DOI 10.1007/s00165-017-0420-8

Formal Aspects of Computing (2017) 29: 877–910

Formal Aspects of Computing

# Timed runtime monitoring for multiparty conversations

Rumyana Neykova[1], Laura Bocchi[2] and Nobuko Yoshida[1]

[1] Imperial College London, 180 Queens Gate, Kensington SW7 2AZ, UK
[2] University of Kent, Canterbury, UK

**Abstract.** We propose a dynamic verification framework for protocols in real-time distributed systems. The framework is based on Scribble, a tool-chain for design and verification of choreographies based on multiparty session types, which we have developed with our industrial partners. Drawing from recent work on multiparty session types for real-time interactions, we extend Scribble with clocks, resets, and clock predicates in order to constrain the times in which interactions occur. We present a timed API for Python to program distributed implementations of Scribble specifications. A dynamic verification framework ensures the safe execution of applications written with our timed API: we have implemented dedicated runtime monitors that check that each interaction occurs at a correct timing with respect to the corresponding Scribble specification. To demonstrate the practicality of the proposed framework, we express and verify four categories of widely used temporal patterns from use cases in literature. We analyse the performance of our implementation via benchmarking and show negligible overhead.

**Keywords:** Session types, Protocols, Real time, Runtime monitoring, Verification, Scribble

## 1. Introduction

### 1.1. Backgrounds

Recent work [BYY14a] extends Multiparty Session Types (MPSTs) with time, to enable the verification of real-time distributed systems. This extension with time allows specifications (i.e., timed-MPSTs) to express properties on the causalities of interactions, on the carried data types, and on the *times* in which interactions occur. The work in [BYY14a] enables modular static type checking of distributed implementations (i.e., processes in a session $\pi$-calculus) against timed-MPSTs.

As observed in [BCD+13] the direct application of static verification techniques based on MPST presents a few obstacles. First, the existing type systems, including the one in [BYY14a], are targeted at implementations written in calculi with first class communication primitives and communication-oriented control flow. Most mainstream languages would need to be extended in this sense to be suitable for static checking. This is particularly problematic when considering that some have quite liberal sets of primitives (e.g., C) and that implementations may be the

*Correspondence and offprint requests to*: Rumyana Neykova, E-mail: rn710@imperial.ac.uk; rumyana.neykova10@imperial.ac.uk

composition of distributed processes written in different languages. Second, in some scenarios, such as Web programming, it is common to use dynamically typed or untyped languages. Third, static verification guarantees safe communications in the overall distributed system *assuming* that all its parts have been independently and locally type-checked. This requires a degree of trust that cannot always be assumed between the principals that provide the parts of the system. These issues are tackled in [BCD+13, DHH+15] by extending the theory of MPSTs to combine static and dynamic techniques in the verification of different parts of a system. Namely, MPSTs can be used in two ways within the same scenario: for static verification (e.g., of trusted processes written in statically typed languages) and for dynamic enforcement via trusted monitors (e.g., for processes that may not have been or cannot be statically checked); global safety still holds in such mixed networks [BCD+13].

In the present work, we apply the theories in [BYY14a, BCD+13] to implement a toolchain for the design of timed specification and for the dynamic verification of real-time distributed applications. This work is motivated by our collaboration with the Ocean Observatories Initiative (OOI) [OOI], directed at developing a large-scale cyber-infrastructure for ocean observation. We focus on *runtime enforcement* which is most relevant in the OOI infrastructure. The type of protocol used in the governance of the OOI infrastructure (e.g., users remotely accessing instruments via service agents) can be suitably expressed using MPSTs. In particular, OOI protocols can be naturally represented as global types [CDCYP15, HYC08] as they are distributed, typically multiparty, and centered on asynchronous communications via FIFO channels. An *untimed* monitoring framework based on MPSTs [DHH+15] is now integrated into OOI. In this work we extend the framework in [DHH+15] for the verification of *timed* interactions. Time is necessary in many OOI use-cases, for instance to associate timeouts to requests when resources can be used for fixed amounts of time, or to schedule the execution of services at certain time intervals to reduce the busy wait and minimise energy consumption.

### 1.2. A motivating example

To give an idea of the type of timed-protocols used in the OOI infrastructure, we illustrate a simple distributed computation of a word count over a set of logs. The timed global protocol, depicted in Fig. 1 using a message sequence chart (MSC)-like notation, involves three roles: a master `M`, a worker `W` and an aggregator `A`. Each participant has a clock, $\mathtt{x}_\mathtt{M}$, $\mathtt{x}_\mathtt{W}$, and $\mathtt{x}_\mathtt{A}$, respectively, initially set to 0.[1]

1. At the beginning of the session `M` sends `W` a message of type TASK together with a variable of type `log` (i.e., the list of log names to crawl) and a variable of type `string` (i.e., the word to search). The message must be sent by `M` within one second ($\mathtt{x}_\mathtt{M} < 1$) and received by `W` at time $\mathtt{x}_\mathtt{W} = 1$. Both `M` and `W` reset their clocks upon sending/receiving the message.
2. The protocol then enters a loop. At each iteration, `W` replies to `M` in exactly 20 seconds with a message of type RESULT along with a variable of type `log` (i.e., the logs that have been crawled in the given amount of time) and a variable of type `data` (i.e., the result of the word search). This message is received by `M` at any time satisfying $21.5 < \mathtt{x}_\mathtt{M} < 22$.
3. A choice is then made locally to `M` at time 22: depending on whether the results are satisfactory or not, the worker chooses to either terminate the session (message of type END), or to continue the crawling (messages of type MORE). If `W` chooses MORE all clocks are reset. In both cases the results of the last iteration are forwarded to `A`.
4. This timed protocol allows `M` to wake up at regular intervals (e.g., every 20 seconds) to evaluate the results and decide when to continue or terminate the loop. Otherwise `M` can remain idle (e.g., sleep).

### 1.3. A timed monitor framework

Building on the theory in [BYY14a] we have extended the toolchain and framework for verification of choreographic sessions in Python [HNY+13] with time. The framework centers on a specification language called Scribble [SCR, HMB+11, YHNN13, HHN+14, HY16]. As it will be illustrated in Sect. 2, our timed extension of Scribble allows a natural representation of global protocols as the one in Fig. 1. Our toolchain supports the top-down development methodology illustrated in Fig. 2 and explained below.

[1] As customary in MPSTs, protocols start synchronously for all roles, hence all clocks start counting at the same time.

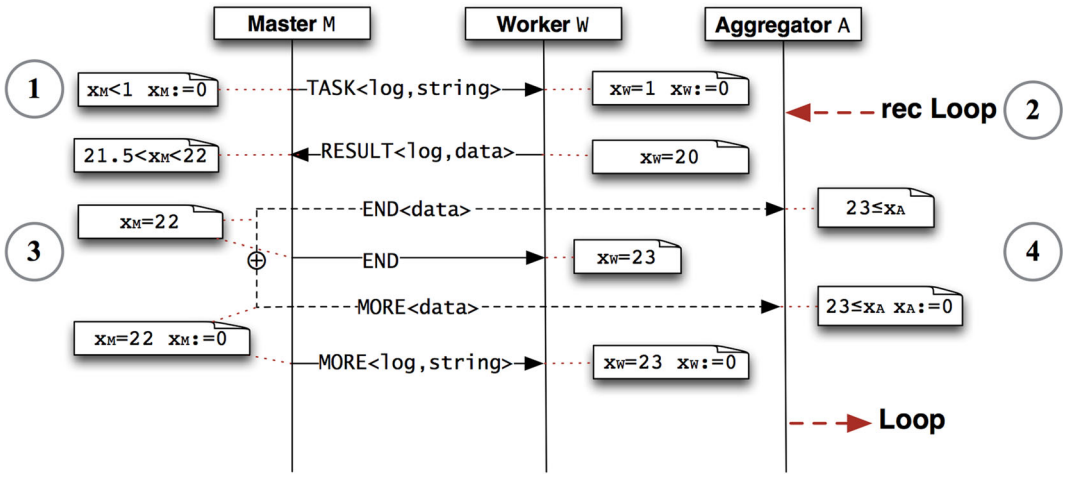


**Fig. 1.** Global protocol for log crawling in scribble

- In step 1, a global communication is *specified* as a Scribble *timed global protocol*. A timed global protocol defines: (a) the causality among interactions in a session involving two or more roles, (b) the datatypes carried by the messages, and (c) the timing constraints of each interaction. We extend Scribble with the notion of time from [KY06a, BYY14a]: each participant owns a clock on which timing constraints can be defined. The clock can be reset many times in a session, and we assume that time flows at the same pace for all clocks and parts of the system.
- In step 2, the Scribble toolchain performs a sanity, or *consistency* check on the timed global protocol produced in step 1. The timed global protocol is checked against two consistency conditions called *feasibility* and *wait-freedom*. These conditions rule out protocols with unsatisfiable constraints which would intrinsically force well-intentioned principals to either stop performing actions or continue by violating the protocol's constraints for the role they implement.
- In step 3, the Scribble toolchain is used to algorithmically *project* the timed global protocol to *timed local protocols*. Each timed local protocol specifies the actions in a session (and their timing) from the perspective of a single role.
- In step 4, principals over a network *implement* one or more, possibly interleaved, timed local protocols. We will call these implementations *timed endpoint programs*. In our prototype implementation, timed local protocols are written in native Python using our in-house developed conversation API. Our Python conversation API is a message passing library that supports the core primitives for communication programming of MPSTs.
- Finally, in step 5, the timed endpoint programs are executed. Each endpoint is associated to a dedicated and trusted monitor. A monitor checks that the interactions of the monitored timed endpoint program conform to the implemented timed local protocols. In case of violation, the monitor either throws a time error (error detection mode), or triggers recovery actions to amend the conversation (error prevention/recovery mode).

Note that there is a substantial difference between step 2 and step 5: whereas step 2 checks that the protocol is *satisfiable*, step 5 checks that the protocol is actually *satisfied* by a specific implementation.

### 1.4. Contributions and outline

The main contribution of the present work is a toolchain for timed interactions, allowing to

- define timed protocols with Scribble—step 1 in Fig. 2;
- automatically verify the consistency of these timed protocols (w.r.t. the consistency principle envisaged in [BYY14a])—step 2 in Fig. 2;
- automatically project timed protocols onto local timed protocols—step 3 in Fig. 2; and
- automatically derive runtime monitors from each local timed protocol to check the incoming/outgoing interactions of the corresponding timed endpoint program—step 4 in Fig. 2.

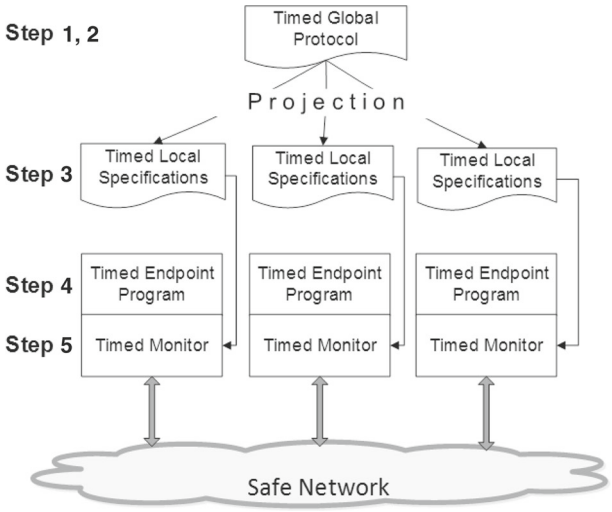


**Fig. 2.** Scribble toolchain framework

This article is based on the theory given in [BYY14a, BYY14b]. Its contribution with respect to [BYY14a, BYY14b] consists of embedding algorithms and theoretical results into the Scribble toolchain, applying them to run-time monitoring and assessing the practicality of the overall approach for timed protocol design and verification.

Concretely, our contributions are: (i) embedding the primitives for *timed protocol specifications* from [BYY14a] into the Scribble toolchain, which include encoding the protocol syntax from [BYY14a] into the Scribble's syntax (given in Appendix A) and giving a concrete implementation into the Scribble toolchain of the algorithms from [BYY14a] for consistency checking and projection; (ii) embedding the calculus for *timed protocol implementations* into a Python API; (iii) exploiting the encoding from timed-MPST into Communicating Timed Automata given in [BYY14a] (and detailed in the corresponding technical report [BYY14b]) to produce run-time monitors for programs implemented using the API from (ii). We present several monitoring *modes*, with different degrees of 'intervention' of the monitor on the ongoing interactions: from just observing interactions to attempts to fix time mismatches. Moreover, we have assessed the practicality of our implementation in two ways. First, to assess the usability of timed Scribble in more general scenarios than those we developed in OOI, we have gathered a wider (albeit not exhaustive) portfolio of properties of timed distributed protocols from literature, and provided a number of *timed patterns* which demonstrate how these properties can be expressed in Scribble. The implementability of the time-properties expressed by timed Scribble in Python is then demonstrated via running examples.

Second, we investigated the concrete effect of time in our monitoring framework via benchmarking. We focused on a property, transparency, which is custom in untimed monitoring frameworks. Transparency (roughly, 'monitors should not effect interactions that are correct') is particularly delicate in timed monitoring frameworks because monitors may introduce time overhead, hence violations of time constraints attached to the specification, in otherwise correct implementations. We introduced a weaker property called *timed transparency* (roughly, 'monitors should have negligible effect interactions that are correct') and provide some experimental observation, which interestingly involve the form of the specific protocols being monitored, to estimate whether the monitor overhead will be negligible or not.

In the following sections we will discuss in detail each of the steps of the methodology illustrated in Fig. 2. In Sect. 2 (steps 1 and 2) we present Scribble timed global and local protocols, which are a practical and more human-readable incarnation of timed global and local types in [BYY14a]. The projection of Scribble timed global protocols onto local protocols has been implemented following [BYY14a, BYY14b]. In Sect. 3 we give a walk-trough of the implementation of an algorithm (from [BYY14b]) for checking consistency over timed Scribble protocols, namely the feasibility and wait-freedom properties. In Sect. 4 we present our timed API (step 3) based on the calculus with delays in [BYY14a] (a simple timed extension of the $\pi$-calculus used to implement timed local types). In Sect. 5 we discuss runtime enforcement of timed properties (step 4). Timed local protocols are automatically encoded into timed automata (using the encoding from timed local types to timed automata presented in [BYY14a, BYY14b]), which are in turn used by our runtime monitors for error detection. Additional mechanisms for error prevention and recovery are implemented and explained. Benchmark results are presented in

Sect. 6. In Sect. 7, to assess the practicality of our approach, and in particular of our timed extension of Scribble, we present a number of temporal patterns drawn from literature together with their Scribble representation. Related work is discussed in Sect. 8. Our prototype implementation is available at [pyt].

## 2. Specifying timed protocols with scribble

### 2.1. Timed global protocols

In Scribble, the interactions between pairs of roles are asynchronous, and can be thought as being broken down into two actions: the sending action, which adds a message to an unbounded FIFO queue, and the receiving action, which collects a message from the queue. We fix a finite set $\mathcal{R} \subset \mathbb{N}$ of roles ranged over by $\mathtt{A}, \mathtt{B}, \ldots$ which exchange messages in a protocol. Each interaction in a protocol transmits a label (we assume the set of labels is finite and is ranged over by $\mathtt{a}, \mathtt{b}, \mathtt{c}$) and a payload of some sort (e.g., integer, boolean, etc.). We assume that each pair of roles, say $\mathtt{A}$ and $\mathtt{B}$, can communicate along two dedicated channel: one for messages from $\mathtt{A}$ to $\mathtt{B}$ and one for messages from $\mathtt{B}$ and $\mathtt{A}$. To model time constraints we fix a set $X$ of real valued clocks (ranged over by $\mathtt{x}, \mathtt{x}', \ldots$) and let each role in a protocol *own* a finite number of clocks in $X$, assuming that the sets of clocks owned by each role in $\mathcal{R}$ is a partition of $X$. We let each sending (resp. receiving) action to be annotated with a time-constraint $\delta$ (or simply *constraint*) and a *reset* $\lambda \subseteq X$. An action can be executed only if the associated constraint is satisfied, and the clocks in $\lambda$ must be reset upon execution of that action. Each role can only reset clocks he/she owns. Note that clocks may have different values at some point in time, since the roles can reset their clocks at different times. However, we assume that time flows at the same pace for all of them (this is a standard assumption e.g. [KY06a]). If the time does not flow at the same pace, no global guarantees can be given.

The syntax of timed Scribble global protocols, or global protocols for short, is given by the grammar below. In the definition of $\delta$, $c$ denotes a constant value in $\mathbb{Q}^{\geq 0}$. Messages are of the form $\mathtt{a}(\mathtt{T})$ with $\mathtt{a}$ being a label and $\mathtt{T}$ being the constant type of the message exchanged (such as `real`, `bool` and `nat`). Protocol names are ranged over by $\mathtt{pro}$, $\mathtt{pro}'$, etc.

$$
\begin{array}{lcll}
\delta & ::= & \mathtt{true} \mid \mathtt{x} < c \mid \mathtt{x} = c \mid \neg\delta \mid \delta_1 \wedge \delta_2 & \textit{constraint} \\
S & ::= & \mathtt{global\ protocol\ pro\ (role\ A_1, \ldots, role\ A_n)}\{G\} & \textit{specification} \\
G & ::= & [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)\ from\ A\ to\ B};\ G & \textit{interaction} \\
 & \mid & \mathtt{choice\ at\ A}\ \{G_1\}\ \mathtt{or}\ \ldots\ \mathtt{or}\ \{G_n\} & \textit{choice} \\
 & \mid & \mathtt{rec\ pro}\ G & \textit{recursion} \\
 & \mid & \mathtt{continue\ pro} & \textit{call} \\
 & \mid & \mathtt{end} & \textit{idle}
\end{array}
$$

A (global) specification $S$ declares a protocol with name $\mathtt{pro}$, involving a list $(\mathtt{A}_1, \ldots \mathtt{A}_n)$ of roles, and prescribing the behaviour in $G$.

