed states are forbidden (Fact 27). Our negative results have a practical relevance on their own: they establish that if you implement a CTA as described above, you have no guarantees of behaviour/progress preservation.

A new semantics for CTA The original semantics for CTA [29] was introduced for studying decidability issues for timed languages. To achieve such goals, [29] adopts the usual language-based approach of computability theory: (1) it always allows time to elapse, even when this prevents the system from performing any available action, and (2) it rules out 'bad' executions *a posteriori*, e.g. only keeping executions that end in final states. Consider, for example, the following two CTA:





The CTA $\mathsf{A_s}$ models a sender $\mathtt{s}$ who wants to deliver a message a to a receiver $\mathtt{r}$. The guard $x \leq 2$ is a time constraint, stating that the message must be sent within 2 time units. The receiver wants to read the message a from $\mathtt{s}$ within 3 time units. In [29], a possible (partial) computation of the system $(\mathsf{A_s}, \mathsf{A_r})$ would be the following:

$$\gamma_0 = ((q_0, q_0'), (\varepsilon, \varepsilon), \{x, y \mapsto 0\}) \xrightarrow{\;5\;} ((q_0, q_0'), (\varepsilon, \varepsilon), \{x, y \mapsto 5\})$$

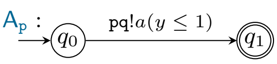
The tuple γ_0 at the LHS of the arrow is the initial *configuration* of the system, where both CTA are in their initial states; the pair $(\varepsilon, \varepsilon)$ means that the communication queues between $\mathtt{r}$ and $\mathtt{s}$ are empty; the last component means that the clocks x and y are set to 0. The label on the arrow represents a delay of 5 time units. This computation does *not* correspond to a reasonable behaviour of the protocol: we would expect the send action to be performed *before* the deadline expires.

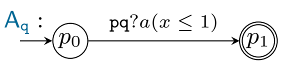
To capture this intuition, we introduce a semantics of CTA, requiring that the elapsing of time does not disable the send action in $\mathsf{A_s}$. Namely, we can procrastinate the send for 2 time units; then, time cannot delay further, and the only possible action is the send:

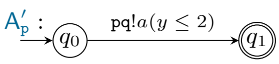
$$\gamma_0 \xrightarrow{\;2\;} ((q_0, q_0'), (\varepsilon, \varepsilon), \{x, y \mapsto 2\}) \xrightarrow{\mathtt{sr}!a} ((q_1, q_0'), (a, \varepsilon), \{x, y \mapsto 2\})$$

We prove (Theorem 7) that our semantics enjoys a form of *persistency*: if at least one receive action is guaranteed to be enabled in the future (i.e. a message is ready in its queue and its time constraint is satisfiable now or at some point in the future) then time passing preserves at least one of these guaranteed actions. Instead, time passing can disable all send actions, but *only if* it preserves at least one guaranteed receive.

It is well known that language-based approaches are not well suited to deal with concurrency issues like those addressed in this paper. To see this, consider the following CTA, where the states with a double circle are accepting:







The systems $S = (\mathsf{A_p}, \mathsf{A_q})$ and $S' = (\mathsf{A_p'}, \mathsf{A_q})$ accept the same language, namely $t_0\ \mathtt{pq}!a\ t_1\ \mathtt{pq}?a\ t_2$ with $t_0 + t_1 \leq 1$ and $t_2 \in \mathbb{R}_{\geq 0}$. So, the language-based approach does not capture a fundamental difference between S and S': S enjoys progress, while S' does not. Our approach to defining CTA semantics provides us with a natural way to reason on standard properties of protocols like progress, and to compare behaviours using e.g., (bi)simulation.

Our semantics allows for CTA with mixed states, by extending the one in [11] (where, instead, mixed states are forbidden). As said above, mixed states enable useful programming patterns. Consider e.g. the code snippet in Figure 1 (left), showing a typical use of the `receive`/`after` construct