- An interaction $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)\ from\ A\ to\ B};\ G$ specifies that a message $\mathtt{a}(\mathtt{T})$ should be sent from role $\mathtt{A}$ to role $\mathtt{B}$ and that the protocol should then continue as prescribed by the continuation $G$.
- Interactions and annotated with constraints and resets, enclosed by square brackets, and explicitly bound to a role. More precisely, the interaction above has two time annotations, $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]$ and $[@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]$, one for the sender $\mathtt{A}$ and one for the receiver $\mathtt{B}$. By $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]$ the message must be sent at a time satisfying the constraint $\delta$. Furthermore, all clocks in $\lambda$ are reset upon sending the message. We assume that only clocks owned by $\mathtt{A}$ occur in $\delta$ and $\lambda$, and tacitly omit $\mathtt{reset}(\lambda)$ when $\lambda = \emptyset$. Similarly, $[@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]$ requires that role $\mathtt{B}$ retrieves the message from the queue at any time satisfying $\delta'$, and that resets all clocks in $\lambda'$. We assume that $\delta'$ and $\lambda$ are defined only on the clocks owned by $\mathtt{B}$.
- A choice $\mathtt{choice\ at\ A}\ \{G_1\}\ \mathtt{or}\ \ldots\ \mathtt{or}\ \{G_n\}$ specifies a branching where role $\mathtt{A}$ chooses to engage in the interactions prescribed by one of the options $G_i$ with $i \in \{1, \ldots, n\}$ and $n > 1$.
- Recursion $\mathtt{rec\ pro}\ G$ defines a scope with protocol name $\mathtt{prot}$ and body $G$. Any call $\mathtt{continue\ pro}$ occurring inside $G$ executes another recursion instance (if $\mathtt{continue\ prot}$ is not in an appropriate scopes than it remains idle).
- $\mathtt{end}$ models the idle process. We will omit trailing occurrences of $\mathtt{end}$.

Figure 3 (left) shows the global protocol of the example 'WordCount' illustrated in Fig. 1 where $\mathtt{M}$ is the master, $\mathtt{A}$ is the aggregator and $\mathtt{W}$ is the worker.

```
global protocol WordCount (role M, role A, role W)
    [@M: xm<1,reset(xm)][@W: xw=1,reset(xw)]
    task(log,string) from M to W;
    rec Loop{
        [@W: xw=20][@M: 21.5<xm<22]
        result(data) from W to M;
        choice at M{
          [@M: xm=22][@A: 23<=xa,reset(xa)]
          more(data) from M to A;
          [@M: xm=22,reset(xm)][@W: xw=23,reset(xw)]
          more(log,string) from M to W;
          continue Loop;
        } or {
          [@M: xm=22][@A: 23<=xa]
          end(data) from M to A;
          [@M: xm=22][@W: xw=23]
          end() from M to W; } }
```

```
local protocol WordCount at M(role A, role W)
    [@M: xm<1,reset(xm)]
    task(log,string) to W;
    rec Loop{
        [@M: 21.5<xm<22]
        result(data) from W;
        choice at M{
          [@M: xm=22]
          more(data) to A;
          [@M: xm=22, reset(xm)]
          more(log,string) to W;
          continue Loop;
        } or {
          [@M: xm=22]
          end(data) to A;
          [@M: xm=22, reset(xm)]
          end() to W; } }
```

**Fig. 3.** Scribble timed global protocol for 'WordCount' (*left*) and Projection onto M (*right*)

```
choice at A
  [@A : ...][@B : ...] a(T) from A to B; end
    or
  [@B : ...][@A : ...] a(T) from B to A; end
```

**Fig. 4.** Global protocol yielding inconsistent local views (not projectable)

The grammar given above provides an overview, albeit slightly simplified, of the constructs supported by Scribble. For readability, the actual code uses different tabs and newlines (e.g., the interactions are shown in two lines, with the annotations above), and constraints may use formulae (e.g, $21.5 < \texttt{xm} < 22$, and $23 <= \texttt{xa}$) which can be easily derived from the ones given in the grammar for $\delta$.

Not all syntactically correct global protocols can be realised as the parallel composition of distributed processes. It may be possible, in fact, to write global protocols yielding to participants' local states that are inconsistent with the global protocol. Figure 4 gives an example of this: A and B may both decide to send a message (each taking a different choice-path) yielding to an inconsistent view of state in the global protocol. We will give later a property, called *projectability* (Definition 2.3), that rules out scenarios as the one we have observed in Fig. 4. Projectability comes with an algorithmic checking procedure that yields local protocols (projections) out of a global protocol.

## 2.2. Formal semantics of Scribble timed global protocols

The formal semantics of global protocols characterises the desired/correct behaviour of the roles in a multiparty protocol. The semantics is based on the Labelled Transition System (LTS) given in [BYY14a] for timed MPSTs and adapted here to the Scribble syntax. The LTS is defined over the following set of transition labels:

$$\ell ::= \texttt{AB!a(T)} \mid \texttt{AB?a(T)} \mid t$$

Label AB!a(T) is for a send action where role A sends to role B a message a(T). Label AB?a(T) is for a receive action where B receives (i.e., collects from the queue associated to the appropriate channel) message a(T) that was previously sent by A. Action $t \in \mathbb{R}^{\geq 0}$ is a time action modelling the elapsing of $t$ time units. We define the subject of an action, modelling the role that has the responsibility of performing that action, as follows:

$$subj(\texttt{AB!a(T)}) = \texttt{A} \qquad subj(\texttt{AB?a(T)}) = \texttt{B} \qquad subj(t) = \emptyset$$

The LTS is defined over states of the form $(\nu, G)$ where $\nu : X \mapsto \mathbb{R}^{\geq 0}$ is a clock assignment mapping clocks to values in $\mathbb{R}^{\geq 0}$. We write:

- $\nu + t$ for the assignment obtained replacing $\nu(x)$ with $\nu(x) + t$ in $\nu$ for all $x \in X$, namely shifting the time forward of $t$ time units.
- $[\lambda \mapsto 0]\nu$ for the clock assignment obtained by setting the value of all $x \in \lambda$ to 0, namely resetting all clocks in $\lambda$.
- $\nu \models \delta$ if the constraint obtained by substituting each clock $x$ occurring in $\delta$ with $\nu(x)$ is satisfied.

As, due to asynchrony, send and receive are two distinct actions, the LTS shall also model the intermediary state where a message has been send but it has not been yet received. To model these intermediary states we introduce the following additional global Scribble interaction:

$$[@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\ G$$

to describe the state in which message a(T) has been sent by A but not yet received by B. We call *runtime* global protocol a protocol obtained by extending the syntax of timed Scribble with these intermediary states.

The transition rules are given in Fig. 5. Rule ⌊SEND⌋ models a sending action: given that the constraint $\delta$ associated with the send action of A is satisfied by the current clock assignment $\nu$ (i.e., $\nu \models \delta$), ⌊SEND⌋ produces a label AB!a(T). The sending action yields a state in which: the clock assignment $\nu'$ is obtained from $\nu$ by resetting all the clocks in $\lambda$, have been reset, and the global protocol is the intermediate state where a(T) has been sent but not received by B. Rule ⌊RECEIVE⌋ models the dual receive action, from the intermediate state to its continuation $G$. Rule ⌊CHOICE⌋ continues the execution of the protocol as the continuation of one of the branches, given that the action is not a time action (all time actions are all handled by one rule, ⌊TIME⌋, which will be discussed later). Due to asynchrony and distribution, in a particular state of a Scribble global protocol it may be possible to trigger more than one action. For instance, the protocol in (1) allows two possible actions: AB!a(T) or CD!a(T).

$$\begin{array}{l}[@\mathtt{A} : \delta_{\mathtt{A}}, \mathtt{reset}(\lambda_{\mathtt{A}})][@\mathtt{B} : \delta_{\mathtt{B}}, \mathtt{reset}(\lambda_{\mathtt{B}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\\ [@\mathtt{C} : \delta_{\mathtt{C}}, \mathtt{reset}(\lambda_{\mathtt{C}})][@\mathtt{D} : \delta_{\mathtt{D}}, \mathtt{reset}(\lambda_{\mathtt{D}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\end{array} \tag{1}$$

This is due to the fact that the two send actions *are not causally related* as they have different subjects (which are independent and distribute roles). We want the semantics of Scribble to allow, in the state with protocol (1), not only the first action that occurs syntactically (e.g., AB!a(T)) but also any action that occurs later, syntactically, but it is not causally related with previous actions in the protocol (e.g., CD!a(T)). Rule ⌊ASYNC1⌋ allows exactly this. In fact, the LTS allows (1) to take one of these two actions: either AB!a(T) by rule ⌊SEND⌋ or CD!a(T) is allowed by ⌊ASYNC1⌋. Rule ⌊ASYNC2⌋ is similar to ⌊ASYNC1⌋ but caters for intermediate states, and is illustrated by the protocol in (2), obtained from (1) via transition AB!a(T) by rule ⌊SEND⌋.

$$\begin{array}{l}[@\mathtt{B} : \delta_{\mathtt{B}}, \mathtt{reset}(\lambda_{\mathtt{B}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\\ [@\mathtt{C} : \delta_{\mathtt{C}}, \mathtt{reset}(\lambda_{\mathtt{C}})][@\mathtt{D} : \delta_{\mathtt{D}}, \mathtt{reset}(\lambda_{\mathtt{D}})]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\end{array} \tag{2}$$

The protocol in (2) can either: execute AB?a(T) by rule ⌊RECEIVE⌋, or CD!a(T) by rule ⌊ASYNC2⌋. Rule ⌊REC⌋ is standard and unfolds recursive protocols.

Before explaining the semantics of time passing, modelled by ⌊TIME⌋, it is necessary to recall a few notions from [BYY14a] (see [BYY14a] for their formal definition): the *ready actions* of a protocol and the *satisfiability of ready actions*. The ready actions of a runtime Scribble global protocol $G$ are the actions that have no causal relationship with other actions that occur earlier, syntactically in $G$, hence could be immediately executed. For example, the ready actions of (1) are AB!a(T) and CD!a(T) and the ready interactions of (2) are AB?a(T) and CD!a(T). We write $\mathtt{rdy}(G)$ for the ready constraints of $G$, namely the set of constraints associated to the ready actions of $G$. For example, if $G$ is the protocol in (1) then $\mathtt{rdy}(G) = \{\{\delta_{\mathtt{A}}\}, \{\delta_{\mathtt{C}}\}\}$, and if $G$ is the protocol in (2) then $\mathtt{rdy}(G) = \{\{\delta_{\mathtt{B}}\}, \{\delta_{\mathtt{C}}\}\}$. The ready constraint set is a *set of sets of constraints*, to cater for the choice construct. For example, in the protocol in (3), $\mathtt{rdy}(G) = \{\{\delta\}, \{\delta_{C1}, \delta_{C2}\}\}$, meaning that there are two ready actions: one (i.e., the receive action of B) that can be executed given that $\delta$ is satisfied, and one (i.e., one of the two possible send action from C) that can be executed given that either $\delta_{C1}$ or $\delta_{C2}$ is satisfied.

$$\begin{array}{l}G = [@\mathtt{B} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a(T)}\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\\ \quad \mathtt{choice\ at}\ \mathtt{C}\ \{[@\mathtt{C} : \delta_{\mathtt{C1}}, \mathtt{reset}(\lambda_{\mathtt{C1}})][@\mathtt{D} : \delta_{\mathtt{D1}}, \mathtt{reset}(\lambda_{\mathtt{D1}})]\ \mathtt{a}_1\mathtt{(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\}\ \mathtt{or}\\ \qquad\qquad\qquad \{[@\mathtt{C} : \delta_{\mathtt{C2}}, \mathtt{reset}(\lambda_{\mathtt{C2}})][@\mathtt{D} : \delta_{\mathtt{D2}}, \mathtt{reset}(\lambda_{\mathtt{D2}})]\ \mathtt{a}_2\mathtt{(T)}\ \mathtt{from}\ \mathtt{C}\ \mathtt{to}\ \mathtt{D}\};\end{array} \tag{3}$$

The aim of the semantics of global protocols is to determine the desirable executions allowed by a protocol. To this aim, we want its semantics of time passing to allow each participant to be able to execute one of the possible next ready action of which this participant is subject/responsible. In other words, we want to prevent the elapsing of time intervals that would invalidate the constraint of the action prescribed by a ready interaction. Consider again the protocol in (1) and assume that $\delta_A = x \leq 20$, $\delta_C = y < 30$ with $\nu(x) = \nu(y) = 0$: we want to prevent the LTS to perform time actions that invalidate any of the two ready actions. Therefore, we will only allow time steps than do not let time elapse of more than $\mathtt{min}(20, 30)$ time units as they would 'not give time' to $A$ and $B$ to perform their ready action. In the case of the protocol in (3) we want that: (1) the only possible action of B (i.e., $\delta$), and (2) at least one of the choice options of C (i.e., $\delta_{C1}$ or $\delta_{C2}$) remain satisfiable at present or some times in the future. In Definition 2.1 we recall satisfiability of ready interactions, written $\nu \models^* \mathtt{rdy}(G)$, from [BYY14a].

$$\frac{\nu \models \qquad ' = [\lambda \mapsto 0]\nu}{(\nu, [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;\ G}) \xrightarrow{\mathtt{AB!a(T)}} (\nu', [@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;\ G})} \lfloor \textsc{send} \rfloor$$

$$\frac{\nu \models \delta' \quad \nu' = [\lambda' \mapsto 0]\nu}{(\nu, [@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;\ G}) \xrightarrow{\mathtt{AB?a(T)}} (\nu', \mathtt{G})} \lfloor \textsc{recv} \rfloor$$

$$\frac{i \in \{1, .., n\} \quad (\nu, G_i) \xrightarrow{\ell} (\nu', G_i') \quad \ell \neq t}{(\nu, \mathtt{choice\ at\ A}\ \{G_1\}\ \mathtt{or}\ \ldots\ \mathtt{or}\ \{G_n\}) \xrightarrow{\ell} (\nu', G_i')} \lfloor \textsc{choice} \rfloor$$

$$\frac{(\nu, G) \xrightarrow{\ell} (\nu', G') \qquad \mathtt{A}, \mathtt{B} \notin subj(\ell) \quad \ell \neq t}{(\nu, [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;\ G}) \xrightarrow{\ell} (\nu', [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;}\ G')} \lfloor \textsc{async1} \rfloor$$

$$\frac{(\nu, G) \xrightarrow{\ell} (\nu', G') \qquad \mathtt{B} \notin subj(\ell) \quad \ell \neq t}{(\nu, [@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;\ G}) \xrightarrow{\ell} (\nu', [@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a(T)}\ \mathtt{from\ A\ to\ B;}\ G')} \lfloor \textsc{async2} \rfloor$$

$$\frac{(\nu, G[\mathtt{rec\ L}\ G/\mathtt{continue\ L}]) \xrightarrow{\ell} (\nu', G')}{(\nu, \mathtt{rec\ L}\ G) \xrightarrow{\ell} (\nu', G')} \lfloor \textsc{rec} \rfloor$$

$$\frac{\nu' = \nu + t \quad \nu' \models^* \mathtt{rdy}(G)}{(\nu, G) \xrightarrow{t} (\nu', G)} \lfloor \textsc{time} \rfloor \tag{4}$$

**Fig. 5.** Labelled transitions for global protocols

**Definition 2.1** (*Satisfiability of ready interactions*) We write $\nu \models^* \mathtt{rdy}(G)$ when the constraints of all ready actions of $G$ are satisfiable under $\nu$ or sometimes in the future. Formally, $\nu \models^* \mathtt{rdy}(G)$ iff $\forall \{\delta_i\}_{i \in I} \in \mathtt{rdy}(G)\ \exists\ t \geq 0,\ j \in I.\ \nu + t \models \delta_j$.

By requiring the satisfiability of some $j \in I$ (i.e., for some branches of a ready action) we allow each action to be executed at any time allowed by its constraints $\delta_j$, not necessarily at the earliest possible time.

Rule $\lfloor$TIME$\rfloor$ allows time to elapse of $t$ time units yielding a new clock assignment $\nu' = \nu + t$ (recall that $\nu + t$ shifts all clocks in $\nu$ of $t$) *as long as* this does not invalidate any of the ready actions of $G$ (i.e., $\nu' \models^* \mathtt{rdy}(G)$). Note that the idle process always allows time to elapse, namely $(\nu, \mathtt{end}) \xrightarrow{t} (\nu + t, \mathtt{end})$ for any $t$ by rule $\lfloor$TIME$\rfloor$, as `end` has no ready actions.

## 2.3. Timed properties of global protocols

The theoretical framework in [BYY14a] sets two consistency conditions on timed global types: *feasibility* and *wait-freedom*. Feasibility (first introduced in [AFK87]) requires that for each partial execution allowed by a specification there is a correct complete one, namely that the protocol will not get stuck due to some unsatisfiable constraint. Wait-freedom requires that if senders respect their time constraints then receivers never have to wait for their messages. These conditions rule out protocols which may intrinsically lead to undesirable scenarios, as shown by the examples in Fig. 6.

The protocol `pro1` in Fig. 6 (a) violates feasibility since it allows `A` to send `msg` at any time satisfying $\mathtt{xa} < 10$, for instance at time 8, for which then `B` has no means to satisfy constraint $\mathtt{xb} < 5$ for the corresponding receive action. The protocol `pro2` in Fig. 6b violates wait-freedom as it allows `A` to send a message when `B` is already waiting for it. Assume `B` to be implemented by a timed endpoint program that receives M1 at time 5, and then engages in a time-consuming activity for 14 seconds before sending M2. The plan of `B` conforms to the timed behaviour prescribed by `pro2` for `B`. If, however, we compose the timed endpoint program described before with an implementation of `A` that sends M1 at time 8, we have that `B` will not find the message in the queue at the expected time 5, will 'get late' with respect to his planned timing, and may end up violating the contract at a later action.

```
global protocol pro1 (role A, role B)
    [@A: xa<10][@B: xb<5]
    msg(string) from A to B;
    ...
```

**(a)**

```
global protocol pro2 (role A, role B)
    [@A: x<10][@B: x<20]
    M1(string) from A to B;
    [@B: x<20][@A: true]
    M2(string) from B to A;
    ...
```

**(b)**

**Fig. 6.** Protocol which violates feasibility (**a**) and wait-freedom (**b**)

In [BYY14a] these conditions yield progress for *statically* validated timed programs, which ensures that the next available action will be executed in the specified time-range. In the case of *dynamically* verified programs against MPSTs (e.g., in [BCD+13] and in our timed framework) progress is difficult to attain. In fact, monitors cannot force timed endpoint programs to send the remaining messages in a protocol when these programs are deliberately refusing or are not able to do so (e.g., their machine is down). Ensuring that conversations are established on feasible and wait-free protocols is, however, a good practice as it prevents violations of time-progress that are induced by the protocol itself.

We implemented a syntactic checker for global protocols of feasibility and wait-freedom, which we will explain in detail in Sect. 3.

## 2.4. Timed local protocols

Scribble timed local protocols, or simply local protocols, describe a session from the perspective of a single participant and they are used to enable local verification of timed processes. The syntax of Scribble local protocols is given below:

$$
\begin{array}{rcl}
T & ::= & [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{to}\ \mathtt{B};\ T \\
& | & [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B};\ T \\
& | & \mathtt{choice\ at\ A}\ \{T_1\}\ \mathtt{or} \ldots \mathtt{or}\ \{T_n\} \\
& | & \mathtt{rec\ pro}\ \{T\} \\
& | & \mathtt{continue\ pro} \\
& | & \mathtt{end}
\end{array}
$$

Local protocol $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{to}\ \mathtt{B};\ T$ models a send action from A to B; the dual local protocol is $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B};\ T$ that models a receive action of A from B. The choice construct is used for both external and internal choices. When $\mathtt{choice\ at\ A}\ \{T_1\}\ \mathtt{or} \ldots \mathtt{or}\ \{T_n\}$ appears in a local protocol for role A it represents an internal choice, when it appears in a local protocol for role $\mathtt{B} \neq \mathtt{A}$ it represents an external choice made by A. For instance, $\mathtt{choice\ at\ B}\ \{\mathtt{m}_1(\mathtt{T}_1)\ \mathtt{from\ B};\ \}\ \mathtt{or}\ \{\mathtt{m}_2(\mathtt{T}_2)\ \mathtt{from\ B};\ \}$ in a local protocol for participant A means that the two receptions, $\{\mathtt{m}_1(\mathtt{T}_1)\ \mathtt{from\ B}\}$ and $\{\mathtt{m}_2(\mathtt{T}_2)\ \mathtt{from\ B}\}$, are conditioned by the decision taken by participant B. The other constructs are similar to the corresponding global protocols.

For convenience we will, sometimes, abuse the notation and refer to the branches in a choice protocol as a union of sets, e.g., writing $\{T\} \cup \{T'\}$ instead of $\{T\}\ \mathtt{or}\ \{T'\}$ and using the notation

$$\mathtt{choice\ at\ A}\ \{[@\mathtt{B} : \delta_i, \mathtt{reset}(\lambda_i)]\ \mathtt{a}_i(\mathtt{T}_i)\ \mathtt{from\ A};\ T_i\}_{i \in \{1,\ldots,n\}}$$

to denote Scribble local protocols of the form

$$\mathtt{choice\ at\ A}\ \{[@\mathtt{B} : \delta_1, \mathtt{reset}(\lambda_1)]\ \mathtt{a}_1(\mathtt{T}_1)\ \mathtt{from\ A};\ T_1\}\ \mathtt{or} \ldots \mathtt{or}\{[@\mathtt{B} : \delta_n, \mathtt{reset}(\lambda_n)]\ \mathtt{a}_n(\mathtt{T}_n)\ \mathtt{from\ A};\ T_n\}$$

with $n > 1$.

We will also omit the choice construct in single-branched choices, e.g., writing $T$ instead of $\mathtt{choice\ at\ A}\ \{T\}$.

Decomposing global protocols into separate but consistent local protocols is called *projection*. The aim of projection is two-folds: checking that a global protocol is distributedly realisable (projectability), and deriving local protocols out of a global protocol. Projection preserves the interaction structures, message exchanges and clock constraints of the global protocol, as it yields local protocols describing which part and responsibilities each target role has in the global conversation. It is a key mechanism to enable distributed enforcement of global properties in our framework. More precisely, projectability is a necessary condition to ensure deadlock freedom and progress of the roles participating in the protocol. Progress of the local processes ensures that every message sent is eventually received, and every process waiting for a message eventually receives one, hence the network cannot end in a deadlock state (a network of processes is deadlocked if all processes are blocked, waiting for messages).

The ability to prove relevant global properties, and in particular progress, by means of local checks requires several restrictions on the shape of the global protocol, notably on the choice construct. These conditions are chosen as to guarantee progress by means of purely local checks on the single peers that participate in the protocol, despite the fact that they will run independently once the session has been established. For timed multiparty session types, projectability of the global protocol is a necessary condition to ensure progress.

Projection requires a role to be involved in all branches of the choice and the local types (obtained via projection of the global type) for each choice branch to be mergeable. Mergeability is given in Definition 2.2.

**Definition 2.2** [*Merge and mergeability* (*adapted from* [YDBH10, BYY14a])] The merge operator $\sqcup$ is defined as a partial commutative operator over two local protocols as follows:

1. $T \sqcup T = T$
2. $\{[@\mathtt{A} : \delta_\mathtt{i}, \mathtt{reset}(\lambda_\mathtt{i})]\, \mathtt{a_i}(\mathtt{T_i})\ \mathtt{from}\ \mathtt{D};\ T_i\}_{i\in I} \sqcup \{[@\mathtt{A} : \delta'_\mathtt{j}, \mathtt{reset}(\lambda'_\mathtt{j})]\mathtt{a'_j}(\mathtt{T'_j})\ \mathtt{from}\ \mathtt{D};\ T'_j\}_{j\in J} =$
   $\{[@\mathtt{A} : \delta_\mathtt{k}, \mathtt{reset}(\lambda_\mathtt{k})]\mathtt{a_k}(\mathtt{T_k})\ \mathtt{from}\ \mathtt{D};\ T_k\}_{k\in I\setminus J} \cup$
   $\{[@\mathtt{A} : \delta'_\mathtt{k}, \mathtt{reset}(\lambda'_\mathtt{k})]\mathtt{a'_k}(\mathtt{T'_k})\ \mathtt{from}\ \mathtt{D};\ T'_k\}_{k\in J\setminus I} \cup$
   $\{[@\mathtt{A} : \delta_\mathtt{k}, \mathtt{reset}(\lambda_\mathtt{k})]\mathtt{a_k}(\mathtt{T_k})\ \mathtt{from}\ \mathtt{D};\ T_k \sqcup T'_k\}_{k\in I\cap J}$
   where $I, J \neq \emptyset$ and $\forall\, k \in I \cap J$, $\mathtt{a_k} = \mathtt{a'_k}$, $\mathtt{T}_k = \mathtt{T}'_k$, $\delta_k = \delta'_k$, and $\lambda_k = \lambda'_k$.

If the merge operation is defined, we say that the local protocols are *mergeable*, otherwise we say that the local protocols are *not mergeable*. The merge operator is defined in two cases: (1) the local types to be merged are identical, and (2) the next actions in the local types to be merged are *receive* actions, which specify the same behaviour in case of common labels (in $I \cap J$). Intuitively, the mergeability condition requires that the two local protocols are identical except for the case in which they are both external choices. In the latter case, any two branches of the two local protocols to merge may be different – in their sorts, time annotations, and continuations – only if they have different labels (the reasons for this will be explained later in this section via examples). The condition $I, J \neq \emptyset$ rules out protocols in which a role does not contribute to all branches of a choice.

We give a definition of projection that is an adaptation of the one from global types to local types in [BYY14a].[2]

**Definition 2.3** (*Projection*) Given $G$ the *projection* of $G$ on a $\mathtt{A} \in \mathcal{P}(G)$, written $G\downarrow_\mathtt{A}$, is defined by induction on $G$ as follows:

$$([@\mathtt{B} : \delta, \mathtt{reset}(\lambda)][@\mathtt{C} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B}\ \mathtt{to}\ \mathtt{C};\ G')\downarrow_\mathtt{A} =$$
$$\begin{cases} [@\mathtt{C} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{B};\ (G'\downarrow_\mathtt{A}) & \text{if } \mathtt{A} = \mathtt{C} \\ [@\mathtt{B} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a}(\mathtt{T})\ \mathtt{to}\ \mathtt{C};\ (G'\downarrow_\mathtt{A}) & \text{if } \mathtt{A} = \mathtt{B} \\ G'\downarrow_\mathtt{A} & \text{if } \mathtt{A} \neq \mathtt{B},\ \mathtt{A} \neq \mathtt{C} \end{cases}$$

$$(\mathtt{choice\ at}\ \mathtt{B}\ \{[@\mathtt{B} : \delta, \mathtt{reset}(\lambda)][@\mathtt{C} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a_i}(\mathtt{T_i})\ \mathtt{from}\ \mathtt{B}\ \mathtt{to}\ \mathtt{C};\ G_i\}_{i\in I})\downarrow_\mathtt{A} =$$
$$\begin{cases} \mathtt{choice\ at}\ \mathtt{B}\ \{[@\mathtt{C} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a_i}(\mathtt{T_i})\ \mathtt{from}\ \mathtt{B};\ (G_i\downarrow_\mathtt{A})\}_{i\in I} & \text{if } \mathtt{A} = \mathtt{C} \\ \mathtt{choice\ at}\ \mathtt{B}\ \{[@\mathtt{B} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a_i}(\mathtt{T_i})\ \mathtt{to}\ \mathtt{C};\ (G_i\downarrow_\mathtt{A})\}_{i\in I} & \text{if } \mathtt{A} = \mathtt{B} \\ \mathtt{choice\ at}\ \mathtt{D}\ (\sqcup\{(G_i\downarrow_\mathtt{A})\}_{i\in I}) & \text{if } \mathtt{A} \neq \mathtt{B}, \mathtt{A} \neq \mathtt{C},\ \forall\, i \in I,\ G_i\downarrow_\mathtt{A} \text{ is of the form } [@\mathtt{D} : \delta_\mathtt{i}, \mathtt{reset}(\lambda_\mathtt{i})]\mathtt{a_i}(\mathtt{T_i})\ \mathtt{from}\ \mathtt{D};\ (G'_i\downarrow_\mathtt{A}) \\ \mathtt{choice\ at}\ \mathtt{A}\ (\sqcup\{(G_i\downarrow_\mathtt{A})\}_{i\in I}) & \text{if } \mathtt{A} \neq \mathtt{B}, \mathtt{A} \neq \mathtt{C}, \forall\, i \in I,\ G_i\downarrow_\mathtt{A} \text{ is } \textit{not} \text{ of the form } [@\mathtt{D} : \delta_\mathtt{i}, \mathtt{reset}(\lambda_\mathtt{i})]\mathtt{a_i}(\mathtt{T_i})\ \mathtt{from}\ \mathtt{D};\ (G'_i\downarrow_\mathtt{A}) \end{cases}$$

$$(\mathtt{rec}\ \mathbf{t}\ \{G'\})\downarrow_\mathtt{A} = \begin{cases} \mathtt{rec}\ \mathbf{t}\ \{(G'\downarrow_\mathtt{A})\} & \text{if } G'\downarrow \mathtt{A} \neq \mathtt{continue}\ \mathbf{t}' \\ \mathtt{end} & \textit{otherwise} \end{cases}$$

$$(\mathtt{continue}\ \mathbf{t})\downarrow_\mathtt{A} = \mathtt{continue}\ \mathbf{t}$$
$$(\mathtt{end})\downarrow_\mathtt{A} = \mathtt{end}$$

[2] For readability, we give the definition for global protocols with guarded choices, that is the first interactions of each choice branch is a message interaction or a choice (not a recursive process). A global protocol can be transformed to a global protocol with a guarded choice after one unfolding of all recursions.

If no side condition applies then $G$ is *not projectable* on A. We say that $G$ is *projectable* if it is projectable for all the participants in $G$. The case for choice uses the merge operator $\sqcup$ and the notion of *mergeability* (Definition 2.2). Mergeability (i.e., the third and fourth cases can be applied only if the merge of $G_i \downarrow_{\mathtt{A}}$ for all $i \in I$ is defined) ensures that either: (1) the locally projected behaviour is independent of the chosen branch (i.e $G_i = G_j$, for all $i, j \in I$), or (2) the chosen branch is identifiable by the receiving participant A via a unique label.

Definition 2.3 is standard except that, as in [BYY14a], each $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]$ in a global protocol interaction is projected onto the sender (resp. receiver) by keeping only the time annotations (constraints and resets) associated to the send action $[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]$ (resp. the receive action $[@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]$). For example, the projection onto M of interaction

$$[@\mathtt{M} : \mathtt{xm} < 1, \mathtt{reset(xm)}][@\mathtt{W} : \mathtt{xw} = 1, \mathtt{reset(xw)}]\ \mathtt{task(log, string)}\ \mathtt{from}\ \mathtt{M}\ \mathtt{to}\ \mathtt{W};\ G$$

yields

$$[@\mathtt{M} : \mathtt{xm} < 1, \mathtt{reset(xm)}]\ \mathtt{task(log, string)}\ \mathtt{to}\ \mathtt{W};\ (G \downarrow_{\mathtt{M}})$$

Another example of projection is given in Fig. 3: the figure presents, on the right-hand side, the local protocol resulting from projecting on M the global protocol 'WordCount' on the left-hand side of the same figure.

Below, we will provide an intuition of projectability, the merge operator, and mergeability, via a set of examples illustrated in Fig. 7. For readability, in this example we will omit time annotations (recall that we also omit trailing occurrences of End). First, the global protocol in Fig. 7a is not projectable on any of its participants, because it is not of the form required by Definition 2.3 for protocols with choice. In fact, the second rule of Definition 2.3 requires global types to be of the form

$$\mathtt{choice\ at\ A}\ \{[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{C} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a_i(T_i)}\ \mathtt{from\ A\ to\ C};\ G_i\}_{i \in I}$$

where each branch consists of an interaction from the same role A to *the same role* C. Since the global protocol in Fig. 7 does not match any of the rules of Definition 2.3, it is not projectable on any of its participants. This restriction on the form of the choice is customary in the literature of MPST. Note that also the (problematic) protocol we have seen in Fig. 4 is not projectable for the same reason: none of the cases of Definition 2.3 matches its syntax as branches in a choice must have the same senders and receivers. The global protocol in Fig. 7b can be projected on A, B, and D, but not on C (hence it is *not projectable*). First, it is easy to observe that the projections of the protocol on A and B are, respectively:

$$\mathtt{choice\ at\ A}\ \{\mathtt{l1()\ to\ B};\ \}\ \mathtt{or}\ \{\mathtt{l3()\ to\ B};\ \} \quad \text{(projection on A)}$$
$$\mathtt{choice\ at\ A}\ \{\mathtt{l1()\ from\ A};\ \}\ \mathtt{or}\ \{\mathtt{l3()\ from\ A};\ \} \quad \text{(projection on B)}.$$

The projection on D is also easy to verify using the merge operator:

$$\mathtt{choice\ at\ C}\ (\{\mathtt{l2()\ from\ C};\ \} \sqcup \{\mathtt{l4()\ from\ C};\ \})$$
$$= \mathtt{choice\ at\ C}\ \{\mathtt{l2()\ from\ C};\ \}\ \mathtt{or}\ \{\mathtt{l4()\ from\ C};\ \}\ \text{(projection on D)}.$$

The problem of the global protocol in Fig. 7b lays in the role of C. Intuitively, C does not know which label has been communicated by A in the first interaction, hence cannot guarantee the causalities between l1 and l2, and between l3 and l4 (i.e., C may send l2 in executions where A had chosen l3). For this reason, the global protocol in Fig. 7b is not projectable on C (hence it is *not projectable*). In the following, we will see how this is reflected in the definition of projection and apply Definition 2.3 to project the global protocol in Fig. 7b on C. First, as C does not participate in the first interaction, we apply the fourth case of choice case in Definition 2.3 which, in turn, uses the merge operator. The merge operator is applied on the projections of the continuations of each branch on C that are:

$$(\mathtt{l2()\ from\ C\ to\ D};\ ) \downarrow_{\mathtt{C}} = \mathtt{l2()\ to\ D};$$
$$(\mathtt{l4()\ from\ C\ to\ D};\ ) \downarrow_{\mathtt{C}} = \mathtt{l4()\ to\ D};$$

First, observe that the two projections result in two different local types (i.e., that send two different labels l2 and l4, respectively), hence case (1) of Definition 2.2 (i.e., $T \sqcup T = T$) cannot be applied to merge them. Second, observe that the two projections result in two *sending* local types (i.e., participant C, on which we are projecting, is sending a message to D), hence case (2) of Definition 2.2, requiring local types to perform *receiving* actions, cannot be applied. As the two projection are not mergeable, then the global protocol in Fig. 7b is not projectable on C. On the contrary, the example in Fig. 7c is projectable on C (and on all other roles) as C is sending the same label l2 in both branches.

```
choice at A {
  l1() from A to C;
  l2() from C to B;
} or {
  l3() from A to D;
  l4() from D to B;
}
```
**(a)**

```
choice at A {
  l1() from A to B;
  l2() from C to D;
} or {
  l3() from A to B;
  l4() from C to D;
}
```
**(b)**

```
choice at A {
  l1() from A to B;
  l2() from C to D;
} or {
  l3() from A to B;
  l2() from C to D;
}
```
**(c)**

```
choice at A {
  l1() from A to B;
  l2() from B to C;
} or {
  l3() from A to B;
  l4() from B to C;
}
```
**(d)**

**Fig. 7.** Examples of non-projectable (**a**, **b**) and projectable (**c**, **d**) global protocols

In the case of the global protocol in Fig. 7c the projection on C is

$$\texttt{choice at C}\ \{(\texttt{l2() to D;}\ ) \sqcup (\texttt{l2() to D;}\ )\} = \texttt{choice at C}\ \{\texttt{l2() to D;}\ \} = \texttt{l2() to D;}$$

since: (a) $(\texttt{l2() to D;}\ ) \sqcup (\texttt{l2() to D;}\ ) = \texttt{l2() to D;}$ by idempotence, (b) we can apply the fourth case for the choice construct (yielding $\texttt{choice at C}\ \{\texttt{l2() to D;}\ \}$), and (c) we omit choice at as the choice consists of only one branch (yielding l2() to D;).