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:3–40:34
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
receive {s,a1} -> Body1
                  ...
        {s,ak} -> Bodyk
after   10     -> p!b
```

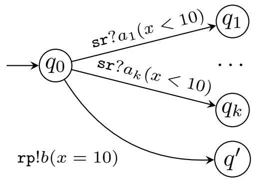


Figure 1 The `receive`/`after` pattern of Erlang (left), and the corresponding CTA (right).

in Erlang. The snippet attempts to receive a message matching one of the patterns `{s,`a_1`}`,...,`{s,`a_k`}`, where `s` represents the identifier of the sender, and $\mathtt{a}_1$,...,$\mathtt{a}_k$ are the message labels. If no such message arrives within 10 ms, then the process in the `after` branch is executed, sending immediately a message `b` to process `p`. This behaviour can be modelled by the CTA in Figure 1 (right), where q_0 is mixed. Our semantics properly models the intended behaviour of timeouts.

Urgency Another practical aspect that is not well captured by the existing semantics of CTA [11, 29] is *urgency*. Indeed, while in known semantics receive actions can be deferred, the receive primitives of mainstream programming languages unblock as soon as the expected message is available. These primitives include the non-blocking (resp. blocking) `WaitFreeReadQueue.read()` (resp. `WaitFreeReadQueue.waitForData()`) of Real-Time Java [16], and `receive`...`after` in Erlang, just to mention some. Analysing a system only on the basis of a non-urgent semantics may result in an inconsistence between the behaviour of the model and that of its implementation. To correctly characterise urgent behaviour, we introduce a second semantics (Definition 28), that is *urgent* in what it forces receive actions as soon as the expected message is available. Theorem 29 shows that the urgent semantics preserves the behaviour of the non-urgent. However, the urgent semantics does *not* enjoy the preservation results of Theorem 26. Still, it is possible to obtain preservation under refinement by combining Theorem 26 with Theorem 33. More specifically, the latter ensures that, if a system of CTA enjoys progress in the non-urgent semantics, then it will also enjoy progress in the urgent one, under a minor and common assumption on the syntax of time constraints. So, one can use Theorem 26 to obtain a progress-preserving refinement (in the non-urgent semantics), and then lift the preservation result to the urgent semantics through Theorem 33. Overall, our theory suggests that, despite the differences between semantics of CTA and programming languages, verification techniques based on CTA can be helpful for implementing distributed timed programs.

Artifact and experiments We validate our approach through a suite of use cases, which we analyse through a tool we have developed to experiment with our theory (`https://github.com/cta-refinement`). The suite includes real-world use cases, like e.g. SMTP [41] and Ford Credit web portal [39]. Experimentation shows that for each use case we can find a refinement which implements the specification in a correct way. All use cases require less than twenty control states, and our tool takes a few milliseconds to perform the analysis. In the absence of larger use cases in literature, we tried the tool on a deliberately large example with thousands of states and multiple clocks: even in that case, termination time is in the order of dozens of minutes. Because of space limitations, performance data are relegated to the appendix, as well as the proofs of our statements.

2 Communicating Timed Automata

We assume a finite set $\mathcal{P}$ of *participants*, ranged over by $\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \ldots$, and a finite set $\mathbb{A}$ of *messages*, ranged over by $a, b, \ldots$ We define the set $\mathcal{C}$ of *channels* as $\mathcal{C} = \{\mathtt{pq} \mid \mathtt{p}, \mathtt{q} \in \mathcal{P} \text{ and } \mathtt{p} \neq \mathtt{q}\}$. We denote with $\mathbb{A}^*$ the set of finite words on $\mathbb{A}$ (ranged over by $w, w', \ldots$), with ww' the concatenation of w and w', and with ε the empty word.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:4–40:34
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Clocks, guards and valuations. Given a (finite) set of *clocks* X (ranged over by $x, y, \dots$), we define the set Δ_X of *guards* over X (ranged over by $\delta, \delta', \dots$) as follows:

$$\delta ::= \mathit{true} \mid x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2 \qquad (c \in \mathbb{Q}_{\geq 0})$$

We denote with $\mathbb{V} = X \to \mathbb{R}_{\geq 0}$ the set of *clock valuations* on X. Given $t \in \mathbb{R}_{\geq 0}$, $\lambda \subseteq X$, and a clock valuation ν, we define the clock valuations: (i) $\nu + t$ as the valuation mapping each $x \in X$ to $\nu(x) + t$; (ii) $\lambda(\nu)$ as the valuation which resets to 0 all the clocks in $\lambda \subseteq X$, and preserves to $\nu(x)$ the values of the other clocks $x \notin \lambda$. Furthermore, given a set K of clock valuations, we define the *past* of K as the set of clock valuations $\downarrow K = \{\nu \mid \exists \delta \geq 0 : \nu + \delta \in K\}$.

Semantics of guards. We define the function $[\![\cdot]\!] : \Delta_X \to \wp(\mathbb{V})$ as follows:

$$[\![\mathit{true}]\!] = \mathbb{V} \qquad [\![x \leq c]\!] = \{\nu \mid \nu(x) \leq c\} \qquad [\![\delta_1 \wedge \delta_2]\!] = [\![\delta_1]\!] \cap [\![\delta_2]\!]$$
$$[\![\neg\delta]\!] = \mathbb{V} \setminus [\![\delta]\!] \qquad [\![c \leq x]\!] = \{\nu \mid c \leq \nu(x)\}$$

Actions. We denote with $\mathsf{Act} = \mathcal{C} \times \{!, ?\} \times \mathbb{A}$ the set of *untimed actions*, and with $\mathsf{TAct}_X = \mathsf{Act} \times \Delta_X \times 2^X$ the set of *timed actions* (ranged over by $\ell, \ell', \dots$). A (timed) action of the form $\mathtt{sr}!a(\delta, \lambda)$ is a *sending action*: it models a participant $\mathtt{s}$ who sends to $\mathtt{r}$ a message a, provided that the guard δ is satisfied. After the message is sent, the clocks in $\lambda \subseteq X$ are reset. An action of the form $\mathtt{sr}?a(\delta, \lambda)$ is a *receiving action*: if the guard δ is satisfied, $\mathtt{r}$ receives a message a sent by $\mathtt{s}$, and resets the clocks in $\lambda \subseteq X$ afterwards. Given $\ell = \mathtt{pr}!a(\delta, \lambda)$ or $\ell = \mathtt{qp}?a(\delta, \lambda)$, we define: (i) $\mathsf{msg}(\ell) = a$, (ii) $\mathsf{guard}(\ell) = \delta$, (iii) $\mathsf{reset}(\ell) = \lambda$, (iv) $\mathsf{subj}(\ell) = \mathtt{p}$, and (v) $\mathsf{act}(\ell)$ is $\mathtt{pr}!$ (in the first case) or $\mathtt{qp}?$ (in the second case). We omit δ if true, and λ if empty.

CTA and systems of CTA. A *CTA* A is a tuple of the form (Q, q_0, X, E), where Q is a finite set of *states*, $q_0 \in Q$ is the initial state, X is a set of clocks, and $E \subseteq Q \times \mathsf{TAct}_X \times Q$ is a set of *edges*, such that the set $\bigcup\{\mathsf{subj}(e) \mid e \in E\}$ is a singleton, that we denote as $\mathsf{subj}(\mathsf{A})$. We write $q \xrightarrow{\ell} q'$ when $(q, \ell, q') \in E$. We say that a state is *sending* (resp. *receiving*) if it has some outgoing sending (resp. receiving) edge. We say that A has *mixed states* if it has some state which is both sending and receiving. We say that a state q is *final* if there exist no ℓ and q' such that $(q, \ell, q') \in E$. *Systems* of CTA (ranged over by $S, S', \dots$) are sequences $(\mathsf{A}_\mathtt{p})_{\mathtt{p} \in \mathcal{P}}$, where each $\mathsf{A}_\mathtt{p} = (Q_\mathtt{p}, q_{0\mathtt{p}}, X_\mathtt{p}, E_\mathtt{p})$ is a CTA, and (i) for all $\mathtt{p} \in \mathcal{P}$, $\mathsf{subj}(\mathsf{A}_\mathtt{p}) = \mathtt{p}$; (ii) for all $\mathtt{p} \neq \mathtt{q} \in \mathcal{P}$, $X_\mathtt{p} \cap X_\mathtt{q} = \emptyset = Q_\mathtt{p} \cap Q_\mathtt{q}$.

Configurations. CTA in a system communicate via asynchronous message passing on FIFO queues, one for each channel. For each couple of participants $(\mathtt{p}, \mathtt{q})$ there are two channels, $\mathtt{pq}$ and $\mathtt{qp}$, with corresponding queues $w_{\mathtt{pq}}$ (containing the messages from $\mathtt{p}$ to $\mathtt{q}$) and $w_{\mathtt{qp}}$ (messages from $\mathtt{q}$ to $\mathtt{p}$). The state of a system S, or *configuration*, is a triple $\gamma = (\vec{q}, \vec{w}, \nu)$ where: (i) $\vec{q} = (q_\mathtt{p})_{\mathtt{p} \in \mathcal{P}}$ is the sequence of the current states of all the CTA in S; (ii) $\vec{w} = (w_{\mathtt{pq}})_{\mathtt{pq} \in \mathcal{C}}$ with $w_{\mathtt{pq}} \in \mathbb{A}^*$ is a sequence of queues; (iii) $\nu : \bigcup_{\mathtt{p} \in \mathcal{P}} X_\mathtt{p} \to \mathbb{R}_{\geq 0}$ is a *clock valuation*. The initial configuration of S is $\gamma_0 = (\vec{q_0}, \vec{\varepsilon}, \nu_0)$ where $\vec{q_0} = (q_{0\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$, $\vec{\varepsilon}$ is the sequence of empty queues, and $\nu_0(x) = 0$ for each $x \in \bigcup_{\mathtt{p} \in \mathcal{P}} X_\mathtt{p}$. We say that $(\vec{q}, \vec{w}, \nu)$ is *final* when all $q \in \vec{q}$ are final.

We introduce a new semantics of systems of CTA, that generalises Definition 9 in [11] to account for mixed states. To this aim, we first give a few auxiliary definitions. We start by defining when a guard δ' is satisfiable *later* than δ in a clock valuation.

▶ **Definition 1** (Later satisfiability). For all ν, we define the relation $\leq_\nu$ as:

$$\delta \leq_\nu \delta' \iff \forall t \in \mathbb{R}_{\geq 0} : \ \nu + t \in [\![\delta]\!] \implies \exists t' \geq t : \ \nu + t' \in [\![\delta']\!]$$

The following lemma states some basic properties of later satisfiability.

▶ **Lemma 2.** *The relation $\leq_\nu$ is a total preorder, for all clock valuations ν. Further, for all guards δ, δ', for all $t \in \mathbb{R}_{\geq 0}$, and $c, d \in \mathbb{Q}_{\geq 0}$: (a) $(x \leq c) \leq_\nu (x \leq c + d)$; (b) $\delta \wedge \delta' \leq_\nu \delta'$; (c) $\delta \leq_\nu \delta' \implies \delta \leq_{\nu+t} \delta'$.*

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:5–40:34
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Definition 3** (FE, LE, ND)**.** In a configuration $(\vec{q}, \vec{w}, \nu)$, we say that an edge $(q, \ell, q') \in E_{\mathtt{p}}$ is future-enabled (FE), latest-enabled (LE), or non-deferrable (ND) iff, respectively:

- $\exists t \in \mathbb{R}_{\geq 0}.\ \nu + t \in [\![\mathsf{guard}(\ell)]\!]$ (FE)
- $\forall \ell', q'' : (q, \ell', q'') \in E_{\mathtt{p}} \implies \mathsf{guard}(\ell') \leq_\nu \mathsf{guard}(\ell)$, and (q, ℓ, q') is FE (LE)
- $\exists \mathtt{s}, w' : \mathsf{act}(\ell) = \mathtt{sp}?, w_{\mathtt{sp}} = \mathsf{msg}(\ell) w'$ and (q, ℓ, q') is FE (ND)

An edge is FE when its guards can be satisfied at some time in the future; it is LE when no other edge (starting from the same state) can be satisfied later than it. The type of action (send or receive) and the co-party involved are immaterial to determine FE and LE edges. A receiving edge is ND when the expected message is already at the head of the queue, and there is some time in the future when it can be read. Note that an edge $(q, \mathtt{sp}?a(\delta, \lambda), q')$ is deferrable when $w_{\mathtt{sp}} = bw'$ and $a \neq b$ (i.e., the first message in the queue is not the expected one). Non-deferrability is not affected by the presence of send actions in the outgoing edges. It could happen that two receiving edges in a CTA are ND, if both expected messages are in the head of each respective queue.

The semantics of systems of CTA is defined in terms of a timed transition system (TLTS) between configurations.

▶ **Definition 4** (Semantics of systems)**.** Given a system S, we define the TLTS $[\![S]\!]$ as $(Q, \mathcal{L}, \rightarrow)$, where (i) Q is the set of configurations of S, (ii) $\mathcal{L} = \mathsf{Act} \cup \mathbb{R}_{\geq 0}$, (iii) $\gamma = (\vec{q}, \vec{w}, \nu) \xrightarrow{\alpha} (\vec{q'}, \vec{w'}, \nu') = \gamma'$ holds when one of the following rules apply:

1. $\alpha = \mathtt{sr}!a$, $(q_{\mathtt{s}}, \alpha(\delta, \lambda), q'_{\mathtt{s}}) \in E_{\mathtt{s}}$, and (a) $q'_{\mathtt{p}} = q_{\mathtt{p}}$ for all $\mathtt{p} \neq \mathtt{s}$; (b) $w'_{\mathtt{sr}} = w_{\mathtt{sr}} a$ and $w'_{\mathtt{pq}} = w_{\mathtt{pq}}$ for all $\mathtt{pq} \neq \mathtt{sr}$; (c) $\nu' = \lambda(\nu)$ and $\nu \in [\![\delta]\!]$;
2. $\alpha = \mathtt{sr}?a$, $(q_{\mathtt{r}}, \alpha(\delta, \lambda), q'_{\mathtt{r}}) \in E_{\mathtt{r}}$, and (a) $q'_{\mathtt{p}} = q_{\mathtt{p}}$ for all $\mathtt{p} \neq \mathtt{r}$; (b) $w_{\mathtt{sr}} = a w'_{\mathtt{sr}}$ and $w'_{\mathtt{pq}} = w_{\mathtt{pq}}$ for all $\mathtt{pq} \neq \mathtt{sr}$; (c) $\nu' = \lambda(\nu)$ and $\nu \in [\![\delta]\!]$;
3. $\alpha = t \in \mathbb{R}_{\geq 0}$, and (a) $q'_{\mathtt{p}} = q_{\mathtt{p}}$ for all $\mathtt{p} \in \mathcal{P}$; (b) $w'_{\mathtt{pq}} = w_{\mathtt{pq}}$ for all $\mathtt{pq} \in \mathcal{C}$; (c) $\nu' = \nu + t$; (d) for all $\mathtt{p} \in \mathcal{P}$, if some *sending* edge starting from $q_{\mathtt{p}}$ is LE in γ, then such edge is LE also in γ'; (e) for all $\mathtt{p} \in \mathcal{P}$, if some edge starting from $q_{\mathtt{p}}$ is ND in γ, then there exists an edge starting from $q_{\mathtt{p}}$ that is ND in γ'.

We write $\gamma \rightarrow \gamma'$ when $\gamma \xrightarrow{\alpha} \gamma'$ for some label α, and $\gamma \xrightarrow{\alpha}$ if $\gamma \xrightarrow{\alpha} \gamma'$ for some configuration γ'. We denote with $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$.

Rules (1), (2) and the first three items of (3) are adapted from [11]. In particular, (1) allows a CTA $\mathtt{s}$ to send a message a on channel $\mathtt{sr}$ if the time constraints in δ are satisfied by ν; dually, (2) allows $\mathtt{r}$ to consume a message from the channel, if δ is satisfied. In both rules, the clocks in λ are reset. Rule (3) models the elapsing of time. Items (a) and (b) require that states and queues are not affected by the passing of time, which is implemented by item (c). Items (d) and (e) put constraints on when time can pass. Condition (d) requires that time passing preserves LE sending edges: this means that if the current state of a CTA has the option to send a message (possibly in the future), time passing cannot prevent it to do so. Instead, condition (e) ensures that, if at least one of the expected messages is already at the head of a queue, time passing must still allow at least one of the messages already at the head of some queue to be received.

Our semantics (Definition 4) enjoys two classic properties [38] of timed systems, recalled below.

▶ **Definition 5.**

$$\gamma \xrightarrow{t} \gamma' \ \wedge\ \gamma \xrightarrow{t} \gamma'' \implies \gamma' = \gamma'' \qquad \text{(Time determinism)}$$
$$\gamma \xrightarrow{t+t'} \gamma' \iff \exists \tilde{\gamma} : \gamma \xrightarrow{t} \tilde{\gamma} \ \wedge\ \tilde{\gamma} \xrightarrow{t'} \gamma' \qquad \text{(Time additivity)}$$

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:6–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

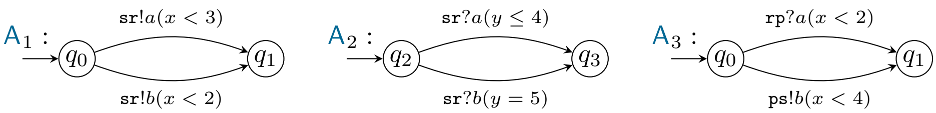


Figure 2 A collection of CTA, to illustrate the semantics of systems.

▶ **Lemma 6.** *The semantics of CTA enjoys time determinism and time additivity [38].*

Our semantics does not, instead, enjoy *persistency* [38], because the passing of time can suppress the ability to perform some actions. However, it enjoys a weaker persistency property, stated by Theorem 7. More specifically, if a receive action is ND, then time passing cannot suppress *all* receive actions: at least a ND action (not necessarily the first one) always remains FE after a delay. Instead, time passing can disable all send actions, but *only if* it preserves at least a ND receive action.

▶ **Theorem 7** (Weak persistency). *For all configurations γ, γ':*

$$\gamma \xrightarrow{t'} \xrightarrow{\mathtt{rp?}} \;\wedge\; \gamma \xrightarrow{t} \gamma' \implies \exists \gamma'', \mathtt{s}, t'' : \gamma' \xrightarrow{t''} \gamma'' \;\wedge\; \mathtt{p} \textit{ has a ND edge in } \gamma''$$
$$\gamma \xrightarrow{t'} \xrightarrow{\mathtt{pr!}} \;\wedge\; \gamma \xrightarrow{t} \gamma' \implies \exists \gamma'', \mathtt{s}, t'' : \gamma' \xrightarrow{t''} \gamma'' \;\wedge\; \mathtt{p} \textit{ has a FE sending edge or a ND edge in } \gamma''$$

Definition 8 below will be useful to reason on executions of systems.

▶ **Definition 8** (Maximal run). A *run* of a system S starting from γ is a (possibly infinite) sequence $\rho = \gamma_1 \xrightarrow{t_1} \gamma'_1 \xrightarrow{\alpha_1} \gamma_2 \xrightarrow{t_2} \cdots$ with $\gamma_1 = \gamma$ and $\alpha_i \in \mathsf{Act}$ for all i. We omit the clause "starting from s" when $\gamma = \gamma_0$. We call *trace* the sequence $t_1\,\alpha_1\,t_2 \cdots$. For all $n > 0$, we define the partial functions: $\mathit{conf}_n(\rho) = \gamma_n$, $\mathit{delay}_n(\rho) = t_n$, $\mathit{act}_n(\rho) = \alpha_n$. We say that a run is *maximal* when it is infinite, or given its last element γ_n it never happens that $\gamma_n \xrightarrow{t} \xrightarrow{\alpha}$, for any $t \in \mathbb{R}_{\geq 0}$ and $\alpha \in \mathsf{Act}$.

We show the peculiarities of our semantics through the CTA in Figure 2. First, consider the system composed of A_0 and A_0. A possible maximal run of $(\mathsf{A}_0, \mathsf{A}_0)$ from the initial configuration $\gamma_0 = ((q_0, q_2), \vec{\varepsilon}, \nu_0)$ is the following:

$$\gamma_0 \xrightarrow{2} \gamma_1 = ((q_0, q_2), (\varepsilon, \varepsilon), \nu_0 + 2) \qquad \xrightarrow{\mathtt{sr!}a} \gamma_2 = ((q_1, q_2), (a, \varepsilon), \nu_0 + 2)$$
$$\xrightarrow{1.5} \gamma_3 = ((q_1, q_2), (a, \varepsilon), \nu_0 + 3.5) \qquad \xrightarrow{\mathtt{sr?}a} \gamma_4 = ((q_1, q_3), (\varepsilon, \varepsilon), \nu_0 + 3.5)$$

The first delay transition is possible because there are no ND edges in A_0 (both edges are sending), and the LE edge $(q_0, \mathtt{sr!}a(x < 3), q_1)$, continues to be LE in $\nu_0 + 2$; further, in A_0 there are no LE sending edges, and no ND edges (since the queue $\mathtt{sr}$ is empty). Note that condition (d) prevents γ_0 from making transitions with label $t \geq 3$, since $(q_0, \mathtt{sr!}a(x < 3), q_1)$ is LE in γ_0, but it is *not* LE in $\nu_0 + t$ if $t \geq 3$. The transition from γ_1 to γ_2 corresponds to a send action. The delay transition from γ_2 to γ_3 is possible because the state of A_0 is final, while the state q_2 of A_0 has a ND edge, $(q_2, \mathtt{sr?}a(y \leq 4), q_3)$, which is still ND at $\nu_0 + 3.5$. Note instead that condition (e) prevents γ_2 from making a transition with $t > 2$, because no edge is ND in $\nu_0 + 2 + t$ if $t > 2$. Indeed, the last moment when the edge $(q_2, \mathtt{sr?}a(y \leq 4), q_3)$ is FE is $y = 4$. Finally, the transition from γ_3 to γ_4 corresponds to a receive action.

The CTA A_0 has mixed states, with the send action enabled for longer than the receive action. We show the behaviour of A_0 (abstracting from its co-parties that, we assume, always allow delays e.g. have all guards set to *true*). This CTA has a LE sending action $(q_0, \mathtt{ps!}b(x < 4), q_1)$ in the initial configuration γ_0. Hence, condition (d) is satisfied in γ_0 iff the delay t is less than 4. Condition (e) is satisfied in γ_0, as there are no ND edges. When A_0 is at state q_0, with $w_{\mathtt{rp}} = a$ and $\nu(x) = 0$, the CTA allows a delay t iff $t < 2$: later, no edge would be ND, so (e) would be violated. If the message a is in the queue but it is too late to receive it (i.e., $\nu(x) \geq 2$), then the receive action would be deferrable, and so a delay would be allowed — if condition (d) is respected.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:7–40:34
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3 Compositional asynchronous timed refinement

In this section we introduce a decidable notion of refinement for systems of CTA. Our system refinement is defined *point-wise* on its CTA. Point-wise refinement $\mathsf{A}' \sqsubseteq_1 \mathsf{A}$ only alters the guards, in the refined CTA A', while leaving the rest unchanged. The guards of A' — both in send and receive actions — must be narrower than those of A. Further, the guards in receive actions must have *the same past* in both CTA. Formally, to define the relation $\mathsf{A}' \sqsubseteq_1 \mathsf{A}$ we use *structure-preserving* functions that map the edges of A into those of A', preserving everything but the guards.

▶ **Definition 9** (Structure-preserving)**.** Let E, E' be sets of edges of CTA. We say that a function $f : E \to E'$ is *structure-preserving* when, for all $(q, \ell, q') \in E$, $f(q, \ell, q') = (q, \ell', q')$ with $\mathsf{act}(\ell) = \mathsf{act}(\ell')$, $\mathsf{msg}(\ell) = \mathsf{msg}(\ell')$, and $\mathsf{reset}(\ell) = \mathsf{reset}(\ell')$.

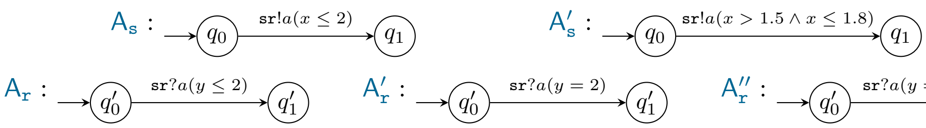
▶ **Definition 10** (Refinement)**.** Let $\mathsf{A} = (Q, q_0, X, E)$ and $\mathsf{A}' = (Q, q_0, X, E')$ be CTA. The relation $\mathsf{A}' \sqsubseteq_1 \mathsf{A}$ holds whenever there exists a structure-preserving isomorphism $f : E \to E'$ such that, for all edges $(q, \ell, q') \in E$, if $f(q, \ell, q') = \ell'$, then:
(a) $\llbracket \mathsf{guard}(\ell') \rrbracket \subseteq \llbracket \mathsf{guard}(\ell) \rrbracket$;
(b) if (q, ℓ, q') is a receiving edge, then $\downarrow \llbracket \mathsf{guard}(\ell') \rrbracket = \downarrow \llbracket \mathsf{guard}(\ell) \rrbracket$.

Condition (a) allows the guards of send/receive actions to be restricted. For receive actions, condition (b) requires restriction to preserve the final deadline.

System refinement reflects a modular engineering practice where parts of the system are implemented independently, without knowing how other parts are implemented.