The global protocol in Fig. 7d is also projectable. It is clearly projectable on A and B, as they are directly involved in the first interaction/choice. For instance, consider the projection of this protocol on B: first, we apply the case for choice of Definition 2.3 (where the participant on which we are projecting is receiving the choice), and then we apply the rule for projecting a message interaction on the continuation. The two steps of the projection of the global protocol in Fig. 7d on B are shown below.

$$\begin{aligned}&(\texttt{choice at A\{l1() from A; l2() from B to C;\}\ or\ \{l3() from A; l4() from B to C;\}})\downarrow_{\texttt{B}}\\ &= \texttt{choice at A\{l1() from A; (l2() from B to C;}\ )\downarrow_{\texttt{B}}\texttt{\}\ or\ \{l3() from A; (l4() from B to C;}\ )\downarrow_{\texttt{B}}\texttt{\}}\\ &= \texttt{choice at A\{l1() from A; l2() to C;\}\ or\ \{l3() from A; l4() to C;\}}\end{aligned}$$

The protocol in Fig. 7d is also projectable on C as the projections of each branch on C are mergeable. In this case, the projections of all branches are receive protocols from the same participant B, hence we can apply the third case for choice of Definition 2.3. The projections on C of the two branches of the protocol are, respectively, l2() from B; and l4() from B; . These two protocols can be merged by using case (2) of Definition 2.2:

$$(\texttt{l2() from B;}\ ) \sqcup (\texttt{l4() from B;}\ ) = \{\texttt{l2() from B;}\ \}\ \texttt{or}\ \{\texttt{l4() from B;}\ \}$$

Therefore, by the third case of choice case in Definition 2.3, the projection of the global protocol in Fig. 7d on C is

$$\texttt{choice at B}\{\texttt{l2() from B;}\ \}\ \texttt{or}\ \{\texttt{l4() from B;}\ \}$$

**Formal semantics of local protocols** The LTS for local protocols is defined by the rules in Fig. 8, which use the same labels of the global semantics in Fig. 5. The rules ⌊SEND⌋, ⌊RECV⌋, ⌊CHOICE⌋, ⌊REC⌋ are similar to the respective rules for global protocols. We do not need rules for modelling asynchrony as each participant is assumed to be single threaded. For the time passing rule ⌊TIME⌋ the constraints of the ready action of $T$ must be still satisfiable after $t$ in $\nu$ except $T$ has only one ready action.

**Formal semantics of configurations** The LTS in Fig. 8 describes the behaviour of each single role in isolation. In the rest of this section we give the semantics of systems resulting from the composition of Scribble local protocols and the communication channels. Given a set of roles $\{1, \ldots, n\}$ we define configurations $(T_1, \ldots, T_n, \vec{w})$ where $\vec{w} ::= \{w_{ij}\}_{i \neq j \in \{1,\ldots,n\}}$ are unidirectional, possibly empty (denoted by $\epsilon$), unbounded FIFO queues with elements of the form $\texttt{a(T)}$. The LTS of $(T_1, \ldots, T_n, \vec{w})$ is defined as follows, with $\nu$ being the overriding union (i.e., $\oplus_{i \in \{1,\ldots,n\}} \nu_i$) of the clock assignments $\nu_i$ of the roles: $(\nu, (T_1, \ldots, T_n, \vec{w})) \xrightarrow{\ell} (\nu', (T_1', \ldots, T_n', \vec{w}'))$ iff:

(1) $\ell = \texttt{AB!a(T)} \Rightarrow (\nu_{\texttt{A}}, T_{\texttt{B}}) \xrightarrow{\ell} (\nu_{\texttt{A}}', T_{\texttt{B}}') \wedge w_{\texttt{AB}}' = w_{\texttt{AB}} \cdot \texttt{a(T)} \wedge (ij \neq \texttt{AB} \Rightarrow w_{ij} = w_{ij}' \wedge T_i = T_i')$

(2) $\ell = \texttt{AB?}l\texttt{a(T)} \Rightarrow (\nu_{\texttt{B}}, T_{\texttt{B}}) \xrightarrow{\ell} (\nu_{\texttt{B}}', T_{\texttt{B}}') \wedge \texttt{a(T)} \cdot w_{\texttt{AB}}' = w_{\texttt{AB}} \wedge (ij \neq \texttt{AB} \Rightarrow w_{ij} = w_{ij}' \wedge T_j = T_j')$

(3) $\ell = t \Rightarrow \forall\, \texttt{A} \neq \texttt{B}.\ (\nu_{\texttt{A}}, T_{\texttt{A}}) \xrightarrow{t} (\nu_{\texttt{A}} + t, T_{\texttt{A}}') \wedge w_{\texttt{AB}} = w_{\texttt{AB}}' \wedge$
$(\ T_{\texttt{A}} = \texttt{choice at A}\ \{[@\texttt{B} : \delta_i, \texttt{reset}(\lambda_i)]\ \texttt{a}_i\texttt{(T}_i\texttt{)}\ \texttt{from A;}\ T_i\}_{i \in \{1,\ldots,m\}}$
$\wedge\ w_{\texttt{AB}} = w_{\texttt{AB}}'' \cdot \texttt{a}_s\texttt{(T}_s\texttt{)}$ for some $w_{\texttt{AB}}''$ and $s \in \{1, \ldots, m\}$ )
$\Rightarrow\ \nu + t \models^* \delta_s$

with $\texttt{A}, \texttt{B}, i, j \in \{1, \ldots, n\}$.

$$\frac{\nu \models \quad ' = [\lambda \mapsto 0]\nu}{(\nu, [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a(T)\ to\ B};\ T) \xrightarrow{\mathtt{AB!a(T)}} (\nu', T)} \lfloor \text{SEND} \rfloor \qquad \frac{\nu \models \quad ' = [\lambda \mapsto 0]\nu}{(\nu, [@A : \delta, \mathtt{reset}(\lambda)]\ \mathtt{a(T)\ from\ B};\ T) \xrightarrow{\mathtt{AB?a(T)}} (\nu', T)} \lfloor \text{RECV} \rfloor$$

$$\frac{i \in \{1, .., n\} \quad (\nu, T_i) \xrightarrow{\ell} (\nu', T_i') \quad l \neq t}{(\nu, \mathtt{choice\ at\ A}\ \{T_1\}\ \mathtt{or}\ \ldots\ \{T_n\}) \xrightarrow{\ell} (\nu', T_i')} \lfloor \text{CHOICE} \rfloor \qquad \frac{(\nu, T[\mathtt{rec\ L}\ T/\mathtt{continue\ L}]) \xrightarrow{l} (\nu', T')}{(\nu, \mathtt{rec\ L}\ T) \xrightarrow{l} (\nu', T)} \lfloor \text{REC} \rfloor$$

$$\frac{\nu' = \nu + t \quad \nu' \models^* \mathtt{rdy}(T)}{(\nu, T) \xrightarrow{t} (\nu', T)} \lfloor \text{TIME} \rfloor$$

**Fig. 8.** Labelled transitions for local protocols

In (1) the configuration makes a send action given that one of the participants can perform that send action. (1) has the effect of adding the message sent to the corresponding queue. In (2) the configuration makes a receive action given that one of its participant can perform such an action and that the message being received is currently stored in the corresponding queue. (2) has the effect of removing the message received from the queue. In (3) the configuration can make a time action $t$ given that all participants can let time elapse for $t$ time units (i.e., according to their ready actions). The additional condition in (3) is needed to cater for scenarios where the protocol being executed is a choice and the message has been sent but not yet received i.e., it is stored in a queue. In this case, the sender has already decided the choice to be taken and it is necessary that time actions preserve the receiver's ability of receiving that specific message (i.e., the constraints of the choice selected by the sender must to be made unsatisfiable by time actions). The condition in (3) is needed to guarantee this, as the receiver (and the definition of receiver's ready actions) do not take into account the state of the queues. Consider for example the protocol below:

```
choice at B {
  [@A : x < 10][@B : x < 10] a1(T1) from B to A
  [@A : x < 20][@B : x < 20] a2(T2) from B to A}
```

If the message $\mathtt{a}_1(\mathtt{T}_1)$ from B is already in the queue, namely the first branch has already been chosen. By default, the $\lfloor$TIME$\rfloor$ rule dictates that the time can elapse as far as there are satisfiable ready actions (e.g., either receiving $\mathtt{a}_1(\mathtt{T}_1)$ or $\mathtt{a}_2(\mathtt{T}_2)$). However, if we allow time to elapse of $t = 19$ *but* the sender has already chosen the first branch, the receiver will not be able to act as prescribed by the protocol (i.e., will not be able to receive the sent message $\mathtt{a}_1(\mathtt{T}_1)$ at the time prescribed by the corresponding constraint). In fact, we must not allow $\lfloor$TIME$\rfloor$ to let elapse for more than 10 unit, otherwise the constraint of the first branch will become unsatisfiable. The constraints in (3) ensures that if there is a message in the queue, the time constraints associated with this messages should be satisfiable after $\nu + t$.

## 2.5. Correspondence of global and local protocols

We write $TR(G)$ for the set of visible traces obtained by reducing $G$ under the initial assignment $\nu_0$. Similarly for $TR(T_1, \ldots, T_n, \vec{\epsilon})$ where $\vec{\epsilon}$ is the vector of empty queues. We denote trace equivalence by $\approx$. Theorem 2.4 gives the correspondence between the traces produced by a global protocol $G$ and those produced by the configuration that consists of the composition of the projections of $G$ onto $\mathcal{P}(G)$.

**Theorem 2.4** (*Soundness and completeness of projection*) Let $G$ be a projectable Scribble timed global protocol and

$$\{T_1, \ldots, T_n\} = \{G \downarrow_{\mathtt{A}}\}_{\mathtt{A} \in \mathcal{P}(G)}$$

be the set of its projections, then $G \approx (T_1, \ldots, T_n, \vec{\epsilon})$.

Theorem 2.4 directly follows by: (i) the correspondence between (Scribble) global protocols and (timed-MPSTs) global types given in Appendix A; (ii) trace equivalence between global types and configuration of projected global types (Theorem 3.3 in [BYY14a]); (iii) the correspondence between configurations of (timed-MPSTs) local types and configurations of (Scribble) local protocols given in Appendix A. Correspondence is important as it ensures that the composition of processes, each implementing some local protocol, will behave as prescribed by the original global specification.

## 3. Checking feasibility and wait-freedom

**Algorithm 1** Algorithm for checking feasibility and wait-freedom in global protocols

```
Require: G = build_time_dependency_graph(AST)                    ▷ Step 1
1: for node in G do
2:     for (constraints, resets) in dfs(root, node) do           ▷ Step 2
3:         constraints, resets = convert_absolute(constraints, resets)   ▷ Step 3
4:         formula = build_z3_formula(constraints, resets)       ▷ Step 4
5:         result = formula.is_satisfiable()
6:         if not result then
7:             return False
8: return True
```

In this section we present a syntactic checker for two time properties on Scribble global protocols: feasibility and wait-freedom. These properties have been first introduced in [BYY14a] to guarantee time-progress for statically validated programs. Time-progress ensures that a validated program is guaranteed to proceed until the completion of all activities of the protocols it implements. A protocol is *feasible* if every partial execution can be extended to a terminated session (i.e., the execution never gets stuck because of an unsatisfiable clock constraint). A protocol is *wait-free* when, in all its distributed implementations, a receiver checking the queue never has to wait for the message. We have discussed in Sect. 2.3 that in our framework, which differently from [BYY14a] is based on dynamic verification, these properties do not guarantee time-progress. Feasibility and wait-freedom are nevertheless important to rule out protocols that may intrinsically lead to violations. Feasibility provides a sanity check on whether the constraints specified by a protocol are satisfiable by some implementation. Wait-freedom rules out timed global protocols whose distributed implementation, built *modularly* and in conformance with each projected timed local protocol, may actually 'get late' with respect to the timing prescribed by the original timed global protocol. In addition, wait-freedom rules out busy waiting, which is critical in areas such as sensor networks, where one of the main sources of energy inefficiency is listening on idle channels [YHE02].

Algorithm 1 presents the high level steps for checking the above mentioned properties on global Scribble protocols. In short, the algorithm is based on the following steps:

- **Step 1:** Building a *time dependency graph*, a directed acyclic graph that models the actual causal dependencies between the actions in a protocol. The time dependency graph is built by traversing the Abstract Syntax Tree (AST) of a global Scribble protocol. The nodes of the graph model the actions of the timed protocol, annotated with constraints and resets, and the edges model the causal dependencies between actions.
- **Step 2:** For each node `n`, do a depth-first-search traversal (dfs) of the graph and find the set of paths from the initial node to `n`.
- **Step 3:** To model the range of 'absolute' times (i.e., *virtual time*) in which the state represented by `n` can be reached, we convert (shift) each time constraint with respect to the clock resets of the preceding nodes, yielding a *virtual time constraint* $\overline{\delta}_{\mathtt{n}}$ for each node `n`. Then the constraint for each node `n` is modified to account for the restrictions imposed by the virtual time constraints of all preceding nodes (constructed in Step 2) of `n`.
- **Step 4:** To check the feasibility property we build a logic formula to check that the modified constraint of a node `n` is satisfiable given the restrictions from previous nodes. For wait-freedom, we check that all solutions for a *receiving node* occur at the same or at a later time with respect to any solutions allowed by constraints of preceding nodes. We feed these formulas to an SMT solver to check if the formulas are satisfiable.

The algorithm terminates either when the maximum number of states are reached, that is all nodes are checked (Line 8 of Algorithm 1), or when a node is found for which the feasibility/wait-freedom formulae is unsatisfiable (Line 6-7 of Algorithm 1). The logic used for evaluating node satisfiability (Step 4) forms a subset of Presburger arithmetic, which is decidable (see e.g., [BYY14b]) therefore the termination of the algorithm depends on the termination of traversing all nodes. To ensure the graph traversing always terminates we impose a condition on time constraints used in recursion bodies (see [BYY14b]). More precisely, we consider only protocols that are infinitely satisfiable:

**Definition 3.1** (*Infinitely satisfiable*) A global protocol is *infinitely satisfiable* if: (1) constraints in recursion bodies have no resets, no equalities, no upper bounds or (2) all participants reset at each iteration.

```
rec Loop {
    n1: msg1(string) from A to B;
    n2: msg2(string) from A to C;
continue Loop;
  }
```

```
n1: msg1(string) from A to B;
n2: msg2(string) from A to C;
n3: msg1(string) from A to B;
n4: msg2(string) from A to C;
```

**Fig. 9.** An example of a recursion protocol (*left*) and its one-time unfolding (*right*)

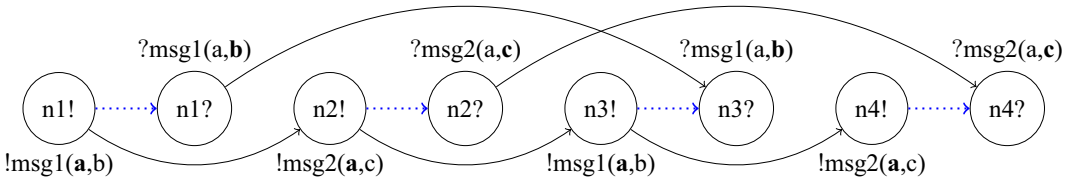


**Fig. 10.** Dependency graph for the protocol in Fig. 9

The above condition ensures that checking the one-time unfolding of each recursion is sufficient to ensure satisfiability of all successive unfoldings. This is the reason the graph traversing in Step 2 is done on acyclic graph, that is built from a global protocol after one-time unfolding of all recursions and subsequently replacing `continue t` with `end` in the body of the protocol. Therefore, the algorithm always terminate as it considers finite paths for a finite number of nodes.

The complexity of the algorithm 1 is mostly affected by creating the dependency graph (linear on the size of the protocol), on enumerating all paths from the root(s) to a node in the graph (polynomial on the size of the graph) and on the satisfiability of Presburger formulae. In general, the asymptotic running-time complexity of satisfiability of Presburger formulae is doubly exponential. SMT solvers use various techniques to reduce the running time and space, thus the precise complexity is solver specific.

In the rest of this section we explain each step of Algorithm 1 in detail.

### 3.1. Step 1: Build the time dependency graph

As shown in Algorithm 1 to make dependencies between constraints explicit when checking feasibility and wait-freedom, we create a representation of global timed protocols as time dependency graphs (hereafter dependency graphs). A dependency graph of a global protocol $G$ is a pair $(N, E)$ where $N$ denotes the set of nodes and $E$ denotes the set of edges.

To construct the time dependency graph we annotate in $G$ each syntactic occurrence of subterms of the form

$$[@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\ G'$$

with a node name (denoted by $\mathtt{n}_1, \mathtt{n}_2, \ldots$).

The nodes of the time graph have the form of $(\mathtt{n}, !, \mathtt{A}, \delta, \lambda)$ which represents a sending action from participant A and $(\mathtt{n}, ?, \mathtt{B}, \delta', \lambda')$ which represents a receiving action at participant B.

The time dependency graph represents all causal dependencies in a protocol. These dependencies are not explicitly captured as the syntactic order of interactions in a Scribble protocol does not necessarily imply a causal dependency between actions. We illustrate message causalities on an example. Consider the protocol in Fig. 9 (left), where we have annotated the interaction nodes (for simplicity, we have omitted time constraints):

As customary in multiparty session types, the receive action of `msg1` by B and of `msg2` by C could happen in any order due to asynchrony of communications, despite appearing in a specific syntactic order in `pro`. However, *some* causal dependencies are indeed enforced by the syntax of protocol `pro` above:

1. (I/O dependencies) The receive action of `msg1` by B causally depends from the send action of `msg1` by A. Thus, two nodes that are generated from the same interaction type should be connected by an edge, e.g $((\mathtt{n}_1, !, \mathtt{A}), (\mathtt{n}_1, ?, \mathtt{B})) \in E$ and $((\mathtt{n}_2, !, \mathtt{A}), (\mathtt{n}_2, ?, \mathtt{C})) \in E$;

2. (Single participant dependencies) The initial sending of msg1 must occur before the sending of msg2, by A (namely the syntactic order of actions in a protocol corresponds to an actual causal dependency when actions are performed by the same role). Thus, two consecutive interactions for the same participant should be connected by an edge, e.g. $((\mathtt{n}_1, !, \mathtt{A}), (\mathtt{n}_2, !, \mathtt{A})) \in E$;
3. (Recursion dependencies) Sending of msg2 by A happens both before and after sending of msg1 by A due to recursion. Following [BYY14a, BYY14b], we consider the class of global protocol that are infinitely satisfiable (Definition 3.1). Infinite satisfiability allows us to check for feasibility and wait-freedom by checking the one-time unfolding (unfolding all recursions in the protocol *only once*), as it guarantees that the same properties will then hold also in successive unfoldings. Figure 9 (right) shows the one-time unfolding for the protocol in Fig. 9 (left). Recursion dependencies, as the ones between msg1 and msg2, are captured in the dependency graph right away, by considering the one-time unfolding of the global protocol.

All the above dependencies are represented in the dependency graph. The dependency graph for the protocol in Fig. 9 is shown in Fig. 10 and the dependency graph for our running example is shown in Fig. 11. The causal dependencies between sending and their corresponding receive actions are represented by dotted edges, the ones reflecting the syntactic order of actions performed by the same role by solid edges, and recursion dependencies are directly captured (as single participant dependencies) by considering the one-time unfolding. As can be seen in the figures, an interaction in Scribble is represented as two nodes in the time dependency graph: a sending node (!(sender, receiver)) and a receiving node (?(sender, receiver)) connected by a directed edge from the sending to the receiving node. For readability, we keep both sender and receiver roles in the node representation but we bold the role that corresponds to subject(n), we also omit time constraints.

Formally, dependency graphs are defined inductively (Definition 3.2) using two functions, nodes($G$) and edges($G$), which given $G$ return the set of nodes and edges, respectively, of its dependency graph by checking all the sub terms of $G$. The function edges($G$) uses an auxiliary mappings D, initially empty. The mapping $\mathtt{D} : \mathcal{P}(G) \to \mathtt{nodes}(G)$ from participants to nodes is for constructing edges of type (2). We use the information in D to return the node for the last action of each participant in $\mathcal{P}(G)$.

**Definition 3.2** (*Dependency graph*) Let $G_0$ be a global protocol and $G$ be its one-time unfolding. The dependency graph of $G_0$ is a pair $(N, E)$ where $N = \mathtt{nodes}(G)$ and $E = \mathtt{edges}(G)$. The functions nodes($G$) and edges($G$) are defined as follows:

1. if $G'$ is $\mathtt{n} : [@\mathtt{A} : \delta, \mathtt{reset}(\lambda)][@\mathtt{B} : \delta', \mathtt{reset}(\lambda')]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\ G''$ then:
   - $\mathtt{nodes}(G') = (\mathtt{n}_1) \cup (\mathtt{n}_2) \cup \mathtt{nodes}(G'')$ where $\mathtt{n}_1 = (\mathtt{n}, !, \mathtt{A}, \mathtt{B}, \delta, \lambda)$ and $\mathtt{n}_2 = (\mathtt{n}, ?, \mathtt{A}, \mathtt{B}, \delta', \lambda')$
   - $\mathtt{edges}(G', \mathtt{D}) = (\mathtt{n}_1, \mathtt{n}_2) \cup (\mathtt{D}(\mathtt{A}), \mathtt{n}_1) \cup (\mathtt{D}(\mathtt{B}), \mathtt{n}_2) \cup \mathtt{edges}(G'', \mathtt{D}[\mathtt{A} \mapsto \mathtt{n}_1, \mathtt{B} \mapsto \mathtt{n}_2])$
2. if $G'$ is $\mathtt{choice}\ \mathtt{at}\ \mathtt{A}\ G_1\ \mathtt{or}\ \ldots\ \mathtt{or}\ G_n$ then
   - $\mathtt{nodes}(G') = \bigcup_{i \in [1 \ldots n]} \mathtt{nodes}(G_i)$
   - $\mathtt{edges}(G', \mathtt{D}) = \bigcup_{i \in [1 \ldots n]} \mathtt{edges}(G_i, \mathtt{D})$

Every time a node n is added to $N$, the mapping is updated $\mathtt{D}[\mathtt{A} \mapsto \mathtt{n}_1, \mathtt{B} \mapsto \mathtt{n}_2]$ and an edge between n and the D(subject(n)) is created, where subject(n) is the role element (the third or the forth element respectively) in n. Note that the choice itself does not introduce any new edges or nodes. The causality between the actions is preserved in the mapping D. All choice branches are traversed with the same mapping D.

### 3.2. Step 2: Collecting all paths to a node

After the time dependency graph is built, for each node n we collect all paths from the initial node to n. Each path to n represents a possible execution to the protocol leading to state n, and captures time constraints and resets that should be taken into account when building the formula to check if the constraint annotating n satisfies feasibility and wait-freedom. Noticeably, the number of paths to n is finite since the dependency graph is acyclic. For collecting all paths, we traverse the graph using a standard modification of depth-first-search with backtracking [Ski08].

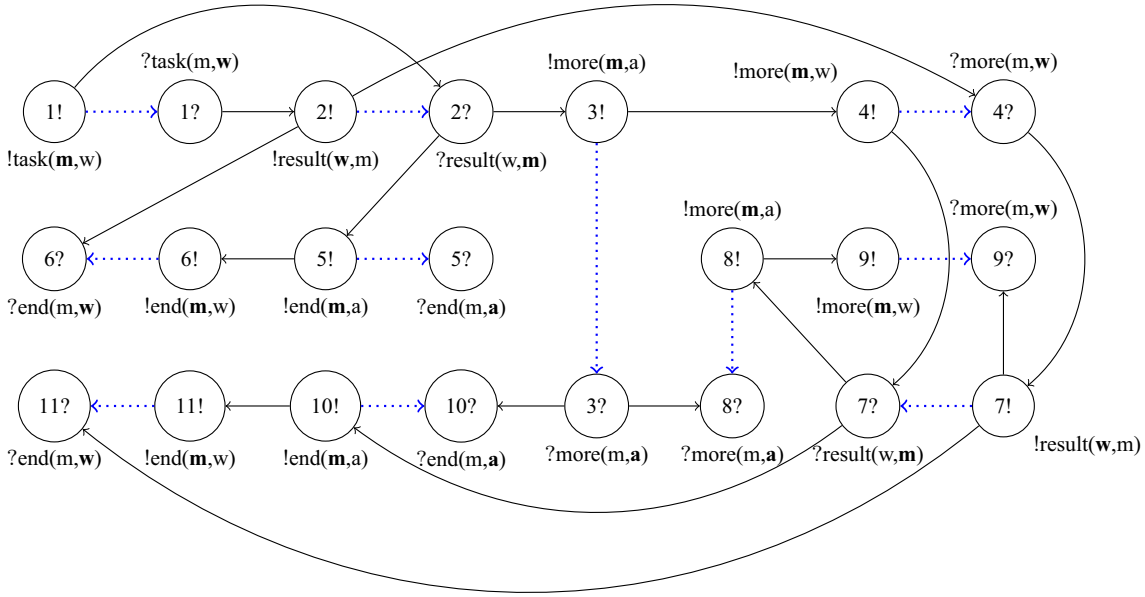


**Fig. 11.** Time dependency graph for the WordCount protocol

### 3.3. Step 3: Virtual time constraints

The value of a clock variable in a Scribble constraint represents the time elapsed since the previous resets of that clock variable. To reason on properties of a constraint, such as its satisfiability, we need to consider all possible time scenarios (e.g., all possible 'pasts') that lead to the execution point in which that constraint must be evaluated. We will do this relying on the notion of *virtual time* (that is the time elapsed since the beginning of the session) as opposed to relative time (time elapses since the previous clock reset) and, more precisely, of *virtual time constraint* of a node. The virtual time constraint $\overline{\delta}_{\mathtt{n}}$ of a node $\mathtt{n}$ of a time dependency graph models the possible virtual times in which the execution flow may reach a node $\mathtt{n}$. The virtual time constraint is calculated by taking into account constraints of past actions and by 'shifting' time to account for previous resets for the clock.

We illustrate Step 3 under the simplifying assumption that each role owns exactly one clock, hence each constrain in a Scribble global protocol has exactly one free clock variable. As remarked in [BYY14b], the extension to multiple clocks is considerably more verbose but does not pose qualitative challenges.

We denote by $fc(\delta)$ the free clock variable of $\delta$, and by $\vec{\mathtt{x}}$ a finite vector of clock variables. Assume $\mathtt{p}$ owns a clock, we denote with $\mathtt{x}_{\mathtt{n}}$ the state of the clock owned by $\mathtt{p}$ in node $\mathtt{n}$, where $\mathtt{subject}(\mathtt{n}) = \mathtt{p}$. We call $\mathtt{x}_{\mathtt{n}}$ a *clock state*. A clock state uniquely identifies (e.g., in a virtual time constraint) the value of the clock of $\mathtt{p}$ in a node $\mathtt{n}$. Below we give the extended definition of $\delta$, which we will use to reason on *clock states*.

**Definition 3.3** [*Extended clock constraints* (*adapted from* [BYY14b])] The set of extended clock constraints $\delta$ is:

$$\delta ::= \mathtt{true} \mid \mathtt{x} > e \mid \mathtt{x} = e \mid \neg\delta \mid \delta_1 \wedge \delta_2$$
$$e ::= \mathtt{x} \mid c \mid e + e$$

To calculate the virtual time in a node, we need to know how many times the clock variable, say $\mathtt{x}$, of that node, say $\mathtt{n}$, has been reset since the beginning of the session; to this aim, we define a function $R$ (Definition 3.4) that takes $\mathtt{n}$ and returns the sum of clock states in which $\mathtt{x}$ has been reset since the beginning of the session. On the basis of Definition 3.4, we obtain time constraints as to represent constraints on virtual (absolute) time as opposed to constraints on relative time. More precisely, we shift a time constraint for a node $\mathtt{n}$ with a clock $\mathtt{x}_{\mathtt{n}}$ by adding all previous states of the clock $\mathtt{x}$ where $\mathtt{x}$ has been reset. The definition of a sum of a time constraint and a clock variable is given in Definition 3.5. For example, consider the partial protocol given below.

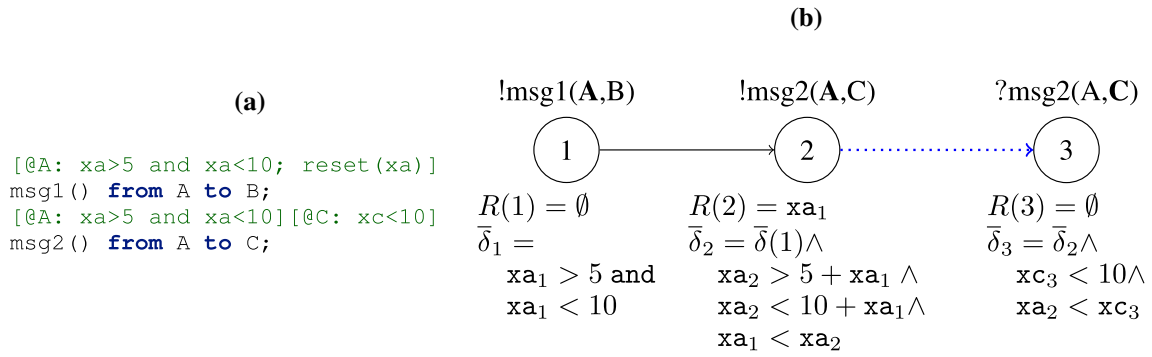


**Fig. 12.** An example of clock resets. **a** A Scribble protocol with resets. **b** Time graph with time constraints and dependency resets

```
[@A: xa>2; reset(xa)]
n1: msg1() from A to B;
[@A: xa<1]
n2: msg2() from A to C;
```

Assume $\mathtt{n}_{1!}$ and $\mathtt{n}_{2!}$ are the sending nodes for the first and the second interaction in the dependency graph for the protocol. Then $R(\mathtt{n}_{2!}) = \{xa_1\}$ and the clock $xa_2$, in constraint $xa_2 < 1$, represents a relative time for the clock at A, e.g $xa_2$ measures the elapsed time between the previous reset of clock $xa$, which happens at time $xa_1$ and the current time. If we modify the constraint $xa_2 < 1$ to $xa_2 < 1 + R$ (e.g $xa_2 < 1 + xa_1$) then $xa_2$ represents a value on the absolute timeline of $xa$.

Next we give the definitions of a reset function $R$ and of $\delta + \mathtt{x}$, adapted from [BYY14b].

**Definition 3.4** [*Reset function* (*adapted from* [BYY14b])] Let $\mathtt{n}$ be a node in the time dependency graph $N$ of $G$. We denote by $\prec_G$ the transitive closure of $<_G$.

The reset set $\overline{R}$ for a node $\mathtt{n}$ contains all clock states where the clock has been reset.

$$\overline{R}(\mathtt{n}) = \{\mathtt{x}_{\mathtt{n}'} \mid \mathtt{n}' \in N \wedge \mathtt{n}' \prec_G \mathtt{n} \wedge \mathtt{subject}(\mathtt{n}) = \mathtt{subject}(\mathtt{n}') \wedge \mathtt{resInfo}(\mathtt{n}')\}$$

where $\mathtt{x}'_{\mathtt{n}}$ is the state of the clock at node $\mathtt{n}'$ and $\mathtt{resInfo}(\mathtt{n}')$ is $\mathtt{true}$ if the clock has been reset at node $\mathtt{n}'$.

The reset function of $\mathtt{n}$ is defined as a sum of all clock states in $\overline{R}$.

$$R(\mathtt{n}) = \textstyle\sum_{\mathtt{n}' \in \overline{R}(\mathtt{n})} \mathtt{x}_{\mathtt{n}'}$$

**Definition 3.5** [*Sum of a constraint with a clock* (*adapted from* [BYY14b])] The sum of $\delta$ with a clock $\mathtt{x}$, written $\delta + \mathtt{x}$ is defined as follows:

$$\mathtt{true} + \mathtt{x} = \mathtt{true}$$

$$(\mathtt{x}' \ \mathtt{rop}\ e) + \mathtt{x} = \begin{cases} \mathtt{x}' \ \mathtt{rop}\ (e + \mathtt{x}) & (f(x, x') = \mathtt{true}) \\ \mathtt{x}' \ \mathtt{rop}\ e & (f(x, x') = \mathtt{false}) \end{cases} \quad \textit{with}\ (\mathtt{rop} \in \{>, =\})$$

$$(\neg\delta) + \mathtt{x} = \neg(\delta + \mathtt{x})$$

$$(\delta \wedge \delta') + \mathtt{x} = (\delta + \mathtt{x}) \wedge (\delta' + \mathtt{x})$$

$$\text{with} \quad f(x_{\mathtt{n}}, x_{\mathtt{n}'}) = \begin{cases} \mathtt{true} & (\mathtt{subject}(\mathtt{n}) = \mathtt{subject}(\mathtt{n}')) \\ \mathtt{false} & (\text{otherwise}) \end{cases}$$

The sum of $\delta$ with a vector of clocks is $\delta + \vec{\mathtt{x}} = (\delta + \mathtt{x}_0) + \vec{\mathtt{x}}'$ with $\vec{\mathtt{x}} = \mathtt{x}_0 + \vec{\mathtt{x}}'$.

Next we give the formal definition of a virtual time constraint of a node. We calculate the time scenario of a node recursively as to account for all previously occurred constraints.