▶ **Definition 11** (System Refinement)**.** Let $S = (\mathsf{A}_1, \dots, \mathsf{A}_n)$, and let $S' = (\mathsf{A}'_1, \dots, \mathsf{A}'_n)$. We write $S \sqsubseteq S'$ iff $\mathsf{A}_i \sqsubseteq_1 \mathsf{A}'_i$ for all $i \in 1 \dots n$.

▶ **Example 12.** With the CTA below, we have: $\mathsf{A}'_{\mathtt{s}} \sqsubseteq_1 \mathsf{A}_{\mathtt{s}}$, $\mathsf{A}'_{\mathtt{r}} \sqsubseteq_1, \mathsf{A}_{\mathtt{r}}$, and $\mathsf{A}''_{\mathtt{r}} \not\sqsubseteq_1 \mathsf{A}_{\mathtt{r}}$.



Theorem 13 establishes decidability of $\sqsubseteq_1$. This follows by the fact that CTA have a finite number of states and that: (i) the function $\downarrow \llbracket \delta \rrbracket$ is computable, and the result can be represented as a guard [10, 27]; (ii) the relation $\subseteq$ between guards is computable.

▶ **Theorem 13.** *Establishing whether* $\mathsf{A}' \sqsubseteq_1 \mathsf{A}$ *is decidable.*

We now formalise properties of systems of CTA that one would like to be preserved upon refinement. *Behaviour preservation*, which is based on the notion of timed similarity [18], requires that an implementation (refining system) at any point of a run allows *only* actions that are allowed by its specification (refined system). Below, we use $\uplus$ to denote the disjoint union of TLTSs, i.e. $(Q_1, \Sigma_1, \to_1) \uplus (Q_2, \Sigma_2, \to_2) = (Q_1 \uplus Q_2, \Sigma_1 \cup \Sigma_2, \{((i,q), a, (i,q')) \mid (q, a, q') \in \to_i\})$, where $Q_1 \uplus Q_2 = \{(i, q) \mid q \in Q_i\}$.

▶ **Definition 14** (Timed similarity)**.** Let $(Q, \mathcal{L}, \to)$ be a TLTS. A timed simulation is a relation $\mathcal{R} \subseteq Q \times Q$ such that, whenever $\gamma_1 \mathcal{R} \gamma_2$:

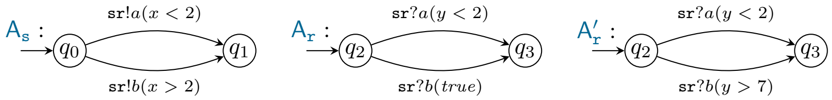
$$\forall \alpha \in \mathcal{L} \ : \ \gamma_1 \xrightarrow{\alpha} \gamma'_1 \implies \exists \gamma'_2 : \gamma_2 \xrightarrow{\alpha} \gamma'_2 \text{ and } \gamma'_1 \mathcal{R} \gamma'_2$$

We call *timed similarity* (in symbols, $\lesssim$) the largest timed simulation relation.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:8–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Definition 15** (Behaviour preservation). Let $\mathcal{R}$ be a binary relation between systems. We say that $\mathcal{R}$ *preserves behaviour* iff, whenever $S_1 \ \mathcal{R} \ S_2$, we have $(1, \gamma_0^1) \lesssim (2, \gamma_0^2)$ in the TLTS $[\![S_1]\!] \uplus [\![S_2]\!]$, where γ_0^1 and γ_0^2 are the initial configurations of S_1 and S_2.

▶ **Example 16** (Behaviour preservation). Let $\mathcal{R}$ be the inclusion of runs, let $S_1 = (\mathsf{A_s}, \mathsf{A'_r})$ and $S_2 = (\mathsf{A_s}, \mathsf{A_r})$, where:



We have that $S_2 \ \mathcal{R} \ S_1$, while $S_1 \ \mathcal{R} \ S_2$ does not hold, since the traces with b in S_1 strictly include those of S_2. The relation $\mathcal{R}$ preserves timed behaviour in $\{S_1, S_2\}$: indeed, $(\gamma_0^2, 1) \lesssim (\gamma_0^1, 2)$ follows by trace inclusion and by the fact that S_1, S_2 have deterministic TLTS. Now, let S_3 be as S_2, but for the guard of $\mathtt{sr}?b(true)$, which is replaced by $y < 2$. We have that $S_3 \ \mathcal{R} \ S_2$, and $\mathcal{R}$ preserves timed behaviour in $\{S_2, S_3\}$. However, S_3 does *not* allow to continue with the message exchange: b is sent too late to be received by $\mathtt{r}$, who keeps waiting while b remains in the queue forever. ◀

As shown by Example 16, behaviour preservation may allow a system (e.g., S_3) to remove "too much" from the runs of the original system (e.g., S_2): while ensuring that no new actions are introduced, it may introduce deadlocks. So, besides behaviour preservation we consider two other properties: *global* progress of the overall system, and *local* progress of each single participant.

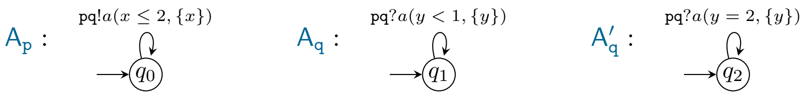
▶ **Definition 17** (Global/local progress). We say that a system S enjoys

- *global progress* when: $\forall \gamma : \gamma_0 \rightarrow^* \gamma$ not final $\implies \exists t \in \mathbb{R}_{\geq 0}, \alpha \in \mathsf{Act} : \gamma \xrightarrow{t} \xrightarrow{\alpha}$
- *local progress* when: $\forall \gamma, \mathsf{p} : \gamma_0 \rightarrow^* \gamma = (\vec{q}, \vec{w}, \nu)$ and $\vec{q} \ni q_{\mathsf{p}}$ not final $\implies$
  $\forall$ maximal runs ρ from $\gamma : \exists n : \mathsf{subj}(act_n(\rho)) = \mathsf{p}$

▶ **Lemma 18.** *If a system enjoys local progress, then it also enjoys global progress.*

The converse of Lemma 18 does not hold, as witnessed by Example 19.

▶ **Example 19** (Global vs. local progress). Consider the following CTA:



The system $(\mathsf{A_p}, \mathsf{A_q})$ enjoys global progress, since, in each reachable configuration, $\mathsf{A_p}$ can always send a message (hence the system makes an action in Act). However, if $\mathsf{A_p}$ sends a after time 1, then $\mathsf{A_q}$ cannot receive it, since its guard $y < 1$ is not satisfied. Formally, in any maximal run starting from $((q_0, q_1), (a, \varepsilon), \{x, y \mapsto 1\})$, there will be no actions with subject q, so $(\mathsf{A_p}, \mathsf{A_q})$ does *not* enjoy local progress. The system $(\mathsf{A_p}, \mathsf{A'_q})$, instead, enjoys both global and local progress. ◀

▶ **Definition 20** (Progress preservation). Let $\mathcal{R}$ be a binary relation between systems. We say that $\mathcal{R}$ *preserves global (resp. local) progress* iff, whenever $S_1 \ \mathcal{R} \ S_2$ and S_2 enjoys global (resp. local) progress, then S_1 enjoys global (resp. local) progress.

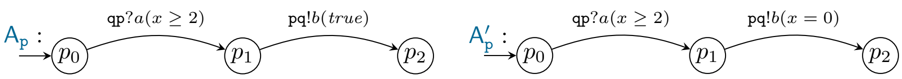
▶ **Example 21.** Let S_1, S_2, S_3 be as in Example 16. While S_1 and S_2 enjoy local and global progress, S_3 does not enjoy neither. Hence, $\mathcal{R} = \{(S_2, S_1), (S_3, S_1), (S_3, S_2)\}$ (i.e., trace inclusion restricted to the three given systems), does not preserve local nor global progress. ◀

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:9–40:34
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

4 Verification of properties of refinements

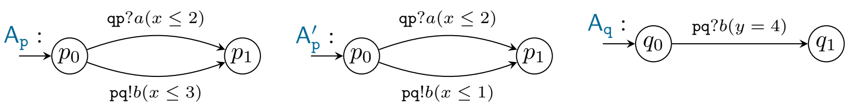
We now study preservation of behaviour/progress upon refinements. Our first result is negative: *in general*, refinement does not preserve behaviour nor (local/global) progress, even for CTA without mixed states. This is shown by the following examples.

▶ **Example 22.** Consider $\mathsf{A_p}$ and $\mathsf{A'_p}$ below, with $\mathsf{A'_p} \sqsubseteq_1 \mathsf{A_p}$.



When $\mathsf{A_p}$ reaches p_1, the guard of the outgoing edge is satisfiable. Instead, $\mathsf{A'_p}$ gets stuck in p_1.

▶ **Example 23.** Let $S = (\mathsf{A_p}, \mathsf{A_q})$, and let $S' = (\mathsf{A'_p}, \mathsf{A_q})$, where:



We have that $\mathsf{A'_p} \sqsubseteq_1 \mathsf{A_p}$, and so $S' \sqsubseteq S$. Behaviour is not preserved as S' allows the run $\gamma_0 \xrightarrow{4}$, while S does not. This is because $\mathsf{A_p}$ has a LE sending edge, which prevents step $\xrightarrow{4}$ by condition 3(d) of Definition 4, while $\mathsf{A'_p}$ does not have a LE sending edge. Progress (local and global) is enjoyed by S. Instead, S' does not enjoy progress: S' allows $\gamma_0 \xrightarrow{2} \gamma = ((p_0, q_0), \vec{\varepsilon}, \nu_0 + 2)$, but there are no t and $\alpha \in \mathsf{Act}$ such that $\gamma \xrightarrow{t}\xrightarrow{\alpha}$ as the sending action is expired and all the queues are empty. ◀

The issue in Example 23 is that a LE sending edge, which was crucial for making execution progress, is lost after the refinement. In Definition 25 we devise a decidable condition — which we call LLESP after *locally LE send preservation* — that excludes scenarios like the above. In Theorem 26 we show that, with the additional LLESP condition, $\sqsubseteq_1$ guarantees preservation of behaviour and progress. Unlike Definition 10, which is defined "edge by edge", LLESP is defined "state by state". This is because LLESP preserves the existence of LE sending edges (outgoing from the given state), and not necessarily the LE sending edge himself, making the analysis more precise.

▶ **Definition 24.** Let $\mathsf{A} = (Q, q_0, X, E)$, let $q \in Q$, and let K be a set of clock valuations. We define the following sets of clock valuations:

$$
\begin{aligned}
Pre^{\mathsf{A}}_q &= \{\nu_0 \mid q_0 = q\} \cup \{\nu \mid \exists q', \ell, \nu' : (q', \ell, q) \in E,\ \nu' \in [\![\mathsf{guard}(\ell)]\!],\ \nu = \mathsf{reset}(\ell)(\nu')\} \\
Les^{\mathsf{A}}_q &= \{\nu \mid q \text{ has a LE sending edge in } \nu\} \\
Post^{\mathsf{A}}_q(K) &= \left\{\nu + t \,\middle|\, \nu \in K \ \wedge\ (\nu \in Les^{\mathsf{A}}_q \implies \nu + t \in Les^{\mathsf{A}}_q)\right\}
\end{aligned}
$$

We briefly comment the auxiliary definition above. The set Les^{A}_q is self-explanatory, and its use is auxiliary to the definition of $Post$. Let $(\vec{q}, \vec{w}, \nu)$, where q is in $\vec{q}$, that can be reached by the initial configuration of some system S containing A. The set Pre^{A}_q contains all (but not only) the clock valuations under which a configuration like the one above can be reached with a label $\alpha \in \mathsf{Act}$ fired by A. Instead, $Post^{\mathsf{A}}_q(K)$ computes a symbolic step of timed execution, in the following sense: if $\nu \in K$ and $\gamma \xrightarrow{t} (\vec{q}, \vec{w}, \nu')$, where q is in $\vec{q}$, then $\nu' \in Post^{\mathsf{A}}_q(K)$. This is obtained by defining $Post^{\mathsf{A}}_q(K)$ as the set of clock valuations that would satisfy item (d) of Definition 4 for A at runtime, when starting from a configuration whose clock valuation is in K. Since every configuration reachable with a finite run and with an action in Act as last label can also be reached by a run ending with a delay (the original run followed by a null delay), the set $Post^{\mathsf{A}}_q(Pre^{\mathsf{A}}_q)$ contains the set of clock valuations ν such that $(\vec{q}, \vec{w}, \nu)$, with q is in $\vec{q}$, can be reached by the initial configuration of some system S containing A.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:10–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Definition 25** (LLESP)**.** A relation $\mathcal{R}$ is *locally LE send preserving* (in short, LLESP) iff, for all $\mathsf{A} = (Q, q_0, X, E)$ and $\mathsf{A}' = (Q, q_0, X, E')$ such that $\mathsf{A}' \mathrel{\mathcal{R}} \mathsf{A}$, and for all $q \in Q$: $Post_q^{\mathsf{A}'}(Pre_q^{\mathsf{A}'}) \cap Les_q^{\mathsf{A}} \subseteq Post_q^{\mathsf{A}'}(Pre_q^{\mathsf{A}'}) \cap Les_q^{\mathsf{A}'}$. We define $\sqsubseteq_1^L$ as the largest LLESP relation contained in $\sqsubseteq_1$.

Basically, LLESP requires that, whenever $\mathsf{A}' \mathrel{\mathcal{R}} \mathsf{A}$, if q has a LE sending edge in ν with respect to A, then q has a LE sending edge in ν with respect to A', where ν ranges over elements of $Post_q^{\mathsf{A}'}(Pre_q^{\mathsf{A}'})$.

It follows our main result: $\sqsubseteq_1^L$ preserves behaviour and progress (both global and local). Further, LLESP is decidable, so paving the way towards automatic verification.

▶ **Theorem 26** (Preservation under LLESP)**.** *$\sqsubseteq_1^L$ preserves behaviour, and global and local progress. Furthermore, establishing whether $\mathsf{A}' \sqsubseteq_1^L \mathsf{A}$ is decidable.*

Negative results on alternative refinement strategies Besides introducing a new refinement we have investigated behavioural/progress preservation under two refinement strategies inspired from literature. They are both variants of our definition of refinement that alter conditions (a) and (b) in Definition 10. The first strategy (e.g., [12]) is a naïve variant of Definition 10 where (b) is dropped. The second strategy (e.g., [24]) is an asymmetric variant of Definition 10 that allows to *relax* guards of the receive actions: (a) is substituted by $[\![\mathsf{guard}(\ell')]\!] \supseteq [\![\mathsf{guard}(\ell)]\!]$ and (b) is dropped.

▶ Fact 27. LLESP restrictions of 'naïve' and 'asymmetric' refinements do not preserve behaviour, global progress, nor local progress, not even if mixed states are ruled out.

Counter-examples of behaviour/progress preservation for LLESP restrictions of 'naïve' and 'asymmetric' refinements without mixed states are relegated to Appendix C.4 (Examples 66 and 67). Note that Example 23 (which has mixed states) is also a counter-example for such refinements.

Experiments We evaluate our theory against a suite of protocols from literature. To support the evaluation we built a tool that determines, given A and A', if $\mathsf{A}' \sqsubseteq_1 \mathsf{A}$ and if $\mathsf{A}' \sqsubseteq_1^L \mathsf{A}$. For each participant of each protocol we construct three refinement *strategies*. For sending edges, if the guard has an upper bound (e.g. $x \leq 10$) then we refine it with, respectively: (strategy #1) the lower bound (e.g. $x = 0$), (strategy #2) the average value (e.g. $x = 5$), and (strategy #3) the upper bound (if any) (e.g. $x = 10$). In all strategies, receiving edges are refines in the same way: if the guard has a not strict upper bound (e.g. $x \leq 10$) then we restrict the guard *as* its upper bound (e.g. $x = 10$); if the upper bound is strict (e.g. $x < 10$) we 'procrastinate' the guard, but making it fully left-closed (Definition 31) (e.g. $10 - \varepsilon \leq x < 10$, where we set ε as a unit of time); if there is no upper bound (e.g. $x > 10$) the guard is left unchanged. Our tool correctly classifies the pairs of CTA defined above as refinements. In Table 1 we show the output of the tool when checking LLESP. We can see that strategies #2 and #3 never break the LLESP property. While this should always hold for strategy #3 (procrastinating sending edges guarantees that LE sending edges are preserved), the case for strategy #2 is incidental. Among the case studies, Ford Credit web portal and SMTP contain mixed states (used to implement timeouts). The fact that, for each protocol, there is always some refinement strategy that satisfies LLESP (hence a provably safe way to implement that protocol) witnesses the practicality of our theory. Surprisingly, the states that falsify LLESP are not mixed. The three models for which strategy #1 does not produce 'good' refinements suffer from the same issue of Example 22: the guard of a sending edge is restricted in a way that makes it possibly unsatisfiable with respect to the guard of the previous action.

5 Preservation under an urgent semantics

The semantics in Definition 4 does not force the receive actions to happen, (unless time passing prevents the CTA from receiving in the future, by condition 3(e). This behaviour, also present

Case study	Strategy #1	Strategy #2	Strategy #3
Ford Credit web portal [39]	✗Server	✓ Server	
Scheduled Task Protocol [11]	✓User ✓Worker ✓Aggregator	✓User ✓Worker ✓Aggregator	✓User ✓Worker ✓Aggregator
OOI word counting [37]	✓Master ✓Worker ✓Aggregator	✓Master	✓Master
ATM [19]	✗Bank, ✓User ✗Machine	✓Bank ✓User ✓Machine	✓Bank ✓User
Fisher Mutual Exclusion [9]	✓Producer ✓Consumer	✓Producer	✓Producer
SMTP [41]	✓Client	✓Client	

Table 1 Benchmarks. Participants satisfying LLESP are marked with ✓, the others with ✗. We omitted participants for which the strategy was not meaningful, or gave identical results as the other columns.

in [11, 29], contrasts with the actual behaviour of the `receive` primitives of mainstream programming languages which return *as soon as* a message is available. We now introduce a variant of the semantics in Definition 4 which faithfully models this behaviour. We make receive actions *urgent* [13, 38] by forbidding delays when a receiving edge is enabled and the corresponding message is at the head of the queue. Below, $\mathsf{Act}^?$ denotes the set of input labels.

▶ **Definition 28** (Urgent semantics of systems). Given a system S, we define the TLTS $[\![S]\!]_u = (Q, \mathcal{L}, \rightarrow_u)$, where Q is the set of configurations of S, $\mathcal{L} = \mathsf{Act} \cup \mathbb{R}_{\geq 0}$, and:

$$\gamma \xrightarrow{\alpha}_u \gamma' \iff \begin{cases} \gamma \xrightarrow{\alpha} \gamma' & \text{if } \alpha \in \mathsf{Act} \\ \gamma \xrightarrow{t} \gamma' & \text{if } \alpha = t \text{ and } \forall t' < t, \gamma'', \alpha' \in \mathsf{Act}^? : \gamma \xrightarrow{t'} \gamma'' \implies \gamma'' \not\xrightarrow{\alpha'} \end{cases}$$

The non-urgent and the urgent semantics are very similar: they only differ in time actions. In the urgent semantics, a system can make a time action t only if no receive action is possible earlier than t (hence no message is waiting in a queue with 'enabled' guard). Theorem 29 formally relates the two semantics. Since the urgent semantics restricts the behaviour of systems (by dropping some timed transitions), the urgent semantics preserves the behaviour of the non-urgent one.

▶ **Theorem 29.** *For all systems S, the relation $\{((1,\gamma),(2,\gamma)) \mid \gamma$ is a configuration of $S\}$ between states of $[\![S]\!]_u \uplus [\![S]\!]$ is a timed simulation.*

In general, however, a system that enjoys progress with the non-urgent semantics may not enjoy progress with the urgent one. This is illustrated by Example 30.

▶ **Example 30.** Consider the system $S = (\mathsf{A_s}, \mathsf{A_r})$, where:

With the non-urgent semantics, $\gamma_0 \xrightarrow{\texttt{sr}!a}\xrightarrow{3} \gamma = ((q_1, q_0'), (a, \varepsilon), \nu_0 + 3) \xrightarrow{t}\xrightarrow{\texttt{sr}?a}$, for all $t \in \mathbb{R}_{\geq 0}$. With the urgent semantics, $\gamma_0 \xrightarrow{\texttt{sr}!a}_u \xrightarrow{3}_u \gamma \not\xrightarrow{\alpha}_u$, for all $\alpha \neq 0$. Hence, the non-urgent semantics leads to a final state, whereas the urgent semantics does not. ◀

The issue highlighted by Example 30 is subtle (but known in literature [13]): if there is no precise point in time in which a guard becomes enabled (e.g. in $x > 3$), then the run may get stuck. In Definition 31 we deal with this issue through a restriction on guards, which guarantees that urgent semantics preserves progress. Our restriction, generalising the notion of *right-open time progress* [13] (to deal with non-convex guards), corresponds to forbidding guards defined as the conjunction of sub-guards of the form $x > c$ (but we allow subguards of the form $x \geq c$). To keep our results independent from the syntax of guards, our definition is based on sets of clock valuations.

▶ **Definition 31** (Fully left closed). For all ν, and for all sets of clock valuations K, let $D_\nu(K) = \{t \mid \nu + t \in K\}$ and let $\inf Z$ denote the infimum of Z. We say that a guard δ is *fully left closed* iff: $\forall \nu : \forall K \subseteq [\![\delta]\!] : \big(D_\nu(K) \neq \emptyset \implies \nu + \inf D_\nu(K) \in [\![\delta]\!]\big)$. We say that a CTA is *input fully left closed* when all guards in its receiving edges are fully left closed. A system is input fully left closed when all its components are such.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:12–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

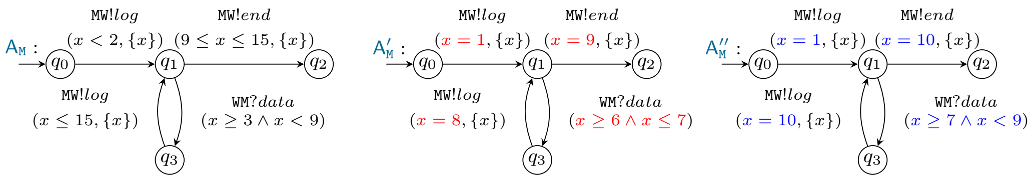


Figure 3 $\mathsf{A}_{\mathtt{M}}$ (left); $\mathsf{A}'_{\mathtt{M}} \not\sqsubseteq_1 \mathsf{A}_{\mathtt{M}}$ (centre); $\mathsf{A}''_{\mathtt{M}} \sqsubseteq_1 \mathsf{A}_{\mathtt{M}}$ (right).

Fully left closed guards ensure that there is an exact time instant in which a guard of an urgent action becomes enabled. The requirement that left closedness must hold for any subset K of the semantics of the guards is needed to cater for non-convex guards (i.e. guards with disjunctions). Consider e.g. $\delta = 1 \leq x \leq 3 \vee x > 4$. While δ is left closed, it is *not* fully left closed: indeed, for $K = [\![x > 4]\!] \subseteq [\![\delta]\!]$, it holds that $\inf D_{\nu_0}(K) = 4$, but $\nu + 4 \notin [\![\delta]\!]$.

▶ **Example 32.** The guard $x > 3$ in Example 30 is not fully left closed, as $\inf D_{\nu_0}([\![x > 3]\!]) = \inf \{t \mid t > 3\} = 3$, but $\nu_0 + 3 \notin [\![x > 3]\!]$. Instead, guard $x \geq 3$ is fully left closed. Consider now a variant of the system of Example 30 where guard $x > 3$ is replaced by $x \geq 3$. The run $\gamma_0 \xrightarrow{\mathtt{sr}!a}_u \xrightarrow{3}_u \gamma$ would not get stuck and allow $\gamma \xrightarrow{\mathtt{sr}?a}$. ◀

The following theorem states that urgent semantics preserves progress with respect to non-urgent semantics, when considering fully left closed systems.

▶ **Theorem 33** (Preservation of progress vs. urgency). *Let S be input fully left closed. If S enjoys global (resp. local) progress under the non-urgent semantics, then S enjoys global (resp. local) progress under the urgent semantics.*

6 Implementing protocols via refinement

We illustrate how to exploit our theory to implement timed protocols, by considering the real-world protocol in [37], which distributedly counts the occurrences of a word in a log. Because of space limitations, we slightly simplify and adapt the protocol in [37]. The full protocol is in Appendix D. The system has two nodes: a master `M` and a worker `W`. We focus on `M`, modelled as $\mathsf{A}_{\mathtt{M}}$ in Figure 3 (left). $\mathsf{A}_{\mathtt{M}}$ repeatedly: sends a log to $\mathsf{A}_{\mathtt{W}}$, then either receives data from $\mathsf{A}_{\mathtt{W}}$ (within timeout $x < 9$) or sends a notice and terminates. We implement the CTA in Go, a popular programming language with concurrency features. Here, we just sketch an implementation which intuitively follows the CTA model. A rigorous correspondence between the Go primitives and the CTA model (supporting e.g., automatic code generation) is a future work that is out of the scope of this paper. We use: (i) variables of type `time.Time` as clocks (e.g., `x`), and (ii) function `rel` below to return the value (of type `time.Duration`) of a clock (since the last reset):