**Definition 3.6** [*Virtual time constraint* (*adapted from* [BYY14b])] Let $\mathtt{n}$ be a node in the time dependency graph $N$ of $G$ with $\mathtt{const(n)} = \delta$, and $M = \{\mathtt{n}' \mid \mathtt{n}' \in N \text{ and } \mathtt{n}' <_G \mathtt{n}\}$ be the set of all nodes directly preceding $\mathtt{n}$, The virtual time constraint $\overline{\delta}_{\mathtt{n}}$ for a node $\mathtt{n}$ is

$$\overline{\delta}_{\mathtt{n}} = \begin{cases} (\delta[\mathtt{x}_{\mathtt{n}}/\mathtt{x}] + R(\mathtt{n})) \bigwedge\limits_{\mathtt{n}' \in M} (\overline{\delta}_{\mathtt{n}'} \wedge \mathtt{x}_{\mathtt{n}'} \leq \mathtt{x}_{\mathtt{n}}) & \text{if } fc(\delta) = \mathtt{x} \\ \bigwedge\limits_{\mathtt{n}' \in M} \overline{\delta}_{\mathtt{n}'} & \text{if } fc(\delta) = \emptyset \end{cases}$$

Note that if a node does not have incoming edges $\bigwedge\limits_{\mathtt{n}' \in M} \overline{\delta}_{\mathtt{n}'}$ and $\bigwedge\limits_{\mathtt{n}' \in M} (\overline{\delta}_{\mathtt{n}'} \wedge \mathtt{x}_{\mathtt{n}'} \leq \mathtt{x}_{\mathtt{n}})$ are trivially true.

In the above definition we assume that a node has an unique name $\mathtt{n}$. We first rename the clock in the constraint of $\mathtt{n}$ (e.g., $\delta[\mathtt{x}_{\mathtt{n}}/\mathtt{x}]$) by using this unique node name, e.g. $\mathtt{x}_{\mathtt{n}}$. In this way, $\mathtt{x}_{\mathtt{n}}$ uniquely identifies the state of the clock in that node. We shift the constraint to represent a constraint on the virtual timeline, e.g $\delta[\mathtt{x}_{\mathtt{n}}/\mathtt{x}] + R(\mathtt{n})$. Then we take into account all past constraints $\overline{\delta}_{\mathtt{n}'}$. We model the linear flow of time in causally related actions by requiring that the virtual time in node $\mathtt{n}$ is after the virtual time in preceding nodes, e.g $\mathtt{x}_{\mathtt{n}'} \leq \mathtt{x}_{\mathtt{n}}$.

We illustrate Step 3 by using the example in Fig. 12a. Figure 12b illustrates both the reset function and the virtual time constraints for nodes 1, 2 and 3. In the fragment of the time dependency graph for this example, shown in Fig. 12b, the receive action of `C` causally depends from the send action of `msg2` by `A` (node `2`) which, in turn, depends from the send action of `msg1` from `A` (node `1`). The constraints of node `1` and node `2` alone do not give sufficient information about the virtual time in node `3`. For instance, due to the reset of `xa` in the first action, the action in node `3` will be ready to be executed at an absolute time which is shifted of 5-to-10 time units. Namely, the virtual time for `3` is greater than 10 (i.e., the sum of the lower bounds of the constraints in node `1` and `2`, that is 5 + 5) and smaller than 20 (i.e., the sum of the upper bounds of the constraints in node `1` and `2`, that is 10 + 10). Since `xc` is never reset it will have, in node `3`, some value greater than 10 and smaller than 20. This means that node `3` will be reached when the constraint $\mathtt{xc} < 10$ cannot be satisfied. This example shows that it is necessary to take resets into account when evaluating the satisfiability of a constraint in a node. In this specific example, $\mathtt{xa}_1 \leq \mathtt{xa}_2$ is redundant as it is implicitly expressed by the 'shift' in $\overline{\delta}_2$. In general, however, as no resets may have occurred, this inequality is needed.

### 3.4. Step 4: Construct and check feasibility and wait-freedom formula

Up to this step, wait-freedom and feasibility share the same approach for (a) collecting all nodes occurring before a given node $\mathtt{n}$ (i.e., the nodes that are occurring before $\mathtt{n}$ in some one-unfolding path), and (b) modifying the constraints with renaming and, and (c) calculating the virtual time constraint $\overline{\delta}_{\mathtt{n}}$, a constraint giving a range of possible absolute times in which the execution of the protocol could reach the state modelled by $\mathtt{n}$. The difference in checking the two properties lies in the formulas for the satisfiability property for that node. These formulas are then solved by using an off-the-shelf SMT solver, Z3 [Z3C], which is a tool for checking satisfiability of logical formulas over one or more theories.

In this subsection we first remind the two properties, feasibility and wait-freedom, and give their representation as logical formulas, as shown in [BYY14b]. Then we present the formulas in a Z3 input format.

**Feasibility** of a global protocol requires the satisfiability of each constraint in the protocol, in every possible scenario that satisfies the previously occurred constraints. We define protocol feasibility in terms of node satisfiability. i.e a protocol is feasible if all nodes in its time dependency graph are satisfiable. For a node to be satisfiable its constraint must be satisfiable given all restrictions posed by constraints of preceding nodes. Next we give the formal definition of a satisfiable node, adapted from [BYY14b].[3]

**Definition 3.7** [*Satisfiable node* (*adapted from* [BYY14b])] Let $\mathtt{n}$ be a node of the (unfolding) time graph of a global protocol $G$, $\mathtt{const(n)} = \delta$, $\mathtt{fn}(\delta) = \mathtt{x}$, and $M = \{\mathtt{n}' \mid \mathtt{n}' \in N \text{ and } \mathtt{n}' <_G \mathtt{n}\}$. Node $\mathtt{n}$ is *satisfiable* if

$$\bigwedge_{\mathtt{n}' \in M} \overline{\delta}_{\mathtt{n}'} \supset \exists\, \mathtt{x}_{\mathtt{n}}.((\delta[\mathtt{x}_{\mathtt{n}}/\mathtt{x}] + R(\mathtt{n})) \bigwedge_{\mathtt{n}' \in M} \mathtt{x}_{\mathtt{n}'} \leq \mathtt{x}_{\mathtt{n}})$$

[3] In [BYY14b] the definition is given in the general case for multiple clocks per role, in this article we assume one clock per role.

The formulae expresses that given all previously occurred virtual time constraints $\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$ there is a solution $\mathtt{x_n}$ of the shifted time constraint for the given node ($\delta[\mathtt{x_n}/\mathtt{x}] + R(\mathtt{n})$) and this solution is not in the past (i.e., before the current virtual time), thus it is strictly bigger than any solutions $\mathtt{x}_{\mathtt{n}'}$ of previously occurred constraints.

The translation of the logical formulae to a format, accepted by the Z3 checker is straightforward. The formulae is given below, where $\bigcup_{\mathtt{n}'\in M} \mathtt{fn}(\overline{\delta}_{\mathtt{n}'}) = \{\mathtt{x}_1, \ldots, \mathtt{x}_m\}$, $\delta_{\mathtt{n}} = \bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$ and $\delta_{\mathtt{x_n}} = \delta[\mathtt{x_n}/\mathtt{x}]$

$$\text{ForAll}(\mathtt{x}_1, \ldots, \mathtt{x}_m, \text{Implies}(\delta_{\mathtt{n}}, \text{Exists}(\mathtt{x_n}, (\delta_{\mathtt{x_n}} + R(\mathtt{n})) \texttt{ and } \mathtt{x}_1 \leq \mathtt{x_n} \texttt{ and } \ldots \texttt{ and } \mathtt{x}_m \leq \mathtt{x_n})))$$

**Wait-freedom** requires that all solutions of time constraints of receivers in a global protocol do not precede solutions posed by previously occurred constraints. Next we give the formal definition of a wait-free node.

**Definition 3.8** [*Wait-free node* (*adapted from* [BYY14b])] Let $\mathtt{n}_?$ be a receiving node of the (unfolding) time graph of $G$ with $\mathtt{const}(\mathtt{n}_?) = \delta$, and $\mathtt{fn}(\delta) = \mathtt{x}$, and $M = \{\mathtt{n}' \mid \mathtt{n}' \in N \text{ and } \mathtt{n}' <_G \mathtt{n}\}$. We say that $\mathtt{n}_?$ is *wait-free* if

$$(\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}) \wedge (\delta[\mathtt{x_n}/\mathtt{x}] + R(\mathtt{n}) \supset \bigwedge_{\mathtt{n}'\in M} \mathtt{x}_{\mathtt{n}'} \leq \mathtt{x_n}$$

The formula states that all solutions $\mathtt{x}_{\mathtt{n}'}$ of virtual time constraints of nodes preceding node $\mathtt{n}_?$, e.g solutions of $\bigwedge_{\mathtt{n}'\in M} \overline{\delta}_{\mathtt{n}'}$, and all solutions $\mathtt{x_n}$ of the time constraint of $\mathtt{n}_?$, e.g solutions of $\delta[\mathtt{x_n}/\mathtt{x}] + R(\mathtt{n})$, are such that the virtual time at node $\mathtt{n}_?$ is after the time posed by the previously occurred constraints, e.g $\bigwedge_{\mathtt{n}'\in M} \mathtt{x}_{\mathtt{n}'} \leq \mathtt{x_n}$.

The formula, as accepted in Z3, is given below.

$$\text{ForAll}(\mathtt{x}_1, \ldots, \mathtt{x}_m, \mathtt{x_n}, \textit{Implies}(\delta_{\mathtt{n}_?} \texttt{ and } \delta_{\mathtt{x_n}}, \mathtt{x}_1 \leq \mathtt{x_n} \texttt{ and } \ldots \texttt{ and } \mathtt{x}_m \leq \mathtt{x_n}))$$

where $\mathtt{x}_1, \ldots, \mathtt{x}_m$ are the free variables of $\delta_{\mathtt{n}_?}$, and $\mathtt{x_n}$ is the (renamed) clock of $\mathtt{n}_?$.

A node $\mathtt{n}$ such that $\mathtt{subject}(\mathtt{n}) = \emptyset$ or $\mathtt{fn}(\mathtt{const}(\mathtt{n})) = \emptyset$ is always satisfiable and wait-free.

We feed the constructed formulas to the Z3 SMT solver and check if they are satisfiable. When these formulas are satisfiable for each node of a time dependency graph, then the original Scribble global protocol is feasible and wait-free. This follows from the fact that the checking method presented and implemented in this article closely follows the theoretical constructions used to prove decidability of feasibility and wait-freedom for infinitely satisfiable timed global protocol in [BYY14a, BYY14b] (Proposition 5.1).

## 4. Implementing timed protocols with python

To implement and verify timed local protocols we extend the Python monitoring framework, previously presented in [HNY+13], with time. The framework in [HNY+13] provides (1) a monitoring tool for runtime checking of untimed session protocols and (2) a library for distributed programming. The latter offers a high-level interface, called *conversation channel*, for communication programming. A conversation channel maps the interaction primitives of session types to lower-level communication actions on concrete transports. Our current implementation supports AMQP [AMQ] transport (a messaging middleware on top of TCP).[4] In summary, conversation channels provide functionality for (1) session initiation and joining and (2) sending and receiving of messages. Thus, role actions from a Scribble protocol should be implemented as actions on a dedicated conversation channel. Internally, each channel is implemented as a separate *greenlet* (or micro/green thread), which is a light-weight cooperatively-scheduled execution unit in Python.

We have extended our conversation channel API with two standard time primitives, *sleep* and *timeout*, and the monitor tool with capabilities for checking time violations.

Specifically, the API is extended with:

- a *delay* primitive which puts the current green thread to sleep for a prescribed time. The primitive is implemented using a lower level function provided by the 'gevent' library: `gevent.sleep` which lets time elapse for a specified amount of time.

[4] More precisely, the primitives for sending and receiving of a message use the primitives for sending and receiving from a Python communication library, called pikka https://github.com/pika/pika library, which is a Python implementation of the AMQP protocol.

```
1  def master_proc():
2      c = Conversation.create(...)
3      c.send('W', 'task',
4        'log', 'string')
5      c.delay(22)
6      c.receive('W')
7      while more_tasks():
8        c.send('A','more', 'data')
9        c.send('W', 'more', 'log',
10            'string')
11       c.delay(22)
12       c.receive('W')
13     c.send('A', 'end', 'data')
14     c.send('W', 'end')
```

**(a)**

```
def worker_proc():
    c = Conversation.join(...)
    c.delay(1)
    log = c.receive('M')
    while conv_msg.label != 'end':
        data = self.crawl(log,
                  timeout=20)
        c.send('M','result', data)
        c.delay(23)
        conv_msg = c.receive('M')
```

**(b)**

```
def aggregator_proc()
    c = Conversation.join(...)
    op = None
    while op !='end':
            c.delay(23)
            conv_msg = c.receive('M')
            op = conv_msg.label
```

**(c)**

**Fig. 13.** Participants implementation in Python. **a** Master program. **b** Worker program. **c** Aggregator program

- a *timeout* primitive which interrupts an ongoing computation to meet an approaching deadline. Timeouts are useful for *computation-intensive functions*, operations that take an amount of time which is not negligible such as, for instance, the log crawling performed by the worker in our running example. It may be difficult to foresee the exact duration of a computation-intensive function; in order to ensure that its execution does not exceed the time prescribed by the local protocol, we associate each computation-intensive function to a parameter `timeout` that is an upper bound to the duration of its execution; an exception is raised if the function is not completed in the given time frame. In the implementation of the worker in the running example, the function `self.crawl(log, word, timeout=20)` which interrupts the crawling after 20 seconds; the resulting exception can be handled by simply proceeding with the computation.

In [BYY14a] processes are modelled using a simple extension of the $\pi$-calculus with a delay operator `delay(t)`.$P$ that executes as process $P$ after waiting exactly `t` units of time. All of the other actions are assumed to take no time. In practice, however, operations do take time and the delay primitive models standard operations happening between communication actions, as well as time primitives such as delays and timeouts.

The monitor tool is augmented with a local clock and performs several checks summarised below:

- when an action (send or receive) is performed on the conversation channel, the monitor checks if the action is performed within the time constraints specified in a protocol;
- when a delay is issued, the monitor checks that the specified delay time does not exceed time constrains specified in the protocol; and
- when a timeout is issued, the monitor checks that the timeout does not exceed the prescribed time constraints.

We illustrate more concretely the primitives introduced earlier in this section through a Python implementation of the running example. Figure 13 shows the Python program for the roles of our running example.

The implementation for the master process is given in Fig. 13a.

Line 1-2 start creates a *conversation channel* `c`. Then, following the local protocol, the master sends a request to the worker passing the log name and the word to be counted. The send method, called on conversation channel `c`, takes as arguments the destination role, message operator and payload values. This information is encapsulated in the message payload as part of a conversation header and is later used for checking by the runtime verification module. The receive method can take the sender as a single argument, or additionally the operator of the desired message. The code continues with the *delay* operator. The implementation for the worker process is given in Fig. 13b; in Line 6 operation `self.crawl(log, word, timeout=20)` models a computation-intensive function. The aggregator process is shown in Fig. 13c.

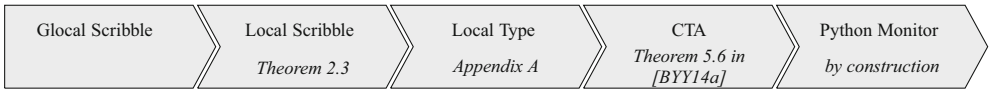


**Fig. 14.** The work-flow for deriving monitors from Global Scribble protocols

## 5. Runtime verification and enforcement of time properties

In this section we introduce our monitoring framework and discuss the challenges of monitoring timing of actions. A monitor acts as a membrane between one endpoint and the rest of the network, checking that the send and receive actions performed by that timed endpoint program conform to the implemented timed Scribble local protocols. The main property enforced by our framework is that in a network where all endpoints are monitored then either all actions will occur at the prescribed timing, or an error will be detected.

Figure 14 summarises the work-flow of constructing a configuration of local monitors from a global Scribble protocol: (1) a global Scribble protocol is projected into a set of local Scribble protocols; (2) each local Scribble protocol corresponds to a local type in [BYY14a]; (3) each local type corresponds to a CTA (Communicating Timed Automaton); and (4) a CTA is used to monitor an application.

The monitor has two purposes (or *modes*) with respect to time: error detection and error prevention/enforcement.

### 5.1. Error detection

The monitor verifies the communication actions of the monitored endpoint against Scribble timed local protocols, expressed as timed automata. First, the monitor verifies that the type (operation and payload) of each message matches its specification and that occurs in the right causal order w.r.t. the Scribble protocol (as in the untimed Scribble toolchain). Second, the monitor checks the correct timing of actions. For each ongoing protocol, the monitor is augmented with a *local clock*. A synchronisation has been introduced in the prototype to ensure that all processes and monitors will start a protocol at the same time, with clocks set to 0. When a timed endpoint program executes an action the monitor checks the clock constraint of that action (in the timed automaton) against the value of the local clock. The value of the local clock is determined simply as a difference between two timestamps (the initial and the current time). The time is measured using the built-in python function *timeit*.[5] If an action complies with the prescribed timing it is made visible (i.e., forwarded) into the network, otherwise the monitor raises a *TimeException*. For example, if we change the delay of the program in Fig. 13a to be `delay(30)` this will result in a *TimeException*. Thus the monitor stops the time error from propagating into the other endpoints.

### 5.2. Error prevention/enforcement

This mode relies on the error detection mechanism: when a violation occurs the monitor enforces the clock constraints by generating compensation actions, which postpone an execution . We have two types of scenarios: an action is launched by the local endpoint too early or too late (or not at all) w.r.t. the prescribed timing. In the first case, the monitor generates a *delay* equal to the time that is left until an appropriate time is reached, and then it forwards the action to the rest of the network. For example, if we delete the line `delay(20)` in Fig. 13a or modify it with a smaller delay then the monitor will introduce the missing delay so that the monitored application will appear correct to the network. When a deadline is reached but its associated action is still not executed, the monitor raises a *TimeoutException*. The application can try and recover itself using the exception handler, e.g., by interrupting an ongoing computation and continuing the conversation, or restarting the protocol with different settings.

The monitor looks at the next action prescribed by the timed automaton (or *prescribed action*) and acts according to the pre- and post-actions in the table. Pre-actions (resp. post-actions) denote actions performed by the monitor before (resp. after) that the timed endpoint program executes the action that corresponds to the prescribed action. Table 1 summarises the actions generated by the monitor in error prevention/enforcement mode. In the table $\mathtt{x}_{cur}$ is the local clock of the monitor.

[5] The function timeit returns the time in seconds since the epoch, i.e., the point where the time starts. As recommended in http://pythoncentral.io/measure-time-in-python-time-time-vs-time-clock/, this is the preferable method for measuring time in Python.

**Table 1.** Compensation actions generated by the monitor

| Prescribed action | Clock constraint | Pre-action | Post-action |
|---|---|---|---|
| `s.send` | $x \geq c$ | | `s.sleep`$(c - x_{cur})$ |
| `s.send` | $x \leq c$ | `s.tim eout`$(c - x_{cur})$ | |
| `s.recv` | $x \geq c$ | `s.sleep`$(c - x_{cur})$ | |
| `s.recv` | $x \leq c$ | | `s.timeout`$(c - x_{cur})$ |

```
global protocol WordCount2 (role M, role R, role W)
    [@M: xm<0.01,reset(xm)][@W: xw=0.01,reset(xw)]
    task(log,tring) from M to W;
    rec Loop{
        [@W: xw=0.20][xm@M: 0.21<xm<0.22]
        result(data) from W to M;
        choice at M{
          [@M: xm=0.22][@A: 0.23<=xa,reset(xa)]
          more(data) from M to A;
          [@M: xm=0.22,reset(xm)]
          [@W: xw=0.23,reset(xw)]
          more(log,string) from M to W;
          continue Loop;
        } or {
          [@M: xm=0.22][@A: 0.23<=xa]
          end(data) from M to A;
          [@M: xm=0.22][@W: xw=0.23]
          end() from M to W; } }
```

**Fig. 15.** A recursive protocol with resets

If the clock constraint of the prescribed action specifies a lower bound $x \geq c$ then the monitor introduces a delay of exactly $c$ (mapped to the low level Python *gevent.sleep* primitive). In case of send we have a post-action: the monitor sleeps *after* observing the action of the endpoint and forwards it to the network at the right time. In case of receive we have a pre-action: the monitor sleeps *before* observing the receive action so that the incoming message will be read at the appropriate time. Similarly, when the clock constraint specifies an upper bound $x \leq c$ the monitor inserts a *timeout* (a timer triggering a *TimeoutException*).

## 6. Benchmarks on transparency of timed monitors

The practicality of our timed monitoring framework depends on the transparency of the execution in a monitored environment. By transparency we mean: *a program that executes all actions at the right times when running unmonitored will do so when running monitored*. Transparency and overhead are closely related in the timed scenario, since the overhead introduced by the monitor may interfere with the time at which the interactions are executed. We have tested the transparency by providing two different protocols - a protocol with resets and a protocol without resets. The former proves the usability of the monitor in a typical scenario, while the latter demonstrates its limitations.

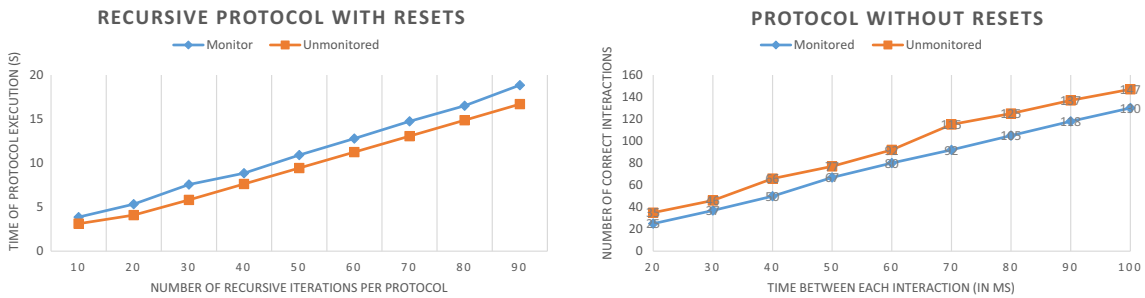


**Fig. 16.** Benchmark graphs. **a** The execution time per number of recursions for the protocol in Fig. 15. **b** The maximum number of correct interactions for the protocol in Fig. 17

To set up the benchmark, we have fixed a Scribble timed protocol and manually created a correct implementation of the participants in that protocol using our timed Python API. We run the implementation in two scenarios using our monitoring framework, and with the monitors 'turned off'. The graphs on Fig. 16 present the average execution time with a confidence interval of 95%. The result is obtained by repeating each example 30 times. This experimental design is influenced by [GBE07], where a typical large repetition size is $\mathtt{n} \geq 30$.

Participants were run as separate Python applications on the same machine (Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, Ubuntu 14.04.3 LTS), to minimise the latency between the endpoints. If latency is bigger, the protocol might become unsatisfiable and therefore an error will be triggered by the monitor before the completion of the protocol. For example, in a scenario where a sender's constraint is $x \leq 1$, a receiver's constraint is $x \leq 1$, and a latency is 1.5s (such latency is bigger than the monitor error margin), even if the sender sends the message on time, the monitor at the receiver endpoint cannot receive it on time due to network delay. We are not interested in testing such scenarios, because the errors are not due to monitor overhead. In cases of big latency the monitor correctly detects constraint violations. In the presented results, on average the latency between two endpoints is 0.04. The full benchmark protocols, the applications and the raw data are available from the project page [pyt]. To measure the execution time we use the built-in Python function *timeit*.[6]

**Scenario 1** We have initially considered a protocol with the same structure as the protocol in Fig. 3. Running the example with the initial time constraints, however, requires significantly long time runs where most of the execution time is spent performing *time.sleep* at each endpoint (see the Python implementations). Since we are interested in measuring the monitor overhead, which is not affected by how big the actual value constraints are, we decrease the constants in clock constraints and in *time.sleep* by a scale of 100. We used the implementation in Fig. 13 with delays updated to match the protocol, as shown in Fig. 15 (left). The outcome is presented in Fig. 16 (a). The graph illustrates the time for completing a protocol for increasing number of recursive executions.

This experiment shows that for the given protocol and implementation all executions are without constraint violations. Transparency is guaranteed (i.e., the overhead induced by the monitor does not affect the correctness of the program). Since resets prevent the monitor overhead to *accumulate up to a non negligible overall delay*, delays are bigger than the error margin. The monitor clock is reset at each iteration and therefore the monitor overhead does not accumulate.

**Scenario 2** Our second experiment was specifically targeted at checking how many interactions can generate a non-negligible accumulation of delays. We do this by removing resets. In case of no resets both the unmonitored and monitored programs are expected to start violating the constraints after certain number of executions. In Scenario 1 recursion allowed us to express repeated interactions by using resets. In order to observe a large number of repeated interactions *without resets* we have created the sequential protocol in Fig. 17 (left) and implementation (middle). We have generated a protocol with 200 consecutive point to point interactions happening at increasing times (at each interaction the time is increased by $c$). We run the experiment for different values of $c$ (horizontal axis on the figure) and measure the maximum number of interactions (vertical axis on the figure) that can be executed before the program violates the time constraint.

The experiment confirmed that, the monitored application performs 90% of the maximum number of possible interactions. This example comes to show the limitations of the timed monitoring framework. The practical scenarios we have encountered so far did not include long sequences of interactions, and repetitive operations are handled via recursions with resets at each cycle.

**Further discussion on benchmarks** Overall, the time transparency of the monitoring framework depends on the following:

- *Monitor overhead*. The complexity of monitor checking is linear in the number of protocol states and therefore the upper bound of the overhead can be approximated. Note that errors can be caused by accumulated monitor overhead. If clocks are not reset, the overhead of the monitor is propagated at each run. The accumulated delay can be calculated as a sum of the delays in the longest path between two actions without resets and executed on the same participant in the dependency graph.

[6] The function *timeit* returns the time in seconds since the epoch, i.e., the point where the time starts. As recommended in http://pythoncentral.io/measure-time-in-python-time-time-vs-time-clock/, this is the preferable method for measuring time in Python.

**(a)**

```
global protocol ClientServer(
  role C, role S)
    [@C: xc<c][@S: xs=c]
    ping(data) from C to S;
    [@C: xc<2*c][@S: xs=2*c]
    ping(data) from C to S;
    [@C: xc<3*c][@S: xs=3*c]
    ping(data) from C to S;
    [@C: xc<3*c][@S: xs=4*c]
    ping(data) from C to S;
    ...
    [@C: xc<200*c][@S: xs=200*c]
    ping(data) from C to S;}
```

**(b)**

```
def client_proc(t):
    c = Conversation.
      create(...)
    c.receive('S')
    while true:
      c.delay(t)
      c.receive('W')
```

**(c)**

```
def server_proc(t, n):
    c = Conversation.
      create(...)
    c.send('C')
    for i in range(0, n)
      c.delay(t)
      c.send('C')
```

**Fig. 17.** A protocol with accumulating delays (**a**) and a Python implementation for Client (**a**) and Server (**b**) roles

- *Error margin.* The monitor accepts an application as correct if the timing $t$ for a state $\mathtt{n}$ conforms to the time constraints $\delta(\mathtt{n})$ in the specification within a certain margin of error $\epsilon$. More precisely, assume that $t(\mathtt{n})$ is a function that returns the current time for the clock for a state $\mathtt{n}$. Then the monitor checks if $[t(\mathtt{n})-\epsilon, t(\mathtt{n})+\epsilon] \cap \text{sols}(\delta(\mathtt{n})) \neq \emptyset$ where $\text{sols}(\delta(\mathtt{n}))$ denotes the set of solutions for $\delta(\mathtt{n})$. The error margin $\epsilon$ is a parameter set to the monitor system during its initial configuration. The error margin is application and environment specific, but it should be at least as big as the time accuracy guaranteed by the execution environment. For example, in the Python documentation, the preciseness of the *time* function is specified as follows: *"It returns the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second."*[7]. Therefore, for our benchmark experiments we have chosen an error margin within a second of the specified constraints ($\epsilon = 0.5s$).

Time errors can also be caused by the execution environment when the machine (or the network) is overloaded. Other system activity can make the kernel unable to schedule the endpoint process as soon as needed. The purpose of the monitor is to detect time errors at their first occurrence and to stop time drifts propagating to other endpoints. Therefore, although in the above case time violations are not a result of a programming error, the executed program should be flagged as wrong, because the time constraints in the protocol are not satisfied.

Note that when the system is under load the time requirements will be violated due to the high load of the system, not because of the overhead induced by the monitor. However, the system load have an impact on the protocol execution, because in a system with inconsistent time drifts the protocol constraints can become unsatisfiable. As a future work we plan to identify formal transparency requirements to calculate time drifts that make a protocol unsafisfiable. We also plan to test the preciseness of time violations on real-time kernels, such as the Ubuntu RT_PREEMPT patch set. This will allow us to specify and enforce hard time deadlines. In an operating system without real-time guarantees, our framework reasons only about soft time deadlines, such as the deadline patterns expressed in Sect. 7.

## 7. Temporal patterns in global protocols

In this section we present a number of timed patterns that we have collected from literature, and which include industrial cases studies [BFM98, KCD+09], verification tools [UPP, CPS09] and web service specifications such as the Twitter API and Simple Mail Transfer Protocol (SMTP). Each pattern will be first introduced with a motivation and reference to literature, then modelled using Scribble and applied to a real-case scenario. The given set of patterns does not aim to be exhaustive, but to allow us to asses the usability of Scribble in known timed scenarios. Table 2 summarises the timed patterns, with reference to the corresponding use cases and references to literature.

**Request-response timeout pattern** This pattern allows to enforce quality of service requirements on the timing of a response. The requirement can be set either at the server side or at the client side, as we identified in the two use cases that follow. In the first use case, drawn from [CPS09], a service is requested to reply in a timely manner: *"An acknowledgement message* `ACK` *should be sent (by the server) no later than one second after receiving the request message* `REQ`*"*.

[7] https://docs.python.org/2/library/time.html

**Table 2.** Timed patterns

| Pattern | Use case | Source |
|---|---|---|
| Request-response timeout | Travel agency, SMTP | [CPS09, SMT] |
| Messages in a time frame | Denial of service attacks, Travel agency | [CPS09] |
| Action duration | Progress properties, User inactivity | [UPP] |
| Repeated constraint | Pull notification services | [OOI] |

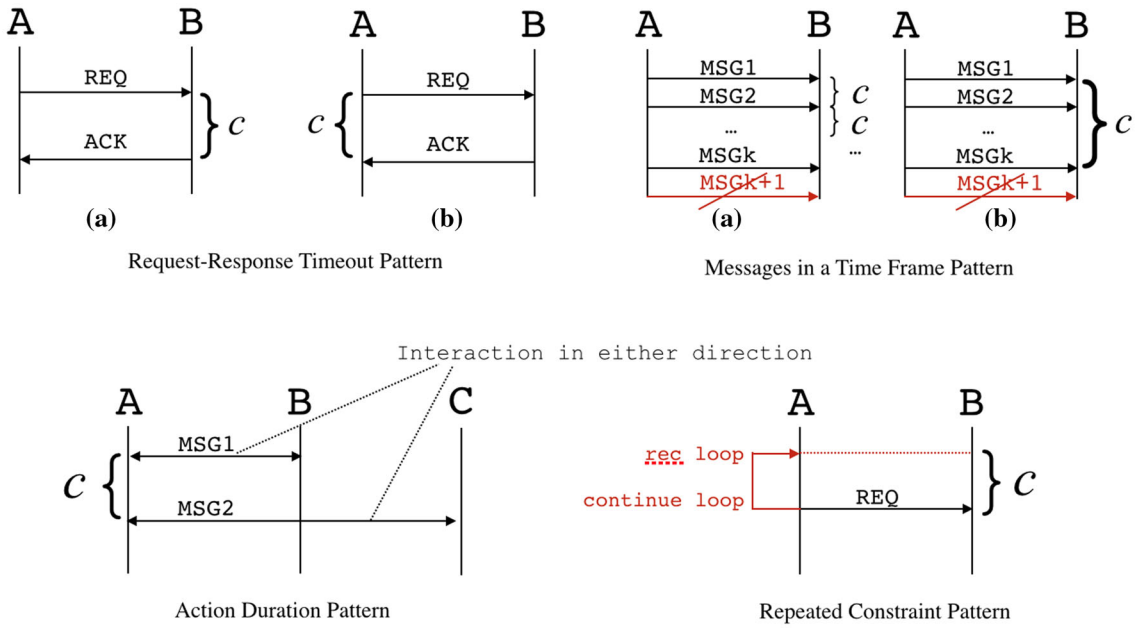


**Fig. 18.** An illustrated summary of the timed patterns

Namely, the *sending* of a reply message should be executed within a fixed time after the corresponding request message has been *received*. In the second use case, the Travel Agency web service specification, also drawn from [CPS09], the timeout is specified at the client side : *"A user should be given a response* `RES` *within one minute of a given request* `REQ`*"*. In this case, it is the *receiving* of the response from the server that must be possible after the request, within the specified timeout.

The above requirements can be generalised by the following pattern, which is also illustrated in Fig. 18:

(a) After receiving a message `REQ` from role `A`, role `B` must send the acknowledgment `ACK` to `A` within $c$ time units.

(b) After sending a message `REQ` to role `B`, role `A` must be able to receive the acknowledgment `ACK` from `B` within $c$ time units.

The corresponding skeletal Scribble specification is given in Fig. 19, where in (a) the sending of a reply message should be executed within a fixed time after the corresponding request message has been received, and in (b) the receiving of a message after sending a request should be possible within a timeout. A concrete specification can be obtained by instantiating `REQ`, `ACK`, `A`, and `B` with actual messages, roles, and $c$ with a value in $\mathbb{Q}^{\geq 0}$. In Fig. 19a, `xb` is a clock of `B` and "..." stands for any clock constraints, clock resets, or for any interaction that does not include resets to `xb` or `ACK`.

```
[@A : ...][@B : reset(xb)] REQ  from A to B;
...;
[@B : xb<=c][@A : ...] ACK from B to A;
```

**(a)**

```
[@A : reset(xa)][@B : ...] REQ  from A to B;
...;
[@B : ...][@A : xa<=c] ACK from B to A;
```

**(b)**

**Fig. 19.** Request-response timeout pattern in scribble: server side (**a**) and client side (**b**)

```
[@U : reset(xu1)][...] MAIL  from U to S;
[@U : reset(xu2)][...] DATABLOCK from U to S;
[...][@U : xu2<=180000] DATABLOCK_REPLY from S to U;
[...][@U : xu1<=300000] MAIL_REPLY from S to U;
```

**Fig. 20.** SMTP timeouts in Scribble (Request-Response Timeout Pattern)

```
Ga    = [@A:reset(xa)][@B:...] MSG from A to B; Ga(k-1)
Ga(k) = choice at A {
          [@A:xa>=c and xa<=d, reset(xa)][@B:...] MSG from A to B; Ga(k-1)
        } or {
          Ga(0)
              }
Ga(0) = [@A:xa>=c and xa<=d][@B:...] END from A to B; End

Gb    = [@A:reset(xa)][@B:...] MSG from A to B; Gb(k-1)
Gb(k) = choice at A {
         [@A:xa>=c and xa<=d][@B:...] MSG from A to B; Gb(k-1)
        } or {
          Gb(0)
              }
Gb(0) = [@A:xa>=c and xa<=d][@B:...] END from A to B; End
```

**Fig. 21.** Messages in a time frame pattern in scribble: single message time frames (`Ga`) and overall timeframe (`Gb`)

First, the time of receiving the request `REQ` is recorded by resetting the clock `xb`, and then the clock constraint `xb` $\leq c$ at the reply interaction `ACK` sets the maximum delay to be of $c$ time units. The specification in Fig. 19b is similar, but with reset and constraints for the user clock `xa`.