```
func rel(x time.Time) time.Duration {return time.Now().Sub(x)}
```

A naïve implementation in Go We first attempt to implement $\mathsf{A}_{\mathtt{M}}$ following $\mathsf{A}'_{\mathtt{M}}$ (Figure 3). $\mathsf{A}'_{\mathtt{M}}$ is obtained from $\mathsf{A}_{\mathtt{M}}$ by restricting guards obliviously of our results. We start from the edge from q_0 to q_1, assuming that the preparation of the log to send takes 1s (with negligible jitter). This could result in the snippet below:

```
x := time.Now() // initial setting of clock x
time.Sleep(time.Second * 1 - rel(x)) // sleep for 1s
x = time.Now() // reset x
MW <- "log" // send string "log" on FIFO channel MW
```


29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:13–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The statement in line 2 represents the invocation of a time-consuming function that prepares the log to be sent in line 4 (here we send the string `"log"`). In general, implementations may be informed by estimated durations of code instructions. Providing such information is made possible by orthogonal research on cost analysis, e.g. [28]. Next, we want to (i) implement the receive action from q_1 to q_3 as a *blocking* primitive with timeout, (ii) minimise the waiting time of the master listening on the channel, and restrict the interval to $x \geq 6 \wedge x \leq 7$. This could result in the following:

```
1   time.Sleep(time.Second *6 - rel(x))
2   select { case res := <- WM:
3              // here goes the implementation of edge q3 ---> q1
4            case <- time.After(time.Second * 7 + time.Nanosecond * 1 - rel(x)):
5              // here goes the implementation of edge q1 ---> q2
```

Note that without the addition of one nanosecond in line 4 above the snippet would implement a constraint $(x \geq 6 \wedge x < 7)$. To enable the program to read the message when $x = 7$, we add the smallest time unit in Go, which is negligible with respect to the protocol delays. The study of implementability of such equality constraints at this granularity of time is left as future work.

Next, we implement the edge from q_1 to q_2 by substituting line 5 above with:

```
time.Sleep(time.Second *9 - rel(x))
x = time.Now() // reset x
MW <- "end" // send string "end"
```

The edge from q_3 to q_1 can be implemented in a similar way, where the sleep statement represents a time-consuming log preparation of 1s, as before.

Assessing implementations via our tool The implementation sketched in the previous paragraphs corresponds to $\mathsf{A}'_{\mathtt{M}}$ (Figure 3). Analysis of $\mathsf{A}'_{\mathtt{M}}$ with our tool reveals that $\mathsf{A}'_{\mathtt{M}} \not\sqsubseteq_1 \mathsf{A}_{\mathtt{M}}$: the constraints of receiving edges of $\mathsf{A}_{\mathtt{M}}$ have been restricted not respecting the final deadlines. From Section 4 we know that $\mathsf{A}'_{\mathtt{M}}$ may *not* preserve behaviour and progress. Suppose that the worker node is set to send the data to $\mathsf{A}_{\mathtt{M}}$ when $x = 8.5$: according to the original specification $\mathsf{A}_{\mathtt{M}}$, this message is in time, hence the worker will expect a log message back from the master. However, in the implementation reflected in $\mathsf{A}'_{\mathtt{M}}$, the master will reply with an end message, potentially causing a deadlock. Thanks to Theorem 26 we know that we can, instead, safely restrict the constraints using $\sqsubseteq_1$: guard $x \geq 6 \wedge x \leq 7$ of $\mathsf{A}'_{\mathtt{M}}$ can be amended as $x \geq 7 \wedge x < 9$. After this amendment, however, the tool detects a violation of LLESP: the deadlines set by guards of sending edges from q_3 and q_1 are after the deadline of the receive action. A correct refinement $\mathsf{A}''_{\mathtt{M}} \sqsubseteq_1^L \mathsf{A}_{\mathtt{M}}$ is shown in Figure 3 (right) and can be used to produce the following implementation in Go:

```
MW := make(chan string, 100)
WM := make(chan string, 100)
go func(){
  // q0 ---> q1
  x := time.Now()
  time.Sleep(time.Second *1 - rel(x))
  x = time.Now()
  MW <- "log"
  // q1 ---> q3
  time.Sleep(time.Second *7 - rel(x))
  select {
    case res := <- WM:
      // q3 ---> q2
      x = time.Now()
      time.Sleep(time.Second *10 - rel(x))
      MW <- "log"
    case <- time.After(time.Second *9 - rel(x)):
      // q1 ---> q2
      time.Sleep(time.Second *10 - rel(x))
      x = time.Now()
      MW <- "end"  }}()
  }
```

Practicality In some scenarios, one may want to implement receive actions with *non-blocking* primitives (unlike above, where we have used blocking ones). Non-blocking primitives can be modelled as CTA refinements where constraints (e.g., $x \leq 9$) are restricted to a point in time (e.g., $x = 9$). Punctual guards can be attained in the real world by assuming a tolerance (e.g., around 9) that is negligible against the scale of x. In some cases, it may be desirable to *not* restrict the constraint of receive actions, to be able to receive a message as soon as possible.

CTA can capture delays of the communication medium e.g., by adding them at the receiver side. This is common when using semantics where actions are timeless and delays are modelled separately, as these semantics can be encoded into ones where actions have an associated duration.

Our theory can be applied to non real-time operating systems and languages (like, e.g., Go), as long as the time granularity of the modelled protocols is coarse enough with respect to the jitter of the operating system / language. However, negligible delays may accumulate, eventually compromising the correctness of long-lived protocols. In this case, adjustments like e.g. those suggested in [37] or based on analysis on the robustness of protocols to jitters [31], may be in order to recover correctness.

7 Conclusions

Our theory provided a formal basis to support implementation of well-behaved systems from well-behaved models. This is obtained through a decidable refinement relation, and a condition (LLESP) that guarantees behaviour and progress preservation. To overcome the undecidability results of refinement in asynchronous models [15, 30], we considered "purely timed" refinements, that only affect time constraints. While not fully general, our refinement captures the practical relations between models, and implementations obtained by following them (Section 6). Moreover, our refinement and the LLESP condition apply well to realistic protocols expressed as CTA (Section 4): for each participant of each protocol in our portfolio, there exist one or more non-trivial (i.e. not the identity) LLESP refinements, from which one can derive behaviour- and progress-preserving implementations of that protocol. Evaluating our theory was facilitated by a tool, that can also be used to guide implementations. Being this the first work which enables refinements between CTA, there is no benchmark against which to study limitations or compare with. Other "purely timed" refinements strategies inspired by literature gave only negative results (Fact 27) when applied to the asynchronous timed setting, hence e.g., even if an implementation preserves the interactions structure of the initial CTA, and even if the timings of actions chosen for the implementation are within the range of the guards of the initial CTA, still that implementation may not preserve behaviour or progress.

Technically, we focused on interaction-based (rather than language-based) semantics, improving the state of the art in two ways: mixed choices and urgency. Mixed choices cannot be expressed in models based on session types of [11, 12]. There, interactions follow two constructs: *selection*, which corresponds to an internal choice of send actions, and *branching*, an external choice of receive actions. The behaviour of mixed states captured by our semantics falls somewhere in between internal and external choices, so it is not expressible in the setting of [11, 12]. Besides, the known semantics [11, 12, 29] do not account for urgency. Our preservation results from non-urgent to urgent semantics pave the way to *implementations* of refinements that preserve behaviour and progress (e.g. derived incrementally using the non-urgent semantics, and relying on the results in Section 4).

Other work on relating timed models with implementations is, e.g. [2, 3]. The work [2] approximates dense time models in synchronous models with fixed sampling rates, so to enable for hardware implementations. Here, instead, we considered asynchronous models, and delays at a coarser granularity, aiming at time-sensitive (not necessarily real-time) languages. The work [3] generates Erlang code from real-time Rebeca models (so, focussing on the actor model, rather than on FIFO channels). Extending our tool in this direction is an ongoing work of ours.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. What is decidable about perfect timed channels? *CoRR*, abs/1708.05063, 2017. arXiv:1708.05063.

2 Parosh Aziz Abdulla, Pavel Krcál, and Wang Yi. Sampled semantics of timed automata. *Logical Methods in Computer Science*, 6(3), 2010. URL: http://arxiv.org/abs/1007.2783.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:15–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3 Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed Rebeca. *Sci. Comput. Program.*, 89:41–68, 2014.

4 Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

5 Advanced Message Queuing protocols (AMQP) homepage. https://www.amqp.org/.

6 Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

7 Massimo Bartoletti, Tiziana Cimoli, and Maurizio Murgia. Timed session types. *Logical Methods in Computer Science*, 13(4), 2017. doi:10.23638/LMCS-13(4:25)2017.

8 Massimo Bartoletti, Alceste Scalas, and Roberto Zunino. A semantic deconstruction of session types. In *CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 402–418. Springer, 2014. doi:10.1007/978-3-662-44584-6_28.

9 Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. doi:10.1007/b110123.

10 Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. doi:10.1007/b98282.

11 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.CONCUR.2015.283.

12 Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2014. doi:10.1007/978-3-662-44584-6_29.

13 Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer, 1997. doi:10.1007/3-540-49213-5_5.

14 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

15 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.

16 Eric J. Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

17 Stefano Cattani and Marta Z. Kwiatkowska. A refinement-based process algebra for timed automata. *Formal Asp. Comput.*, 17(2):138–159, 2005.

18 Karlis Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer, 1992. doi:10.1007/3-540-56496-9.

19 Prakash Chandrasekaran and Madhavan Mukund. Matching scenarios with timing constraints. In *FORMATS*, volume 4202 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2006. doi:10.1007/11867340.

20 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Logical Methods in Computer Science*, 13(2), 2017.

21 Chris Chilton, Marta Z. Kwiatkowska, and Xu Wang. Revisiting timed specification theories: A linear-time perspective. In *FORMATS*, volume 7595 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2012. doi:10.1007/978-3-642-33365-1_7.

22 Lorenzo Clemente, Frédéric Herbreteau, Amélie Stainer, and Grégoire Sutre. Reachability of communicating timed processes. In *FoSSaCS*, volume 7794 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2013. doi:10.1007/978-3-642-37075-5_6.

23 Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, Louis-Marie Traonouez, and Andrzej Wasowski. Real-time specifications. *STTT*, 17(1):17–45, 2015. doi:10.1007/s10009-013-0286-x.

24 Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 280–296. Springer, 2011. doi:10.1007/978-3-642-23217-6_19.

25 Henning Dierks, Sebastian Kupferschmid, and Kim Guldstrand Larsen. Automatic abstraction refinement for timed automata. In *FORMATS*, volume 4763 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007. doi:10.1007/978-3-540-75454-1_10.

26 Harald Fecher, Mila E. Majster-Cederbaum, and Jinzhao Wu. Refinement of actions in a real-time process algebra with a true concurrency model. *Electr. Notes Theor. Comput. Sci.*, 70(3):260–280, 2002. doi:10.1016/S1571-0661(05)80496-7.

27 Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

28 Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 132–157. Springer, 2015. doi:10.1007/978-3-662-46669-8.

29 Pavel Krcál and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2006. doi:10.1007/11817963.

30 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In *FoSSaCS*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7.

31 Kim G. Larsen, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. Robust synthesis for real-time systems. *Theor. Comput. Sci.*, 515:96–122, 2014. URL: https://doi.org/10.1016/j.tcs.2013.08.015, doi:10.1016/j.tcs.2013.08.015.

32 Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

33 Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003.

34 Dimitris Mostrous. *Session Types in Concurrent Calculi: Higher-Order Processes and Objects*. PhD thesis, Imperial College London, November 2009.

35 Dimitris Mostrous and Nobuko Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2009. doi:10.1007/978-3-642-02273-9_16.

36 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009.

37 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017. doi:10.1007/s00165-017-0420-8.

38 Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In *CAV*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer, 1991.

39 Julien Ponge, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and applications of timed service protocols. *ACM Trans. Softw. Eng. Methodol.*, 19(4):11:1–11:38, 2010.

40 Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

41 The Simple Mail Transfer Protocol. http://tools.ietf.org/html/rfc5321.

42 Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:17–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

43 Weifeng Wang and Li Jiao. Trace abstraction refinement for timed automata. In *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2014. doi:10.1007/978-3-319-11936-6.

44 Sergio Yovine. Kronos: A verification tool for real-time systems. (Kronos user's manual release 2.2). *STTT*, 1(1-2):123–133, 1997. doi:10.1007/s100090050009.

A Proofs and additional material for Section 2

▶ **Definition 34** (Timed LTS). A timed labelled transition system (in short, TLTS) is a triple $(Q, \mathcal{L}, \rightarrow)$, where Q is the set of *states*, $\mathcal{L} \supseteq \mathbb{R}_{\geq 0}$ is the set of *labels*, and $\rightarrow \subseteq Q \times \mathcal{L} \times Q$ is the *transition relation*. We use $\alpha, \beta, \ldots$ to range over $\mathcal{L}$.

▶ **Example 35.** In the initial clock valuation ν_0, we have that:

$$(x \leq 2) \ \leq_{\nu_0} \ (x \leq 3) \quad (x \leq 2) \ \leq_{\nu_0} \ (x = 3) \quad (x \leq 3) \ \not\leq_{\nu_0} \ (x \leq 2)$$

Note that the relation $\leq_\nu$ is *not* antisymmetric: e.g., for all $c, c' \in \mathbb{R}_{\geq 0}$, we have that $(x > c) \leq_\nu (x > c') \leq_\nu (x > c)$, even if $c \neq c'$.

Proof of Lemma 2

Proving reflexivity and transitivity is straightforward. To prove that $\leq_\nu$ is total, we show that if $\delta \not\leq_\nu \delta'$, then it must be $\delta' \leq_\nu \delta$. Since $\delta \not\leq_\nu \delta'$, then there exists some $t_0 \in \mathbb{R}_{\geq 0}$ such that $\nu + t_0 \in [\![\delta]\!]$, but:

$$\nexists t' \geq t_0 : \nu + t' \in [\![\delta']\!] \tag{1}$$

To prove $\delta' \leq_\nu \delta$, let $t' \in \mathbb{R}_{\geq 0}$ such that $\nu + t' \in [\![\delta']\!]$ (if no such t' exists, we trivially obtain the thesis). By (1), it must be $t' < t_0$. Hence, we have found a $t_0 > t'$ such that $\nu + t_0 \in [\![\delta]\!]$, from which we conclude that $\delta' \leq_\nu \delta$.

For item (a), assume that $d \geq 0$. Let $t \in \mathbb{R}_{\geq 0}$ be such that $\nu + t \in [\![x \leq c]\!]$. Then, $\nu(x) + t \leq c$, and so $\nu(x) + t \leq c + d$. Choosing $t' = t$, we have $\nu + t' \in [\![x \leq c + d]\!]$ from which we obtain the thesis $(x \leq c) \leq_\nu (x \leq c + d)$. To prove the converse, assume that $d < 0$. Let $t = c - \nu(x)$. Then, $\nu(x) + t \leq c$, but there exists no $t' \geq t$ such that $\nu(x) + t' \leq c + d$. Hence, $(x \leq c) \not\leq_\nu (x \leq c + d)$.

For item (b), let $t \in \mathbb{R}_{\geq 0}$ be such that $\nu + t \in [\![\delta \wedge \delta']\!] = [\![\delta]\!] \cap [\![\delta']\!]$. Choose $t' = t$. Then, $\nu + t \in [\![\delta']\!]$, from which we obtain the thesis $\delta \wedge \delta' \leq_\nu \delta'$.

For item (c), assume that $\delta \leq_\nu \delta'$. Let $\tilde{t} \in \mathbb{R}_{\geq 0}$ be such that $(\nu + t) + \tilde{t} \in [\![\delta]\!]$. Since $\delta \leq_\nu \delta'$, there exists $\tilde{t}' \geq t + \tilde{t}$ such that $\nu + \tilde{t}' \in [\![\delta']\!]$. Choose $t' = \tilde{t}' - t$. Then, $t' \geq \tilde{t}$, and $(\nu + t) + t' = (\nu + t) + (\tilde{t}' - t) = \nu + \tilde{t}' \in [\![\delta']\!]$. Therefore, $\delta \leq_{\nu + t} \delta'$. ◀

▶ **Example 36.** Fix a configuration $(\vec{q}, \vec{w}, \nu)$ of some system that includes one of the CTA in Figure 4. The edge $(q_0, \mathtt{pr}!b(x < 3), q_2)$ of A_1 is future-enabled iff $\nu(x) < 3$; the latest-enabled edge is $(q_0, \mathtt{ps}!b(x < 5), q_3)$ if $\nu(x) < 5$, otherwise there are no latest-enabled edges. Both outgoing edges of A_2 are latest-enabled if $\nu(x) \leq 2$. In A_3 we can have two non-deferrable edges: $(q_0, \mathtt{rp}?a(x < 2), q_1)$ if $w_{\mathtt{rp}} = aw'_{\mathtt{rp}}$ and $\nu(x) < 2$, and $(q_0, \mathtt{sp}?b(x < 3), q_2)$ if $w_{\mathtt{sp}} = bw'_{\mathtt{sp}}$

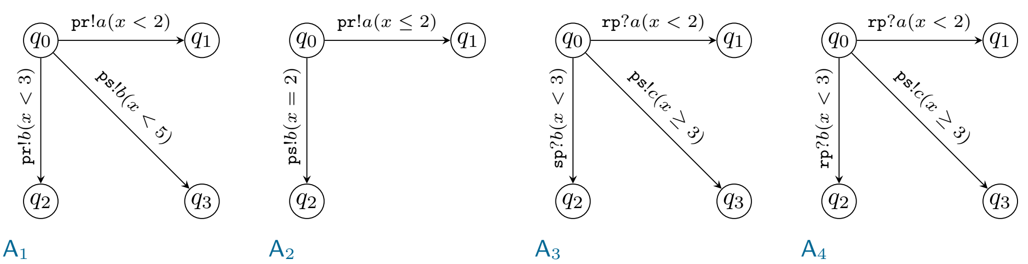


Figure 4 CTA for Example 36.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:19–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and $\nu(x) < 3$. In A_4, if $w_{\mathtt{rp}} = aw'_{\mathtt{rp}}$ and $\nu(x) < 2$, then the edge $(q_0, \mathtt{rp}?a(x < 2), q_1)$ is non-deferrable. Otherwise, if $w_{\mathtt{rp}} = bw'_{\mathtt{rp}}$ and $\nu(x) < 3$, then $(q_0, \mathtt{rp}?b(x < 3), q_2))$ is non-deferrable. In this CTA, only one of the two edges can be non-deferrable, as they both receive from $\mathtt{r}$. If the head of the queue $w_{\mathtt{rp}}$ is neither a nor b, then A_4 does not have non-deferrable edges.

▶ **Lemma 37.** *Let $e = (q, \ell, q') \in E_{\mathtt{p}}$. Then:*

$$e \textit{ future-enabled in } (\vec{q}, \vec{w}, \nu + t) \implies e \textit{ future-enabled in } (\vec{q}, \vec{w}, \nu)$$

Proof. Trivial. ◀

Proof of Lemma 6

We have to prove that, for all configurations $\gamma, \gamma', \gamma''$, and for all $t, t' \in \mathbb{R}_{\geq 0}$, we have that:

$$\gamma \xrightarrow{t} \gamma' \ \wedge \ \gamma \xrightarrow{t} \gamma'' \implies \gamma' = \gamma'' \qquad \text{(Time determinism)}$$
$$\gamma \xrightarrow{t+t'} \gamma' \iff \exists \tilde{\gamma} : \gamma \xrightarrow{t} \tilde{\gamma} \ \wedge \ \tilde{\gamma} \xrightarrow{t'} \gamma' \qquad \text{(Time additivity)}$$

Time determinism follows immediately by Definition 4. Time additivity follows by Lemma 37. ◀

Proof of Theorem 7

For receive persistency, since $\gamma \xrightarrow{t'} \xrightarrow{\mathtt{rp}?}$ then some message expected from $\mathtt{p}$ is already at the head of the queue $\mathtt{rp}$ in configuration γ. Hence, there exists some edge in $E_{\mathtt{p}}$ which is non-deferrable in γ. Since $\gamma \xrightarrow{t} \gamma'$, condition 3(e) of Definition 4 ensures that there exists some non-deferrable edge also in γ'. Since $\leq_\nu$ is total, then there exists a non-deferrable edge e which is enabled after all the other non-deferrable edges. Assume that $\mathsf{act}(e) = \mathtt{sp}?$. Since non-deferrable edges are also future-enabled, there exists t'' such that the guard of e is true in $\nu' + t''$, where ν' is the clock valuation of γ'. We show that the transition $\gamma' \xrightarrow{t''}$ is admitted by our semantics. Condition 3(d) of Definition 4 is satisfied, because if the latest-enabled edge is a sending edge, then the latest time when it can be fired falls after t'' (otherwise, the edge e would be the latest enabled one). Condition 3(e) of Definition 4 holds as well, because the edge e itself remains non-deferrable in $\nu' + t''$. Therefore, by condition 2 of Definition 4, we obtain the thesis $\gamma' \xrightarrow{t''} \xrightarrow{\mathtt{sp}?}$.

For send persistency, since $\gamma \xrightarrow{t'} \xrightarrow{\mathtt{pr}!}$ then there exists a sending edge in $E_{\mathtt{p}}$ which is future-enabled. Since $\leq_\nu$ is total, then there exists a sending edge e_s which is enabled after all the other sending edges. Note that e_s is not necessarily latest-enabled, because the latest-enabled edge could be a receiving one. There are two cases:

1. If there exist no non-deferrable receiving edges, let t'' be such that the guard of e_s is true in $\nu' + t''$, where ν' is the clock valuation of γ' (such t'' always exists, because e_s is future-enabled). Assume that $\mathsf{act}(e) = \mathtt{ps}!$. We show that the transition $\gamma' \xrightarrow{t''} \gamma''$ is admitted by our semantics. Condition 3(d) of Definition 4 holds: indeed, by the choice of e_s and of t'' it follows that if e_s was latest-enabled in γ', then it is latest-enabled also in γ''. Condition 3(e) holds trivially, because in this case we are assuming all receiving edges to be deferrable. Further, by condition (1) of Definition 4, $\gamma'' \xrightarrow{\mathtt{ps}!}$. Therefore we conclude that $\gamma' \xrightarrow{t''} \xrightarrow{\mathtt{ps}!}$.
2. If there exists some non-deferrable receiving edges, let e_r be the latest-enabled among them. Let t_r be the latest delay where the guard of e_r is satisfied from ν', and let t_s be the latest delay where the guard of e_s is satisfied from ν'. There are two further subcases:
   a. if $t_r \geq t_s$, we show that the latest sending action is preserved. Let $t'' = t_s$, and let $\mathsf{act}(e_s) = \mathtt{ps}!$. We show that the transition $\gamma' \xrightarrow{t''} \gamma''$ is admitted by our semantics. Condition 3(d) of Definition 4 holds: indeed, by the choice of e_s and of t'' it follows that if e_s

was latest-enabled in γ', then it is latest-enabled also in γ''. Condition 3(e) holds, because $t_r \geq t_s$, and so the edge e_r is still non-deferrable in γ''. Since the guard of e_s is satisfied in $\nu' + t''$, by condition (1) of Definition 4, we obtain the thesis $\gamma' \xrightarrow{t''} \xrightarrow{\texttt{ps!}}$.

b. otherwise, if $t_r < t_s$, we show that the latest receiving action is preserved. Let $t'' = t_r$, and let $\mathsf{act}(e_r) = \texttt{sp?}$. We show that the transition $\gamma' \xrightarrow{t''} \gamma''$ is admitted by our semantics. Condition 3(d) of Definition 4 holds: indeed, by the choice of e_s and of t'' it follows that e_s is latest-enabled in γ''. Condition 3(e) holds, because t_r is the longest delay such that e_r is still non-deferrable in γ''. Since the guard of e_r is satisfied in $\nu' + t''$, by condition (2) of Definition 4, we obtain the thesis $\gamma' \xrightarrow{t''} \xrightarrow{\texttt{sp?}}$. ◀

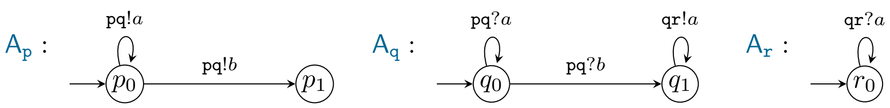
B Proofs and additional material for Section 3

Proof of Lemma 18

We prove the contrapositive. Assume that S does *not* enjoy global progress, i.e. by Definition 17 there exists $\gamma = (\vec{q}, \vec{w}, \nu)$ not final such that $\gamma_0 \rightarrow^* \gamma$, but for no t and $\alpha \in \mathsf{Act}$ it holds that $\gamma \xrightarrow{t} \xrightarrow{\alpha}$. So, by Definition 8, $\rho = \gamma$ is a maximal run of γ. Since γ is not final, there exists a participant p such that q_{p} is not final. Clearly, since the run γ has no transitions, there is no n such that $\mathsf{subj}(act_n(\rho)) = \mathsf{p}$. Therefore, by Definition 17, we conclude that S does not enjoy local progress. ◀

The following example shows a situation where all runs admit a continuation containing send/receive actions of *all* CTA (if not in a final state). Yet, local progress does *not* hold, because there also exist (maximal) runs where a CTA is stuck.

▶ **Example 38.** Consider the following CTA (guards are $true$, and clocks are immaterial):



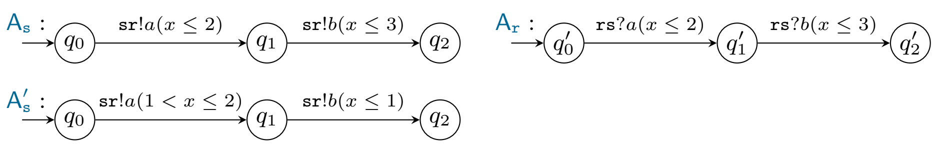
In any reachable configuration of $(\mathsf{A_p}, \mathsf{A_q}, \mathsf{A_r})$ there is a continuation where any CTA can make an action in Act. However, in runs where $\mathsf{A_p}$ always sends a to $\mathsf{A_q}$, the CTA $\mathsf{A_r}$ is stuck. Hence, the system enjoys global progress, but it does not enjoy local progress.

C Proofs and additional material for Section 4

This section contains the proof of the main result Theorem 26. The proof relies on properties of some subclasses of the "naïve" refinement strategy mentioned in the main text (and formally defined in Definition 52), which we will refer to with symbol $\sqsubseteq_{sr}$. In particular, we introduce two properties LESP and NDP (Definitions 46 and 49) on system refinements that guarantees preservation of behaviour and progress of $\sqsubseteq_{sr}$ (Theorem 58). Note that LESP and NDP are not direcly usable in practice due to their undecidability (Theorems 48 and 51). We then show that $\sqsubseteq_1$ is a NDP restriction of $\sqsubseteq_{sr}$ (Lemma 59), and that if a point-wise refinement is LLESP, then the system refinement associated with it is LESP (Theorem 65). From the above it follows that $\sqsubseteq_1^L$ preserves behaviour and progress.

Additional counter-example of preservation in the general case. Note that all CTA are not mixed.

▶ **Example 39.** Let $S = (\mathsf{A_s}, \mathsf{A_r})$, and and let $S' = (\mathsf{A'_s}, \mathsf{A_r})$, where:



29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:21–40:34
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Note that $\mathsf{A}'_{\mathsf{s}} \sqsubseteq_1 \mathsf{A}_{\mathsf{s}}$, and so $S' \sqsubseteq S$. Consider the run $\gamma_0 \xrightarrow{2} \gamma_1 \xrightarrow{\mathtt{sr}!a} \gamma_2 \xrightarrow{\mathtt{sr}?a} \gamma_3 \xrightarrow{3}$ of S', where γ_i are uniquely determined by the labels. The last delay is possible since S' has no FE actions in γ_3 (so, it is stuck). Instead, in S the last delay is *not* possible, since A_{s} has a FE action in γ_3, but no FE actions after a delay of 3 time units. Since S' has a trace not allowed by S, behaviour is *not* preserved. Similarly, S enjoys progress (both global and local), while S' gets stuck in $((q_1, q'_1), (\varepsilon, \varepsilon), \nu)$, with $1 < \nu(x) \leq 2$. So, global and local progress are *not* preserved.

C.1 General properties

We start with some auxuliary definitions and lemmas about CTA and refinements.

▶ **Definition 40.** For all configurations $\gamma = (\vec{q}, \vec{w}, \nu)$ of a given system S, and for all $t \in \mathbb{R}_{\geq 0}$, we define:

$$BSE_\gamma(t) = \left\{ e \;\middle|\; \begin{array}{r} \exists \mathrm{p}, q'_{\mathrm{p}}, \ell : e = (q_{\mathrm{p}}, \ell, q'_{\mathrm{p}}) \text{ latest enabled sending edge in } \gamma \\ \text{and not future enabled in } (\vec{q}, \vec{w}, \nu + t) \end{array} \right\}$$

$$BRE_\gamma(t) = \left\{ e \;\middle|\; \begin{array}{r} \exists \mathrm{p}, q'_{\mathrm{p}}, \ell : (q_{\mathrm{p}}, \ell, q'_{\mathrm{p}}) \text{ non-deferrable edge in } \gamma \text{ and} \\ \forall \ell', q''_{\mathrm{p}} : (q_{\mathrm{p}}, \ell', q''_{\mathrm{p}}) \text{ not non-deferrable in } \nu + t \end{array} \right\}$$

$$BE_\gamma(t) = BSE_\gamma(t) \cup BRE_\gamma(t)$$

We then define the following family of sets (indexed over $n \in \mathbb{N}$):

$$R^n(\gamma) = \{\gamma' \mid \exists t \in \mathbb{R}_{\geq 0}, \alpha \in \mathsf{Act} : \gamma' = (\vec{q}, \vec{w}, \nu + t) \xrightarrow{\alpha} \text{ and } |BE_\gamma(t)| = n\}$$

Intuitively, the set $BE_\gamma(t)$ contains the actions that prevent the timed transition $\gamma \xrightarrow{t}$. It is composed by the sets $BSE_\gamma(t)$ and $BRE_\gamma(t)$, that contains the actions violating, respectively, conditions (d) and (e) of Definition 4 in an attempt to derive $\gamma \xrightarrow{t}$. Finally, the set $R^n(\gamma)$ is composed of those configurations γ' that can perform a discrete transition immediately, and in a derivation of $\gamma \xrightarrow{t} \gamma'$ there are exactly n edges that breaks one among conditions (e), (d), while the other conditions hold. Such an apparently contorted definition will be useful in proofs, as it enables us to perform inductions on the index n of $R^n(\gamma)$.

▶ **Lemma 41.** *Let $\gamma = (\vec{q}, \vec{w}, \nu)$ be such that $\gamma_0 \rightarrow^* \gamma$. For all $t \in \mathbb{R}_{\geq 0}$ and $\alpha \in \mathsf{Act}$:*

$$\gamma \xrightarrow{t} \gamma' \xrightarrow{\alpha} \iff \gamma' \in R^0(\gamma)$$

Proof. Direct consequence of the fact that $BSE_\gamma(t)$ and $BRE_\gamma(t)$ are composed by those edges that, respectively, break conditions (e) and (d) of Definition 4. ◀

▶ **Lemma 42.** *Let S be a system of machines, and let γ be any configuration such that $\gamma_0 \rightarrow^* \gamma$. Then, for all $t, t', f \in \{BSE_\gamma, BRE_\gamma, BE_\gamma\}$:*

$$t \leq t' \implies f(t) \subseteq f(t')$$

Proof. Let S and γ be as in the statement, and suppose $t \leq t'$. We proceed by cases on f.

- $f = BSE_\gamma$. Suppose $e = (q_{\mathrm{p}}, \ell, q'_{\mathrm{p}}) \in BSE_\gamma(t)$. It must be ℓ latest enabled sending in γ and not future enabled in $\nu + t$. It remains to show ℓ not future enabled in $\nu + t'$: an easy inspection of definition 3, using the assumption $t < t'$.
- $f = BRE_\gamma$. Similar to the above.
- $f = BE_\gamma$. Immediate consequence of the above cases. ◀

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:22–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Lemma 43.** *Let S be a system of machines, and let $\gamma = (\vec{q}, \vec{w}, \nu)$ be a configuration such that $\gamma_0 \rightarrow^* \gamma$. Then, for all t and for all $\alpha \notin \mathbb{R}_{\geq 0}$:*

$$(\vec{q}, \vec{w}, \nu + t) \xrightarrow{\alpha} \wedge\, BRE_\gamma(t) = \emptyset \implies \exists t', \ell' : \gamma \xrightarrow{t'} \xrightarrow{\ell'}$$

Proof. Let S and γ be as in the statement. Since, for all t, any γ' in the form $(\vec{q}, \vec{w}, \nu + t)$ and such that $\gamma' \xrightarrow{\alpha}$ is a member of $R^n(\gamma)$, for some n, we proceed by induction on n.
The base case follows by lemma 41.
For the inductive step, let $n > 0$ and let $\gamma' \in R^n(\gamma)$ be such that $BRE_\gamma(t) = \emptyset$. Pick any member e of $BE_\gamma(t)$. Since $BRE_\gamma(t)$ is empty, it must be $e \in BSE_\gamma(t)$, and thus $e = (q_{\mathsf{p}}, \ell, q'_{\mathsf{p}})$ for some p, with ℓ latest enabled (and hence future-enabled) sending in γ, but not future enabled in $\nu + t$. Then, there is $t' < t$ such that $\nu + t' \in [\![\mathsf{guard}(\ell)]\!]$. Therefore, ℓ is future enabled in $\nu + t'$ and hence $e \notin BE_\gamma(t')$. Thanks to this, together with lemma 42, we can conclude $BE_\gamma(t') \subset BE_\gamma(t)$ and therefore $(\vec{q}, \vec{w}, \nu + t') \in R^{n'}(\gamma)$ for some $n' < n$. By the induction hypothesis, it follows that $\gamma \xrightarrow{t''} \xrightarrow{\ell'}$ for some $t'', \alpha' \notin \mathbb{R}_{\geq 0}$. ◀

▶ **Lemma 44.** *Let S_1 and S_2 be systems such that every maximal run ρ of S_1 is a maximal run of S_2. Then, S_2 has local progress $\implies$ S_1 has local progress.*

Proof. Let S_1 and S_2 be as in the statement, and suppose S_2 has strong local progress. We have to show that S_1 has strong local progress as well. So, suppose $\gamma_0 \rightarrow^*_{S_1} \gamma = (\vec{q}, \vec{w}, \nu)$, with run, say, $\rho = \gamma_0 \xrightarrow{t_1}_{S_1} \gamma'_0 \xrightarrow{\alpha_1}_{S_1} \gamma_1 \ldots \gamma$. According to definition 17, we have to show that, for all p such that q_{p} is not final, and for all maximal runs ρ' of S_1 starting from γ, there is n such that $\mathsf{subj}(act_n(\rho')) = \mathsf{p}$. So, suppose q_{p} is not final, and let ρ' be a maximal run of S_1 starting from γ. Clearly, $\rho\rho'$ is a maximal run of S_1 and hence of S_2. Thus, γ is reachable by S_2 and ρ' is a maximal run of S_2 starting from γ. Note that q_{p} is not final with respect to S_2 as well, and hence, since S_2 has strong local progress by assumption, there is n such that $\mathsf{subj}(act_n(\rho')) = \mathsf{p}$. ◀

▶ **Lemma 45.** *Let S be a system that progress. Then, for all $\gamma = (\vec{q}, \vec{w}, \nu)$ not final such that $\gamma_0 \rightarrow^* \gamma$, there is p such that q_{p} has an edge ℓ such that ℓ is latest-enabled sending in γ or ℓ is non-deferrable in γ.*

Proof. Let S and γ be as in the statement. Suppose, by contradiction, that, for every p, every edge of q_{p} is neither latest-enabled sending nor non-deferrable in γ. As a consequence, for all t and for all input labels α, $(\vec{q}, \vec{w}, \nu + t) \not\xrightarrow{\alpha}$. It is still possible that $(\vec{q}, \vec{w}, \nu + t) \xrightarrow{\alpha}$ for some t and some output label α. If this is not the case, system S does not enjoy progress, and we are done by contradiction. Otherwise, there must be some p such that q_{p} has a future-enabled sending edge ℓ in γ. Since q_{p} has no latest-enabled sending edges, q_{p} must have a reading edge ℓ' such that $\mathsf{guard}(\ell') \not\leq_\nu \mathsf{guard}(\ell)$. Then, there is t such that $\nu + t \in [\![\mathsf{guard}(\ell')]\!]$ and for all $t' \geq t$, $\nu + t' \notin [\![\mathsf{guard}(\ell)]\!]$. Then, $(\vec{q}, \vec{w}, \nu + t') \not\xrightarrow{\mathsf{act}(\ell)\mathsf{msg}(\ell)}$ for all $t' \geq t$. Since every machine has finitely many edges, and S is composed by finitely many machines, iterating the above argument we can find a t such that, for every $t' \geq t$, $(\vec{q}, \vec{w}, \nu + t') \not\rightarrow$. Since, for every machine of γ, conditions (d) and (e) of Definition 4 hold trivially for any delay by assumption, $\gamma \xrightarrow{t} (\vec{q}, \vec{w}, \nu + t)$, which is stuck. Hence S does not progress: contradiction. ◀

C.2 LESP and NDP

▶ **Definition 46** (Latest-enabled send preservation)**.** We say that a relation $\mathcal{R}$ between systems is *latest-enabled send preserving* (in short, LESP) iff, whenever $S_1 \,\mathcal{R}\, S_2$, for all $\gamma = (\vec{q}, \vec{w}, \nu)$ such that $\gamma_0 \rightarrow^*_{S_1} \gamma$, and for all p, if q_{p} has a latest-enabled sending edge in γ for S_2, then q_{p} has a latest-enabled sending edge in γ for S_1.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:23–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Example 47.** Recall S and S' from Example 23. The relation $\mathcal{R} = \{(S', S)\}$ is *not* LESP. In S, the sending edge $(p_0, \mathsf{pq}!b(x \leq 3), p_1)$ is latest-enabled in γ_0, but the only sending edge in S', i.e. $(p_0, \mathsf{pq}!b(x \leq 1), p_1)$, is *not* latest-enabled in γ_0. Indeed, from state p_0 of A'_p there is a receiving edge with guard $x \leq 2$, and $(x \leq 2) \not\leq_{\nu_0} (x \leq 1)$. Now, let A''_p be equal to A'_p but for the guard on the send action, which is replaced by $x \leq 2$, and let $S'' = (\mathsf{A}''_\mathsf{p}, \mathsf{A}_\mathsf{q})$. We have that $\mathcal{R}' = \{(S'', S)\}$ is LESP.

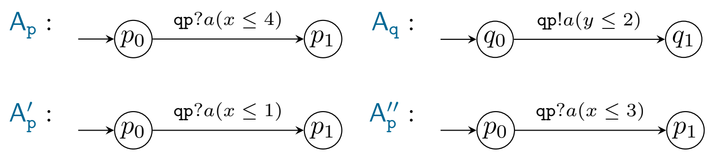
The LESP property is complex to check, in the general case, as it concerns the behaviour of the whole system: in fact, we prove it is undecidable (Theorem 48).

▶ **Theorem 48** (Undecidability of LESP)**.** *Establishing whether restrictions of the system refinement $\sqsubseteq_1$ are LESP is undecidable.*

Sketch. The proof consist in showing that a solution to the problem in the statement would solve (a variation of) the reachability problem for CTA, known to be undecidable. Let S be a system including a machine A owned by p, and let q be a state of A. The reachability problem asks whether a configuration $(\vec{q}, \vec{w}, \nu)$, with $q_\mathsf{p} = q$, is reachable from the initial configuration γ_0. If the answer is positive we say that q is reachable in S. Let A' and A'' be two slighly modified copies of A: they differ in the fact that q has only one exiting edge. This edge is sending, and its guard is false for A' and true for A''. Note that $\mathsf{A}' \sqsubseteq \mathsf{A}''$ for $\sqsubseteq \in \{\sqsubseteq_{sr}, \sqsubseteq_a, \sqsubseteq_1\}$ by construction. Let S' and S'' be as S except for A, that is substituted with A' and A'' respectively. Note that q is reachable in S is equivalent to q is reachable in S' and to q is reachable in S''. This follows by the fact that S' and S'' are equal to S, except for the edges exiting from q. We claim that the relation $\{(S', S'')\}$ is not LESP iff q is reachable in S. This follows by the above and by the fact that the only configurations that breaks the LESP property of $\{(S', S'')\}$ are those in the form $(\vec{q}, \vec{w}, \nu)$, with $q_\mathsf{p} = q$. Since a set is recursive if and only if its complement is, LESP checking is undecidable. ◀

▶ **Definition 49** (Non-deferrable preserving)**.** We say that a relation $\mathcal{R}$ between systems is *non-deferrable preserving* (in short, NDP) iff, whenever $S_1 \ \mathcal{R} \ S_2$, for all $\gamma = (\vec{q}, \vec{w}, \nu)$ such that $\gamma_0 \to^*_{S_1} \gamma$, and for all p, if q_p has a non-deferrable future-enabled edge in γ for S_2, then q_p has a non-deferrable future-enabled edge in γ for S_1.

▶ **Example 50** (Non-deferrable preserving)**.** Consider the following CTA:



Let $S = (\mathsf{A}_\mathsf{p}, \mathsf{A}_\mathsf{q})$, $S' = (\mathsf{A}'_\mathsf{p}, \mathsf{A}_\mathsf{q})$, $S'' = (\mathsf{A}''_\mathsf{p}, \mathsf{A}_\mathsf{q})$, let $\mathcal{R}' = \{(S', S)\}$, and $\mathcal{R}'' = \{(S'', S)\}$. Clearly, both $\mathcal{R}'$ and $\mathcal{R}''$ are LESP, because the configurations of S do not have latest-enabled sending edges. We have that $\mathcal{R}'$ is *not* NDP. To show that, let $\gamma = ((p_0, q_1), (\varepsilon, a), \nu_0 + 2)$, which is reachable both in S and S' since $\gamma_0 \xrightarrow{2} \xrightarrow{\mathsf{qp}!a} \gamma$. The only edge of A_p is non-deferrable in γ, while the edge of A'_p is deferrable, as it is *not* future-enabled in γ. Instead, $\mathcal{R}''$ is NDP, because the edge of A''_p is non-deferrable in γ.

Similarly to LESP, also NDP is undecidable (Theorem 51).

▶ **Theorem 51** (NDP undecidability)**.** *Establishing whether restrictions of the system refinements $\sqsubseteq_{sr}$, $\sqsubseteq_a$, $\sqsubseteq_1$ are NDP is undecidable.*

We now formally define naïve ($\sqsubseteq_{sr}$) and asymmetric ($\sqsubseteq_a$) refinements. $\sqsubseteq_{sr}$ will be useful for proving properties of $\sqsubseteq_1$.

▶ **Definition 52.** Let $\mathsf{A} = (Q, q_0, X, E)$ and $\mathsf{A}' = (Q, q_0, X, E')$ be CTA. The relation $\mathsf{A}' \sqsubseteq_{sr} \mathsf{A}$ holds whenever there exists a structure-preserving isomorphism $f : E \to E'$ such that, for all edges $(q, \ell, q') \in E$, if $f(q, \ell, q') = \ell'$, then $[\![\mathsf{guard}(\ell')]\!] \subseteq [\![\mathsf{guard}(\ell)]\!]$. The relation $\mathsf{A}' \sqsubseteq_a \mathsf{A}$ holds whenever there exists a structure-preserving isomorphism $f : E \to E'$ such that, for all edges $(q, \ell, q') \in E$, if $f(q, \ell, q') = \ell'$ and ℓ is sending, then $[\![\mathsf{guard}(\ell')]\!] \subseteq [\![\mathsf{guard}(\ell)]\!]$; if $f(q, \ell, q') = \ell'$ and ℓ is reading, then $[\![\mathsf{guard}(\ell)]\!] \subseteq [\![\mathsf{guard}(\ell')]\!]$

▶ **Lemma 53.** *Let $\sqsubseteq$ be a LESP & NDP restriction of $\sqsubseteq_{sr}$. Then, for all systems S_1 and S_2 such that $S_1 \sqsubseteq S_2$ and for all γ, γ':*

$$\gamma_0 \rightarrow^*_{S_1} \gamma \xrightarrow{\alpha}_{S_1} \gamma' \implies \gamma \xrightarrow{\alpha}_{S_2} \gamma'$$

Proof. Suppose $S_1 \sqsubseteq S_2$, with isomorphism f, and $\gamma_0 \rightarrow^*_{S_1} \gamma = (\vec{q}, \vec{w}, \nu) \xrightarrow{\alpha}_{S_1} \gamma' = (\vec{q'}, \vec{w'}, \nu')$. We proceed by cases on the rule of Definition 4 used in the derivation $\gamma \xrightarrow{\alpha}_{S_1} \gamma'$.

For rule item 1, it must be $\alpha = \mathtt{pr}!a$ and $q_{\mathtt{p}} \xrightarrow{\ell = \mathtt{pr}!a(\delta,\lambda)}_{S_1} q'_{\mathtt{p}}$, for some $\mathtt{p}, \mathtt{r}, a, \delta, \lambda$ such that $\nu \in [\![\delta]\!]$. By definition 10:

$$q_{\mathtt{p}} \xrightarrow{f(\ell) = \mathtt{pr}!a(\delta',\lambda)}_{S_2} q'_{\mathtt{p}}$$

for some δ' such that $[\![\delta]\!] \subseteq [\![\delta']\!]$. Then, $\nu \in [\![\delta']\!]$ and hence, by rule item 1, $\gamma \xrightarrow{\alpha}_{S_2} \gamma'$.

The case for rule item 2 is similar.

For rule item 3, it must be $\alpha = t$, for some t, and $\gamma' = (\vec{q}, \vec{w}, \nu + t)$. Hence, we have to show:

$$\gamma \xrightarrow{t}_{S_2} \gamma'$$

The only possible rule for the above transition is item 3. Items (a),(b) and (c) clearly hold for S_2 as well. It remains to show it is the case also for items (d) and (e).

For item (d), suppose $e_{S_2} = (q_{\mathtt{p}}, \ell_{S_2}, q'_{\mathtt{p}})$ is a non-deferrable edge of $q_{\mathtt{p}}$ with respect to S_2 in γ, for some $\mathtt{p}$. We have to show there is an edge $e'_{S_2} = (q_{\mathtt{p}}, \ell'_{S_2}, q''_{\mathtt{p}})$ in S_2 that is non-deferrable in γ'. Since $\sqsubseteq$ is NDP, $q_{\mathtt{p}}$ must have an edge ℓ_{S_1} non-deferrable in γ for S_1, and hence, since condition (d) holds for S_1 by the assumption $\gamma \xrightarrow{t}_{S_1} \gamma'$, it holds that $q_{\mathtt{p}} \xrightarrow{\ell'_{S_1}}_{S_1} q''_{\mathtt{p}}$ for some $q''_{\mathtt{p}}, \ell'_{S_1}$ such that ℓ'_{S_1} is non-deferrable in γ'. Now, let $(q_{\mathtt{p}}, \ell'_{S_2}, q''_{\mathtt{p}}) = e'_{S_2}$ be the unique edge of S_2 such that $e'_{S_1} = f(e'_{S_2})$. Since, by definition 10, $[\![\mathsf{guard}(\ell'_{S_1})]\!] \subseteq [\![\mathsf{guard}(\ell'_{S_2})]\!]$, ℓ'_{S_2} is future-enabled in $\nu + t$, and hence item (d) holds for S_2.

For item (e), suppose that, for some p, $q_{\mathtt{p}}$ has a latest enabled (with respect to S_2) sending action ℓ in γ. We have to show ℓ is future enabled in ν'. Since $\sqsubseteq$ is latest-enabled send preserving, $q_{\mathtt{p}}$ has a latest-enabled (with respect to S_1) sending edge $(q_{\mathtt{p}}, \ell', q''_{\mathtt{p}})$ in ν'. By definition 10, it follows that $(q_{\mathtt{p}}, f(\ell'), q''_{\mathtt{p}}) \in E_{\mathtt{p}}$ for S_2, and $\mathsf{guard}(\ell') \subseteq \mathsf{guard}(f(\ell'))$, and hence $(q_{\mathtt{p}}, f(\ell'), q''_{\mathtt{p}})$ is future-enabled in ν'. Now, since ℓ is latest-enabled with respect to S_2, by definition 3 it follows $\mathsf{guard}(f(\ell')) \leq_{nu} \mathsf{guard}(\ell)$. Hence ℓ is future-enabled in ν' as well. ◀

▶ **Lemma 54.** *LESP+NDP restrictions of $\sqsubseteq_{sr}$ preserve behaviour.*

Proof. Let $\sqsubseteq$ be a LESP & NDP restriction of $\sqsubseteq_{sr}$, and let S_1 and S_2 be systems such that $S_1 \sqsubseteq_{sr} S_2$. We have to show there is a timed simulation r between states of $S_1 \uplus S_2$ that relates the initial configuration of S_1 with the initial configuration of S_2, i.e. $((1, s_0), (2, s_0)) \in r$. Define:

$$r \stackrel{\text{def}}{=} \{((1, \gamma), (2, \gamma)) \mid \gamma_0 \rightarrow^*_{S_1} \gamma\}$$

Clearly, $((1, \gamma_0), (2, \gamma_0))$ is a member of r. The fact that r is a timed simulation is an immediate consequence of Lemma 53. ◀

▶ **Lemma 55.** *LESP+NDP restrictions of $\sqsubseteq_{sr}$ preserve global progress.*

Proof. Let S_1, S_2 and $\sqsubseteq$ be as in the statement, and suppose that S_2 has global progress. We have to show that S_1 progress. Suppose $\gamma_0 \rightarrow^*_{S_1} \gamma = (\vec{q}, \vec{w}, \nu)$. If γ is final we are done. If not, by lemma 54, it follows $\gamma_0 \rightarrow^*_{S_2} \gamma$ as well. Since γ is not final also with respect to S_2, $\gamma \xrightarrow{t}_{S_2} \xrightarrow{\alpha}_{S_2}$, for some t, α. By lemma 41, $BRE^{S_2}_{\gamma}(t) = \emptyset$, and, by lemma 42, $BRE^{S_2}_{\gamma}(0) = \emptyset$ as well. We wish to prove there is a t and a $\alpha \notin \mathbb{R}_{\geq 0}$ such that $(\vec{q}, \vec{w}, \nu + t) \xrightarrow{\alpha}_{S_1}$ and $BRE^{S_1}_{\gamma}(t) = \emptyset$. Since S_2 progress, there is a machine q_{p} of S_2 that has a latest-enabled sending or a non-deferrable edge in γ. Then, by the LESP & NDP assumption, q_{p} enjoys the same property with respect to S_1. Now, among the set of non-deferrable edges in γ with respect to S_1, pick a minimal element $e_{S_1} = (q_{\mathrm{p}}, \ell_{S_1}, q'_{\mathrm{p}}$ with respect to the preorder $\leq_\nu$. Such an element exists because $\leq_\nu$ is total and the set is finite and not empty. Then, since ℓ_{S_1} is future-enabled in γ, there is some t such that $(\vec{q}, \vec{w}, \nu + t) \xrightarrow{\alpha}_{S_1}$, where α is the action associated to ℓ_{S_1}. Since e_{S_1} is minimal with respect to $\leq_\nu$, every non-deferrable edge in γ of S_1 is non-deferrable in $(\vec{q}, \vec{w}, \nu + t)$. Therefore, by Lemma 43, $\gamma \xrightarrow{t'}_{S_1} \xrightarrow{\alpha'}_{S_1}$ for some t' and $\alpha' \notin \mathbb{R}_{\geq 0}$. ◀

▶ **Lemma 56.** *Let S_1 and S_2 be systems of machines such that S_2 progress $S_1 \sqsubseteq S_2$ for some LESP & NDP restriction $\sqsubseteq$ of $\sqsubseteq_{sr}$. Then:*

$$\rho \text{ is a maximal run of } S_1 \implies \rho \text{ is a maximal run of } S_2$$

Proof. Let S_1 and S_2 be as in the statement, and suppose ρ is a maximal run of S_1. We first show that ρ is a run of S_2, and then we show it is maximal for S_2. For the first part, it suffice to show that, for all i such that $\gamma_i \xrightarrow{t_i}_{S_1} \gamma'_i \xrightarrow{\alpha_i}_{S_1} \gamma_{i+1}$ appairs in ρ, it holds that $\gamma_i \xrightarrow{t_i}_{S_2} \gamma'_i \xrightarrow{\alpha_i}_{S_2} \gamma_{i+1}$. But this follows by lemma 53. For the second part, i.e. ρ is maximal with respect to S_2, first note that if ρ is infinite the thesis is trivial. So, suppose ρ is finite, with last state γ_n. Since ρ is maximal with respect to S_1, $\neg \gamma_n \xrightarrow{t}_{S_1} \xrightarrow{\alpha}_{S_1}$ for all t, α. But then, since S_2 progress, by lemma 55 S_1 progress as well, and thus γ_n is final for S_1. Therefore, γ_n is final for S_2 too, and ρ is maximal for S_2. ◀

▶ **Lemma 57.** *LESP+NDP restrictions of $\sqsubseteq_{sr}$ preserve local progress.*

Proof. Suppose S_2 has local progress. By lemma 18, S_2 has global progress as well. Then, by lemma 56, maximal runs of S_1 are maximal runs of S_2. Therefore, by lemma 44, S_1 has local progress. ◀

▶ **Theorem 58.** *LESP+NDP restrictions of $\sqsubseteq_{sr}$ preserve behaviour, global and local progress.*

Proof. Composition of lemma 54, lemma 55 and lemma 57. ◀

▶ **Lemma 59.** *$\sqsubseteq_1$ is a NDP restriction of $\sqsubseteq_{sr}$.*

Proof. The fact that $\sqsubseteq_1$ is a restriction of $\sqsubseteq_{sr}$ follows by an easy inspection of definition 10. It remains to show $\sqsubseteq_1$ is NDP. Let A_1 and A_2 be systems of machines such that $A_1 \sqsubseteq_1 A_2$, with isomorphism f, and let $\gamma = (\vec{q}, \vec{w}, \nu)$ be such that $\gamma_0 \rightarrow^*_{S_1} \gamma = (\vec{q}, \vec{w}, \nu)$. Suppose q_{p} has a non-deferrable future-enabled edge $e = (q_{\mathrm{p}}, \ell, q'_{\mathrm{p}})$ in γ for S_2. Then, $f(e) = (q_{\mathrm{p}}, \ell', q'_{\mathrm{p}})$, for some ℓ'. Note that ℓ' is non-deferrable in γ for S_1. It remains to show ℓ' is future-enabled in γ. Since ℓ is future-enabled in γ and, by definition 10, $\downarrow [\![\mathsf{guard}(\ell)]\!] \subseteq \downarrow [\![\mathsf{guard}(\ell)]\!]$, ℓ' is future-enabled in γ. ◀

▶ **Theorem 60.** *LESP restrictions of $\sqsubseteq_1$ preserve behaviour, global and local progress.*

Proof. Composition of Theorem 58 and Lemma 59. ◀

C.3 Local LESP

▶ **Lemma 61.** *For all $S = (\mathsf{A}_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$, for all $\gamma = (\vec{q}, \vec{w}, \nu)$ such that $\gamma_0 \rightarrow^* \gamma$, and for all* $\mathtt{p}$: $\nu \in \mathit{Post}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}}(\mathit{Pre}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}})$.

Proof. Let S be as in the statement. We show the thesis holds for all $\gamma = (\vec{q}, \vec{w}, \nu)$ and for all n such that $\gamma_0 \rightarrow^* \gamma$. By induction on n. For the base case, it must be $\gamma = \gamma_0$, and since all $q_{\mathtt{p}}$ are initial in the respective machines, $\nu_0 \in \mathit{Pre}_{q_{\mathtt{s}}}^{\mathsf{A}_{\mathtt{p}}} \subseteq \mathit{Post}_{q_{\mathtt{s}}}^{\mathsf{A}_{\mathtt{p}}}(\mathit{Pre}_{q}^{\mathsf{A}_{\mathtt{p}}})$ for all $\mathtt{p}$. For the inductive case, let $\gamma = (\vec{q}, \vec{w}, \nu)$ be such that $\gamma_0 \rightarrow^n \gamma' \rightarrow \gamma$ for some $\gamma' = (\vec{q}', \vec{w}', \nu')$. We proceed by cases on the rule of definition 4 used for deriving $\gamma' \rightarrow \gamma$.

- Rule item 1. It must be $\alpha = \mathtt{pr}!a$, $(q'_{\mathtt{p}}, \alpha(\delta, \lambda), q_{\mathtt{p}}) \in E_{\mathtt{p}}$, $\nu = \lambda(\nu')$ and $\nu' \in [\![\delta]\!]$. Since machines do not share cloks, by the induction hypothesis it follows $\nu \in \mathit{Post}_{q_{\mathtt{s}}}^{\mathsf{A}_{\mathtt{s}}}(\mathit{Pre}_{q_{\mathtt{s}}}^{\mathsf{A}_{\mathtt{s}}})$ for all participant $\mathtt{s} \neq \mathtt{p}$. For $\mathtt{p}$, note that $\nu \in \mathit{Pre}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}}$. Therefore $\nu \in \mathit{Post}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}}(\mathit{Pre}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}})$.
- Rule item 2. Similar to the above.
- Rule item 3. It must be $\alpha = t$, $\vec{q} = \vec{q}'$, $\vec{w} = \vec{w}'$, $\nu = \nu' + t$ and conditions (d) and (e) hold. By the induction hypothesis $\nu' \in \mathit{Post}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}}(\mathit{Pre}_{q_{\mathtt{p}}}^{\mathsf{A}_{\mathtt{p}}})$. The thesis follows by condition (d). ◀

We recall some operations on sets of clock valuations from [10], that can be lifted to guards. They are instrumental in the proof of the decidability of LLESP.

▶ **Definition 62.** For all sets of clock valuations K, and for all reset sets λ, we define:

$$\begin{aligned} \uparrow K &\stackrel{\text{def}}{=} \{\nu + t \mid \nu \in K\} \\ \lambda(K) &\stackrel{\text{def}}{=} \{\lambda(\nu) \mid \nu \in K\} \end{aligned}$$

Below we define the sets of clock valuations that satisfies, respectively, the guard of a sending edge and the guard of a receiving edge.

▶ **Definition 63.** For all A and for all q state of A, we define the following sets of clock valuations:

$$\begin{aligned} q^! &\stackrel{\text{def}}{=} \{\nu \mid \exists \mathtt{p}, \mathtt{r}, a, \delta, \lambda : q \xrightarrow{\mathtt{pr}!a(\delta,\lambda)} \wedge \nu \in [\![\delta]\!]\} \\ q^? &\stackrel{\text{def}}{=} \{\nu \mid \exists \mathtt{p}, \mathtt{r}, a, \delta, \lambda : q \xrightarrow{\mathtt{rp}?a(\delta,\lambda)} \wedge \nu \in [\![\delta]\!]\} \end{aligned}$$

We define the set of guards $\mathsf{RGuards}(q)$ in the following way (λ below is lifted to guards):

$$\mathsf{RGuards}(q) \stackrel{\text{def}}{=} \{\lambda(\delta) \mid \exists \ell : q \xrightarrow{\ell} \wedge\, \delta = \mathsf{guard}(\ell) \wedge \lambda = \mathsf{reset}(\ell)\}$$

And we let δ_0 be the guard that equals every clock to zero.

Below, the symbol $\setminus$ denotes set difference.

▶ **Lemma 64.** *For all A, for all q state of A, and for all K, we have that:*

1. $\mathit{Pre}_q^{\mathsf{A}} = \begin{cases} [\![(\bigvee_{\delta \in \mathsf{RGuards}(q)} \delta) \vee \delta_0]\!] & \text{if } q = q_0 \\ [\![\bigvee_{\delta \in \mathsf{RGuards}(q)} \delta]\!] & \text{otherwise} \end{cases}$
2. $\mathit{Les}_q^{\mathsf{A}} = \downarrow (q^! \setminus \downarrow (q^? \setminus \downarrow q^!))$.
3. $\mathit{Post}_q^{\mathsf{A}}(K) = \uparrow (K \setminus \mathit{Les}_q^{\mathsf{A}}) \cup (\uparrow K \cap \mathit{Les}_q^{\mathsf{A}})$.

Proof. Item 1 follows immediately by the semantics of guards in section 2. For item 2, first note that $\downarrow (q^! \setminus \downarrow (q^? \setminus \downarrow q^!)) =$

$$\{\nu \mid \exists t : \nu + t \in q^! \wedge (\forall t' \geq t : \nu + t' \in q^? \implies \exists t'' \geq t' : \nu + t'' \in q^!)\} \tag{2}$$

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:27–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Indeed:

$$\begin{array}{ll}
\downarrow(q^! \setminus \downarrow(q^? \setminus \downarrow q^!)) & = \\
\downarrow(q^! \setminus \downarrow(\{\nu \mid \nu \in q^?\} \setminus \{\nu \mid \exists t : \nu + t \in q^!\})) & = \\
\downarrow(q^! \setminus \downarrow(\{\nu \mid \nu \in q^? \wedge \forall t : \nu + t \notin q^!\})) & = \\
\downarrow(q^! \setminus \{\nu \mid \exists t : \nu + t \in q^? \wedge \forall t' \geq t : \nu + t' \notin q^!\}) & = \\
\downarrow\{\nu \mid \nu \in q^! \wedge (\forall t : \nu + t \in q^? \implies \exists t' \geq t : \nu + t' \in q^!)\} & = \\
\{\nu \mid \exists t : \nu + t \in q^! \wedge (\forall t' \geq t : \nu + t' \in q^? \implies \exists t'' \geq t' : \nu + t'' \in q^!)\} &
\end{array}$$

Now, suppose that $\nu \in Les_q^{\mathsf{A}}$, i.e. q has a latest-enabled sending edge in ν. Then, $q \xrightarrow{\ell}$ for some sending action ℓ with guard δ such that there is $t : \nu + t \in [\![\delta]\!]$ and, for all ℓ' such that $q \xrightarrow{\ell'}$, it holds that $\mathsf{guard}(\ell') \leq_\nu \delta$. Then, since ℓ is sending, it follows $\nu + t \in q^!$ with t as above. By definition of $\leq_\nu$ (definition 3), it follows that ν satisfies: $(\forall t' \geq t : \nu + t' \in q^? \implies \exists t'' \geq t' : \nu + t'' \in q^!)$. Therefore, by eq. (2), $\nu \in \downarrow(q^! \setminus \downarrow(q^? \setminus \downarrow q^!))$. For the converse, suppose $\nu \in \downarrow(q^! \setminus \downarrow(q^? \setminus \downarrow q^!))$. Then, q has some future-enabled sending edge in ν. Let ℓ be the action associated to the latest-enabled (in ν) among sending edges of q. It must exists because $\leq_\nu$ is total, q has finitely many edges, and a latest-enabled sending edge of q exists. Now, suppose $q \xrightarrow{\ell'}$, for some ℓ'. We have to show $\mathsf{guard}(\ell') \leq_\nu \mathsf{guard}(\ell)$. If ℓ' is sending the thesis follows by the assumption that ℓ is latest-enabled among sending edges. If ℓ' is receiving, suppose $\nu + t' \in \mathsf{guard}(\ell')$, for some t'. By eq. (2), there is some t such that $\nu + t \in q^!$. If $t' \geq t$, there is some $t'' \geq t$ such that $t'' \in q^!$, and hence, since ℓ is latest-enabled among sending edges, ℓ is future-enabled in $\nu + t''$ and we are done. If $t' < t$, it follows ℓ future-enabled in t' with an argument similar to above, and we are done.

For item 3:

$$\begin{array}{rl}
Post_q^{\mathsf{A}}(K) = & \{\nu + t \mid \nu \in K \ \wedge \ (\nu \in Les_q^{\mathsf{A}} \implies \nu + t \in Les_q^{\mathsf{A}})\} \\
= & \{\nu + t \mid \nu \in K \ \wedge \ (\nu \notin Les_q^{\mathsf{A}} \vee \nu + t \in Les_q^{\mathsf{A}})\} \\
= & \{\nu + t \mid (\nu \in K \wedge \nu \notin Les_q^{\mathsf{A}}) \vee (\nu \in K \wedge \nu + t \in Les_q^{\mathsf{A}})\} \\
= & \{\nu + t \mid \nu \in K \wedge \nu \notin Les_q^{\mathsf{A}}\} \cup \{\nu + t \mid \nu \in K \wedge \nu + t \in Les_q^{\mathsf{A}}\} \\
= & \uparrow\{\nu \mid \nu \in K \wedge \nu \notin Les_q^{\mathsf{A}}\} \cup (\{\nu + t \mid \nu \in K\} \cap \{\nu \mid \nu \in Les_q^{\mathsf{A}}\}) \\
= & \uparrow(K \setminus Les_q^{\mathsf{A}}) \cup (\uparrow K \cap Les_q^{\mathsf{A}})
\end{array}$$

from which we obtain the thesis. ◀

▶ **Theorem 65** (From LLESP to LESP). *System refinements induced by LLESP point-wise refinements are LESP.*

Proof. Let $\sqsubseteq$ be a system refinement induced by some locally LESP point-wise refinement, and let $S_1 = (\mathsf{A}_\mathtt{p}^1)_{\mathtt{p} \in \mathcal{P}}$ and $S_2 = (\mathsf{A}_\mathtt{p}^2)_{\mathtt{p} \in \mathcal{P}}$ be systems of machines such that $S_1 \sqsubseteq S_2$. Suppose that $\gamma_0 \rightarrow_{S_1}^* \gamma = (\vec{q}, \vec{w}, \nu)$, and that $q_\mathtt{p}$ has a latest-enabled sending edge e_{S_2} in γ for S_2, i.e. $\nu \in Les_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^2}$. We have to show that $q_\mathtt{p}$ has a latest-enabled sending edge e_{S_2} in γ for S_1, i.e. $\nu \in Les_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1}$. By lemma 61, $\nu \in Post_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1}(Pre_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1})$. Hence, since $\sqsubseteq$ is locally LESP by assumption, $\nu \in Post_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1}(Pre_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1}) \cap Les_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1}$ and hence $\nu \in Les_{q_\mathtt{p}}^{\mathsf{A}_\mathtt{p}^1}$. ◀

Proof of Theorem 26

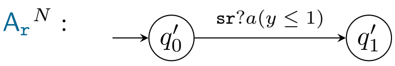
Preservation follows by theorems 60 and 65. Decidability follows by the fact that, by lemma 64, we can effectively construct guards that represents $Post_q^{\mathsf{A}}(Pre_q^{\mathsf{A}}) \cap Les_q^{\mathsf{A}'}$ and $Post_q^{\mathsf{A}}(Pre_q^{\mathsf{A}}) \cap Les_q^{\mathsf{A}}$ respectively, and checking wether $[\![\delta]\!] \subseteq [\![\delta']\!]$ is decidable. ◀

C.4 Counter-examples for alternative refinement strategies

Both ‘naïve’ and ‘asymmetric’ strategies have been formally defined elsewhere in this appendix (Definition 52) and give two refinements denoted with $\sqsubseteq_{sr}$ and $\sqsubseteq_a$, respectively.

▶ **Example 66** (Fact 27: counter-example for naïve strategy ($\sqsubseteq_{sr}$))**.** Consider the following system composed of two CTA:

$\mathsf{A_s}$ and $\mathsf{A_r}$ can be refined, using $\sqsubseteq_{sr}$. We let $\mathsf{A_s}$ unchanged and narrow guards in $\mathsf{A_r}$, yielding $\mathsf{A_r}^N$ below, with $\mathsf{A_r}^N \sqsubseteq_{sr} \mathsf{A_r}$:



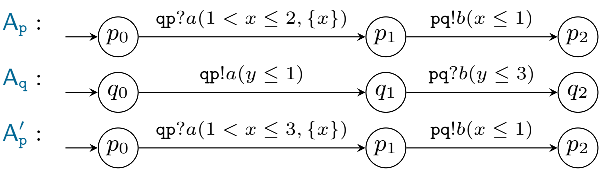
Let $S = (\mathsf{A_s}, \mathsf{A_r})$, and $S' = (\mathsf{A_s}, \mathsf{A_r}^N)$. The relation $\mathcal{R} = \{(S', S)\}$ is LLESP, as the sending edge remains such in S'. Consider the following run (common to S and S'):

$$\gamma_0 \xrightarrow{2} \xrightarrow{\mathtt{sr}!a} \gamma \;=\; ((q_1, q'_0), (a, \varepsilon), \nu_0 + 2)$$

In S' we have $\gamma \xrightarrow{1}$, while in S the only possible timed transition is $\gamma \xrightarrow{0}$. Hence, behaviour is *not* preserved. Since, in S, γ can perform the receive and reach the final configuration, then S enjoys (local/global) progress. Instead, in S' it is too late to receive ($y \leq 1$ is unsatisfiable from $\nu_0 + 2$), hence S' does *not* enjoy (local/global) progress.

Intuitively, the problem with Example 66 is that narrowing the constraints of receiving edges may disable them before the message has been sent.

▶ **Example 67** (Fact 27: counter-example for asymmetric strategy ($\sqsubseteq_a$))**.** Consider the following CTA:



Let $S = (\mathsf{A_p}, \mathsf{A_q})$ and $S' = (\mathsf{A'_p}, \mathsf{A_q})$. We have that $\mathsf{A'_p} \sqsubseteq_a \mathsf{A_p}$. The relation $\{(\mathsf{A'_p}, \mathsf{A_p})\}$ is LLESP. However, behaviour is not preserved, because in S' we have the run $\gamma_0 \xrightarrow{\mathtt{qp}!a} \xrightarrow{3}$, while in S we have $\gamma_0 \xrightarrow{\mathtt{qp}!a} \xrightarrow{t}$ only if $t \leq 2$. Progress is not preserved as well. Indeed, S enjoys global/local progress, while in S' we have:

$$\begin{aligned}
\gamma_0 &\xrightarrow{\mathtt{qp}!a} ((p_0, q_1), (\varepsilon, a), \nu_0)\\
&\xrightarrow{3} ((p_0, q_1), (\varepsilon, a), \nu_0 + 3)\\
&\xrightarrow{\mathtt{qp}?a} ((p_1, q_1), (\varepsilon, \varepsilon), \{x \mapsto 0, y \mapsto 3\})\\
&\xrightarrow{1} ((p_1, q_1), (\varepsilon, \varepsilon), \{x \mapsto 1, y \mapsto 4\})\\
&\xrightarrow{\mathtt{pq}!b} ((p_2, q_1), (b, \varepsilon), \{x \mapsto 1, y \mapsto 4\})
\end{aligned}$$

Since the last configuration in the run is stuck, S' does not enjoy progress.

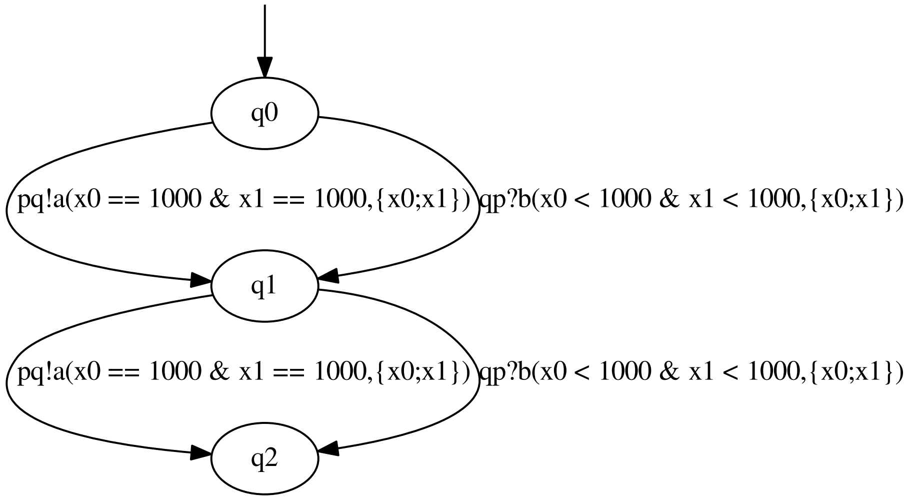


Figure 5 Generated CTA with $m = 3$ and $n = 2$ (image generated by the tool, syntax slightly differs from the one used in the rest of the paper)

C.5 Additional material for Section 4

We evaluate the performance of our tool against automatically generated CTA. We measure termination time parametrically with respect to number of states and number of clocks. We briefly describe the construction procedure. Fix $m, n > 0$ natural numbers denoting, respectively, the number of states and the number of clocks. We construct a CTA $A = (Q, q_0, X, E)$ as follows: $Q = \{q_i \mid i < n\}$ and $X = \{x_i \mid i < m\}$. Let:

$$\begin{aligned} \delta_1 &= x_0 < 1000 \wedge \ldots \wedge x_{n-1} < 1000 \\ \delta_2 &= x_0 = 1000 \wedge \ldots \wedge x_{n-1} = 1000 \\ \lambda &= X \end{aligned}$$

The transition relation E is defined as:

$$\{(q_i, \texttt{rs}?a(\delta_1, \lambda), q_{i+1}) \mid i < m-1\} \cup \{(q_i, \texttt{sr}!b(\delta_2, \lambda), q_{i+1}) \mid i < m-1\}$$

Note that all states but the last are mixed and that all the clocks are used in every guard. This should make the generated CTA fairly more complex to verify than CTA with the same number of edges and clocks, in the average case. We then ask the tool to verify that A is a refinement of herself. Termination times are summarized in table 2. Considering that real-world protocols hardly have thousands of states, the benchmark shows that our refinement based approach is tractable in practice.

	1 clock	5 clocks	10 clocks	20 clocks	40 clocks
100 states (198 edges)	0.09	0.21	0.78	6.66	85.23
1000 states (1998 edges)	3.71	4.98	10.76	70.62	871.40
10000 states (19998 edges)	612.55	638.20	734.04	1385.59	9274.33

Table 2 Benchmarks of the tool. Termination times are expressed in seconds (executed on a Intel i7 laptop with 16GB of RAM).

C.6 Proofs for Section 5

▶ **Example 68.** Consider again the system $(\mathsf{A}_0, \mathsf{A}_0)$ with the CTA in Figure 2. According to the non-urgent semantics, a possible run would be (recalling from Section 2):

$$\gamma_0 \xrightarrow{2} \gamma_1 \xrightarrow{\texttt{sr}!a} \gamma_2 \xrightarrow{1.5} \gamma_3 \xrightarrow{\texttt{rs}?a} \gamma_4$$

Note that $\gamma_2 \xrightarrow{\texttt{rs}?a}$. Hence, the second clause of Definition 28 states that $\gamma_2 \not\xrightarrow{t}$ for all $t > 0$. Then, a maximal run of $(\mathsf{A}_0, \mathsf{A}_0)$ under the urgent semantics would be:

$$\gamma_0 \xrightarrow{2}_u \gamma_1 \xrightarrow{\texttt{sr}!a}_u \gamma_2 \xrightarrow{\texttt{rs}?a}_u \gamma_4' = ((q_1, q_3), (\varepsilon, \varepsilon), \nu_0 + 2)$$

Proof of Theorem 29

A simple inspection of definition 28. ◀

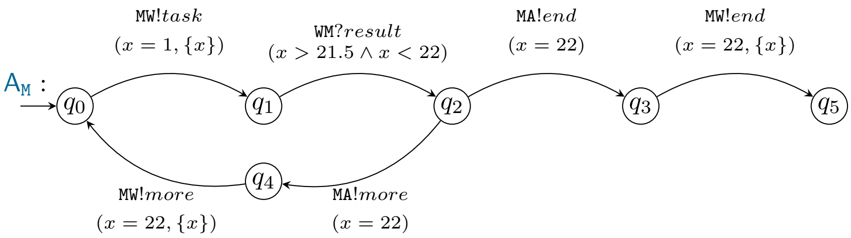
Proof of Theorem 33

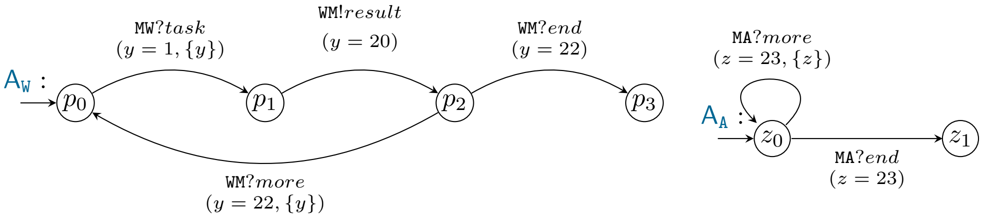
Let S be as in the statement, and suppose S has global progress. We have to show S has global progress also with the urgent semantics. So, suppose $\gamma_0 \to_u^* \gamma = (\vec{q}, \vec{w}, \nu)$. First note that, by theorem 29, $\gamma_0 \to^* \gamma$ as well. If γ is final, we are done. If not, there are t and α such that $\gamma \xrightarrow{t}\xrightarrow{\alpha}$. If $\gamma \xrightarrow{t}_u$ we are done. If not, there must be some p and $t' < t$ such that q_p has non-deferrable edge in $(\vec{q}, \vec{w}, \nu + t')$. Among these edges, pick the minimum element $e = q_\mathsf{p}, \ell, q_\mathsf{p}'$ with respect to $\leq_\nu$. It exists because there are finitely many such edges and $\leq_\nu$ is total. Now, take the least t such that $\nu + t \in [\![\mathsf{guard}(\ell)]\!]$. The existence of such a t follows by the fully left closed assumption. Clearly, $\gamma \xrightarrow{t}_u \xrightarrow{\alpha}$, where α is the label associated to ℓ. ◀

D A larger example of protocol and implementation for Section 6

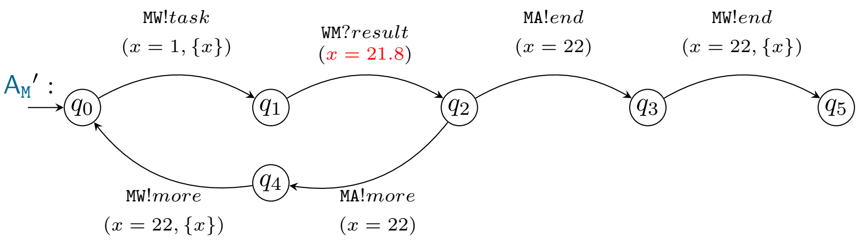
We present the CTA modelling the real-world protocol from [37]. The protocol involves three roles: master, worker, and aggregator, modelled as CTA $\mathsf{A}_\mathtt{M}$, $\mathsf{A}_\mathtt{W}$, and $\mathsf{A}_\mathtt{A}$, respectively. As the protocol in [37] may deadlock, and we assume the initial models to ensure progress (so that refinements *preserve* progress), here we present a patched version (the clock constraint from the initial state of $\mathsf{A}_\mathtt{M}$ has been changed).

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:31–40:34
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

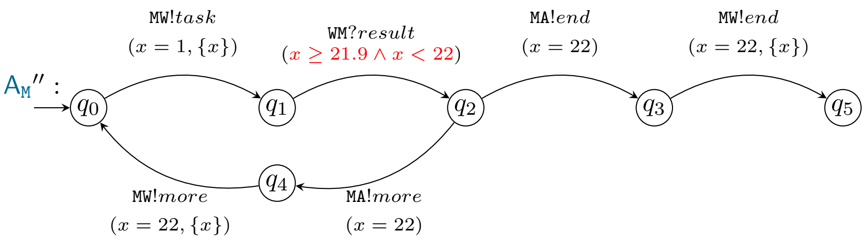




Assuming we wish to use a non-blocking receive primitive, we refine $\mathsf{A_M}$ by choosing a specific point to look in the queue for the result ($x = 21.8$ satisfying $x > 21.5 \wedge x < 22$).



Analysis of $\mathsf{A_M}'$ with our tool reveals that $\mathsf{A_M}' \not\sqsubseteq \mathsf{A_M}$. The problem is in the restriction of the receive action: the automaton is not listening on the queue at the moment of the last deadline of $x > 21.5 \wedge x < 22$. In this case, we could instead restrict the lower bound of open interval $x > 21.5 \wedge x < 22$, as illustrated below.



Analysis of $\mathsf{A_M}''$ with our tool reveals that $\mathsf{A_M}'' \sqsubseteq \mathsf{A_M}$ (and is LLESP).

Automata $\mathsf{A_W}$ and $\mathsf{A_A}$ are not particularly interesting in our scenario as they have punctual constraints only (hence have only themselves as refinements). This is the only case we found with such a widespread use of punctual constraints. A possible implementation of the protocol in Golang is reported below, where each CTA is implemented as a goroutine. The program is parametric to a threshold $\mathtt{timeout_jitt}$ that is added to the timeouts. This makes the program robust against jitter in the communication medium or other delays (e.g. operating system overhead), as soon as they are below the threshold. In all our tests, with a threshold of $1ms$, the program never terminated uncorrectly (i.e. no input timed-out), even with the program in execution for hours.

29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:32–40:34
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
package main

import "fmt"
import "time"

func main() {
abs := time.Now() //starting time of the program, for printing timestamps
timeout_jitt := time.Millisecond //sets tolerance to jitter to 1 ms
MW := make(chan string, 100)
WM := make(chan string, 100)
MA := make(chan string, 100)
Done := make(chan string)//additional channel for goroutine synchronization

//M
go func() {
//initializations
x := time.Now()
var res string
more := true
for more{//loops until no more logs need inspection
  // q0 ---> q1
  time.Sleep(time.Millisecond * 1000 - rel(x))
  MW <- "task"
  x = time.Now()
  fmt.Printf("M: MW!task %v\n",rel(abs))
  // q1 ---> q2
  time.Sleep(time.Millisecond * 21900 - rel(x))
  select {
    case res = <- WM:
      fmt.Printf("M: WM?" + res + " %v\n",rel(abs))
      // q2 ---> q3/q4
      time.Sleep(time.Second *22 - rel(x))
      more = checkMore()
      // checkMore() returns true if more logs need inspection, and false otherwise
      if more {
        // q2 ---> q4
        MA <- "more"
        MW <- "more"
        x = time.Now()//resets x
        fmt.Printf("M: MA!more %v\n",rel(abs))
        fmt.Printf("M: MW!more %v\n",rel(abs))
      } else {
        // q2 ---> q3,q5
        MA <- "end"
        MW <- "end"
        x = time.Now()//resets x
        fmt.Printf("M: MA!end %v\n",rel(abs))
        fmt.Printf("M: MW!end %v\n",rel(abs))
      }
    case <- time.After(time.Second *22 -time.Nanosecond + timeout_jitt - rel(x)):
      handle_timeout("M",abs) // handle_timeout() triggers ad hoc management of receive timeouts
      more = false
  }
}
Done <- "M"//notifies termination
}()

//A
go func (){
//initializations
z := time.Now()
var res string
more := true
for more{ //loops until no more logs need inspection
  time.Sleep(time.Second *23 - rel(z))
  select {
    case res = <- MA:
      // z0 ---> z0
      if res == "more"{
      z = time.Now() //resets y
      fmt.Printf("A: MA?more %v\n",rel(abs))
      }else if res == "end"{
      // z0 ---> z1
      fmt.Printf("A: MA?end %v\n",rel(abs))
      more = false //terminates the loop
      }else{
      //unsespected message handling
      fmt.Printf("A: Unexpected message. Terminating %v\n",rel(abs))
```


29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 40; pp. 40:33–40:34
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
      more = false //terminates the loop
      }
    case <- time.After(time.Second *23 + timeout_jitt - rel(z)):
      //handles timeout
      handle_timeout("A",abs)
      more = false //terminates the loop
  }
}
Done <- "A"//notifies termination
}()

//W
go func (){
//initializations
y := time.Now()
var res string
more := true
for more{ //loops until no more logs need inspection
  // p0 ---> p1
  time.Sleep(time.Second *1 - rel(y))
  if ! (<- MW == "task"){
  //unsespected message handling
  fmt.Printf("W: Unexpected message. Terminating %v\n",rel(abs))
  more = false //terminates the loop
  }
  y = time.Now()//resets y
  fmt.Printf("W: MW?task %v\n",rel(abs))
  // p1 ---> p2
  time.Sleep(time.Second *20 - rel(y))
  WM <- "result"
  fmt.Printf("W: WM!result %v\n",rel(abs))
  time.Sleep(time.Second *22 - rel(y))
  select {
    case res = <- MW:
      if res == "more"{
      // p2 ---> p0
      y = time.Now()//resets y
      fmt.Printf("W: MW?more %v\n",rel(abs))
      } else if res == "end"{
      // p2 ---> p3
      fmt.Printf("W: MW?more %v\n",rel(abs))
      more = false
      } else{
      //unsespected message handling
      fmt.Printf("W: Unexpected message. Terminating %v\n",rel(abs))
      more = false //terminates the loop
      }
    case <- time.After(time.Second *22 + timeout_jitt - rel(y)):
      //handles timeout
      handle_timeout("W",abs)
      more = false //terminates the loop
  }
}
Done <- "W" //notifies termination
}()

//back to main
//waits termination of goroutines
fmt.Printf ("Main: " + <- Done + " terminated %v\n",rel(abs))
fmt.Printf ("Main: " + <- Done + " terminated %v\n",rel(abs))
fmt.Printf ("Main: " + <- Done + " terminated %v\n",rel(abs))
}

//returns clock value
func rel(x time.Time) time.Duration { return time.Now().Sub(x)}

//always returns true for testing purposes
func checkMore() bool {return true}

//just prints debug informations
func handle_timeout(s string, clock time.Time) {fmt.Printf(s + ": Timeout %v\n",rel(clock))}
```