In Fig. 20 we present an example from the SMTP protocol, featuring a composition of instances of the Request-Response Timeout Pattern (b). The specification prescribes that *"A user should have a 5 minutes timeout for the* `MAIL` *command and 3 minutes timeout for the* `DATABLOCK` *command"*. The combination of two instances of the pattern, at the client side, requires the use of two clocks, one for the `MAIL` command (clock `xu1`), and one for the `DATABLOCK` command (clock `xu2`). In the constraints, note that 3 (reps. 5) minutes correspond to 180000 (resp. 300000) milliseconds.

**Messages in a time frame pattern** This pattern allows to set a limit to the number of messages that can be sent in a given time frame. Examples of this requirement can be found in specifications for denial of service attacks [CPS09]:*"A user is allowed to send only three redirect messages to a server with interval between the messages no less than two time units"*, or in other scenarios such as the Travel Agency Service in [CPS09]: *"A customer can change the date of his travel only two times and this must happen between one and five days of the initial reservation"*. These specifications express the repetition of a specified number of messages, occurring either (a) at a specified pace or (b) within an overall specified time-frame.

The above requirements can be generalised by the following pattern, which is also illustrated in Fig. 18:

(a) Role `A` is allowed to send role `B` at most $k$ messages, and at time intervals of at least $c$ and at most $d$ time units.
(b) Role `A` is allowed to send role `B` at most $k$ messages in the overall time frame of at least $c$ and at most $d$ time units.

Figure 21 shows the abstract Scribble protocol for this pattern, where `Ga` accounts for case (a) and `Gb` for case (b). The difference is that in (a) the clock `xa` is reset at each interaction. Note that in this pattern we have a-priori knowledge of $k$, hence any instantiation of the pattern can be resolved as an enumeration of interactions (i.e., without using loops). The definition uses a parametric notation `Ga(k)` for the Scribble 'body' defined in terms of `Ga(k-1)` for the sake of a general and simpler definition of the abstract Scribble protocol. Concrete specification can be obtained by instantiating `A`, and `B` with actual roles, $c$ and $d$ with values in $\mathbb{Q}^{\geq 0}$, and $k$ with non-negative integer values. Note that scenarios in which `A` is supposed to send *exactly* (and not at most) $k$ messages can be modelled by omitting the 'choice' construct and only keeping the first branch (i.e., removing the branch that jumps straight to `Ga(0)` or `Gb(0)`).

Figure 22 shows the protocols for the denial of service attack and the Travel Agency Service in [CPS09]. Here we comment on the latter. First, we specify the resetting of the user clock `xu` on the interaction `reservePack`. Then the clock constraint `xu` $> 1$ and `xu` $< 5$ are set on the following messages `changePack` which are set to be at most two. In this case, we did not convert days into milliseconds for readability.

```
Ga   = [@U:reset(xu)][...] REDIRECT from U to S; Ga(2)
Ga(k) = choice at U {
        [@U:xu>=2, reset(xu)][...] REDIRECT from U to S; Ga(k-1)
      } or {
        [@U:xu>=2, reset(xu)][...] END from U to S; End
          }

Gb   = [@U:reset(xu)][...] reservePack from U to S; Gb(2)
Gb(k) = choice at U {
        [@U:1<=xu<=5][...] changePack from U to S; Gb(k-1)
      } or {
        [@U: true][...] END from U to S; End
      }
```

**Fig. 22.** Denial of service attack (`Ga`) and Travel Agency Service (`Gb`) use cases in Scribble

```
[...][@B: reset(xb)] MSG1 from A to B;
[...][@B: xb<=30, reset(xb)] MSG2  from A to B;
```

```
[@A: reset(xa)][...] MSG1 from A to C;
[@A: xa<=3][...] MSG2  from A to B;
```

**Fig. 23.** Protocols for web service user inactivity (**a**) and for the UPPAAL progress property (**b**)

**Action duration pattern** This pattern sets a constraint on the delay which is allowed for a participant before executing an action. For example, a common requirement in web services is that: "*A user should not be inactive more than* 30 *minutes*". This pattern can also express progress properties verified by the UPPAAL model checker [UPP] such as: "*A user is allowed to stay in the state for no more than three time units*".

The constraints above can be generalised as follows: the time elapsed between two actions performed by the same role `A`, where each of these two actions can be either a send or a receive action, must not exceed $c$ time units. Notice that, unlike in the Request-Response Timeout Pattern, the constraint is specified only in terms of `A` and not on the type of the previous actions performed by `A` or the other roles interacting with `A`.

Figure 23a, b present the two instances of the Action Duration Pattern in Scribble: the protocols for web service user inactivity and the one for the UPPAAL progress property, respectively. In Fig. 23a the clock `xa` is reset on the first message `MSG1` from `A` to `B`. Then the clock check `xa` $\leq 30$ guarantees that the user cannot send another message `MSG2` if the time constraint is violated. The verification of the progress property in Fig. 23b is similar, but more roles are involved (`A`, `B`, `C`). The clock is reset during the interaction between `A` and `C`, while the clock constraint is checked on the interaction between `A` and `B`.

**Repeated constraints pattern** The pattern captures requirements occurring in pull notification services, where the intervals at which a certain interaction should be repeated is fixed. For example: "*The email client should request the emails from the service every 5 seconds*".

In general: role `A` must send messages to `B` every $c$ time units. The main differences with the Messages in a Time Frame Pattern is that the number of messages is not bounded, and the time is required to elapse between messages of exactly $c$ time units. For readability (i.e., to avoid nested 'choice' constructs) we will show the abstract Scribble specification in the case of non-terminating interaction (Fig. 24).

The repetition of the pull interactions is expressed via recursion (`recPull`). At every recursive iteration, the clock `xa` for `A` is checked against the constraint and then reset.

```
rec Pull{
  [...][@A: xa==c; reset(xa)] REQ  from A to B;
continue Pull;}
```

**Fig. 24.** Repeated constraints pattern in scribble

## 8. Related and future work

**Untimed runtime monitoring** Recent works on multiparty session types (MPST) [HNY+13, DHH+15, NYH13] present a runtime verification framework for Python programs. The Scribble toolchain in [NYH13] enhances the protocol specifications with assertions on the message payload. We have built upon this feature, expressing time constraints as assertions in our syntax. There is a substantial difference between the timed and the untimed monitoring framework. We have shown that ensuring time-consistency over timed global protocols is non trivial. It requires additional verification checks to be performed, namely feasibility and wait-freedom. To enable the verification of real-time distributed systems we have designed a timed API for Python and enriched the monitor capabilities, providing recovery mechanism for violated time constraints, which were not explored in the previous works. Preserving time-transparency for monitored systems is a new challenge with time. We have demonstrated preliminary results on the implications of the dynamic verification overhead over program correctness. We plan to conduct further benchmarks to identify a necessary set of transparency requirements for monitoring timed applications.

**Specification languages** The need for specifying and verifying the temporal requirements in a distributed systems is recognised. To this aim, different specification methods and verification tools have been developed, especially in the area of business process modelling (see [CKGJ13] for a survey on verification of temporal properties). The work in [WIH11] describes a framework for analysing choreographies between BPEL processes with time annotations. [GDZ12] extends BPML with time constraints. Via a mapping from BPML to timed automata, it allows verification with the UPPAAL model checker. As a language for timed protocol specifications, the main advantage of Scribble over alternatives such as BPEL, BPML and timed automata, is that it enables enforcement of global properties, e.g., conformance of the interactions to global protocols, providing an in-built mechanism (projection) for decentralisation of the verification. Another expressive formal language for specifying time-properties is XTUS-Automata, introduced in [KCD+09]. Specifications written in XTUS-Automata are automatically translated into BPEL aspects, which ensure temporal constraints at runtime. Both absolute and relative time can be specified. Their approach detects contradictions between the temporal constraints implemented in the composition, but does not verify inconsistency at the specification level, as the Scribble checker can offer.

**Verification tools and frameworks** Among the state-of-the-art runtime verification tools, a few support specification of time properties [dBdGJ+14, CR07, CPS09]. The work in [CR07] presents a generic monitor that can be parameterised on the logic. [dBdGJ+14] combines temporal properties and control flow specifications in a single formalism specified per object class. Our enforcement mechanism resembles the aspect-oriented approaches used in those verifiers, but the combination of control flow checking and temporal properties in the same global specification is an unique characteristic of our work. Our tool statically checks the correctness of the specification itself in addition to the runtime checks for the program. Furthermore, via its formal basis in [BCD+13], our framework allows to combine static verification [BYY14a] and dynamic enforcement.

Other works from the multi-agent community have studied distributed enforcement of temporal properties through monitoring. [MU00] presents a distributed architecture for local enforcements, where monitors detect if agents fulfil their specification within a given deadline. The specifications are expressed in the form of event-condition-action and all agents belonging to a group obey the same roles. Our work presents a different perspective where agents follow personalised laws based on the role they play in each protocol. In runtime verification for Web Services, the work [C+11] transforms a subset of Web Services Choreography Description Language into timed-automata and proves their transformation is correct with respect to timed traces. Their approach is model-based, static and centralised, and does not treat either the runtime verification, or the properties of feasibility and wait-freedom.

In the MCS (Message Sequent Charts) framework [Int98], several articles [HJ00, AHJ15] show that asynchronous processes, following specifications obtained by a simple projection on each component in High-Level Message Sequent Charts (HMSC) [Int98], allow more behaviour than those specified by the HMSCs. Specifically, the work [HJ00] discusses two necessary and sufficient conditions, *order reconstructability* and *locality of choice*, for ensuring correctness of projection of HMSC to CFSMs. The former condition disallows choices where the order of messages cannot be reconstructed from the local behaviour (as in the example on Fig. 7a). The latter condition disallows distributed choices (as the one in the example on Fig. 4) in order to enforce a single party to decide in each choice branch. In our framework, these conditions are automatically induced by a syntactic limitation given by the global types and the projection rules in Definition 2.3. For example, order reconstructability follows from third case of the projection rule of the branching (Definition 2.3) which enforces all branches to be

mergeable when the choice is invoked by different participants. Similarly for locality of choice: the projection rule of the branching (Definition 2.3) requires choices to have the same senders and receivers in order for the global protocol to be projectable. As we have discussed in Sect. 2.3, the protocols in Fig. 7a and Fig. 4 are indeed not projectable.

In the MSC framework, several tools for projecting local specifications from global ones are implemented (a survey is given in [LDD06]). These framework avoid projecting into *incorrect local behaviour* (such as deadlock) or detect incorrect behaviour. The Scribble toolchain prevents incorrect behaviour in local specifications by detecting inconsistent (non-projectable) global protocols, and generates dynamic monitors to enforce that programs (e.g., Python code) conform to the projected protocols.

**Tractable verification of distributed systems** The particular shape of interaction enforced by MPSTs and inherited by Scribble, as outlined in Sect. 2.1, is critical to guarantee tractability of verification based on MPSTs (e.g., relizability and progress). For example, these syntactic constraints yield, in the untimed scenario, a correspondence between MPSTs and a subclass of Communicating Finite State Machines (CFSMs) that, against the general case, satisfy progress and liveness properties [DY13]. Related works on the verification of distributed protocols based on high-level MSC [AEY05, Loh03] tackle the problem of realizability (if there is a distributed implementation that generates precisely the behaviors in the graph). These works do not consider time constraints, and relate to ours mainly through the restrictions we impose by projection (which we inherit from MPSTs). The precise correspondence between implementable MSC, as identified by [Loh03], and projectable multiparty session types is an interesting future research direction.

**Time properties** Several works study properties of timed formalisms, especially in the contexts of automata [KY06a, KY06b] and MSC [GMNK09, AGMK10]. In the timed scenario, problems such as reachability are known to be undecidable in the general case. This has been shown, for example, for timed extensions of Message Sequence Charts (e.g., Time-Constrained MSC [GMNK09]) and CTA [KY06a]. Timed MPSTs [BYY14a] tackle tractability issues by relying on the syntactic constraints also inherited by Scribble and mentioned in Sect. 2.1, and a few extra constraints on time annotations of global protocols. In particular:

- The correspondence between timed MPSTs and CTA [BYY14a], ensuring the progress and liveness properties studied in [DY13], requires an additional constraint on the shape of global protocols, that is feasibility (recall that it ensures that at any point of the protocol the current time constraint should be satisfiable for any possible past).
- Progress of a well-typed program requires two additional properties: feasibility (as above) and wait-freedom.

The work in [BYY14a, BYY14b] proposes a decidable method to check if a global type is feasible and wait-free. This method, however, relies on a further restriction on global types, called *infinite satisfiability* and explained in Sect. 3. Infinite satisfiability has recently been relaxed [BLY15] to allow some clocks not to be reset, as long as there is a viable 'escape' from the loop, but has not yet been integrated in the Scribble toolchain. As the focus of this work is to illustrate the implementation, practicality and usability aspects of the timed Scribble toolchain, we will not recall the technical details of the theory here. The interested reader can refer to [BYY14a, BYY14b, BLY15] for these.

Other approaches focus on the reachability problem, and are related to our algorithms for checking feasibility and wait-freedom of timed global specifications. [Tri99] gives an algorithm to check deadlock freedom for timed automata. The algorithm is based on syntactic conditions on the states relying on invariant annotations. Their approach is not directly applicable to check feasibility on timed global specifications. Other methods based on CTA [KY06a, KY06b] or MSC [GMNK09] address complexity by restricting topology and/or channel size. Remarkably, none of the conditions required by timed MPSTs (hence Scribble) set limitations on the network topology nor buffer size. For example the simple Scribble example below

```
rec pro [@A : x < 10, reset(x)][@B : true] a(T) from A to B;  pro
```

allows the channel from `A` to `B` to have an arbitrary number of messages (as `B` can arbitrarily delay the receive actions). Our restrictions are, rather, induced by the conversation structure of timed global protocols. Tractability of MPSTs derives from the way interactions are structured and the way resets are organised.

The work in [AGMK10], focused on an extension of MSCs with timed events, is particularly related to [BYY14a] (on which theory this article is based): the former focuses on the problem of language inclusion while the latter on statically checking conformance of abstract session types with concrete programs. The work in [AGMK10] gives a verification method that is decidable when the communication graph describing the behaviour of each loop can be modelled as a single strongly connected component, which implies that channels have an upper bound. In this article we hinge at the theory for static checking from [BYY14a] but practically apply it to dynamic checking.

**Calculi with time** Our timed API is based on the session $\pi$-calculus with delays, introduced in [BYY14a]. The delay primitive from [BYY14a] has been used as a model for the timeout and sleep extensions of the Python API presented in Sect. 4. Other recent works that extend process calculi with time are, for example, [BY07, LP12, LZ02] where timeouts are added to the $\pi$-calculus syntax. Time elapses as discrete ticks and delays propagate asynchronously through the system. Timed COWS [LPT07] extends COWS with a 'wait' primitive similar to our delays. The work in [GR+97], which is targeted to testing, introduces a 'wait' construct along with delay annotations for actions. [SG13] extends the $\pi$-calculus with absolute and continuous times, and includes timed constraints inspired by timed automata. Web-$\pi$ [LZ05] and $C^3$ [LP12] extend the $\pi$-calculus and the Conversation Calculus (CC), respectively, to enable the reasoning on the interplay between time and exceptions.

The main focus of our work is different from the above works: we use timed specifications *protocols*, rather than enriching timed primitives in the $\pi$-calculus (Python) syntax level. However, these works inspire several future directions for our framework. Extensions allowing dynamic time-passing as in [SG13] and [LZ02] at the $\pi$-calculus level are possible, for instance, by extending with time the approaches in [BHTY10] and [BDY12], in which the global specifications directly model properties of the message contents. A recent work [HNY+13, DHH+15] introduces exception handling constructs to the Scribble toolchain. We plan to integrate our time annotations into the full Scribble syntax and to investigate the implications of propagating timeouts and delays as exceptions.

## 9. Conclusion

This work presents the design and implementation of a real-time verification framework. At the specification level our time extension controls and checks, via timed Scribble, the feasibility and wait-freedom of processes at the early design phase. At the programming level, we have introduced primitives for a fine grained management of time which enable to schedule interactions at exact timings w.r.t. a given protocol. At runtime, early error detection is guaranteed by raising time exceptions when time deadlines are not met. Furthermore, the monitor is augmented with enforcement capabilities for recovering from runtime violations of the time constraints. Benchmarking has revealed an interesting relationship between transparency and overhead introduced by time: monitoring overhead may break transparency by introducing delays, hence violations. The practicality of the proposed approach for specification and dynamic verification of distributed interactions has been demonstrated via: (1) the representation of a number of scenarios (i.e., a OOI use case as well as time patterns distilled from literature) with Scribble, and (2) benchmarking, showing that the overhead introduced by our monitor is, in the scenarios we encountered, negligible.

## Acknowledgements

We thank the anonymous reviewers for their insightful comments, which helped us to improve the article. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1,EP/N027833/1 and EP/N028201/1; by EU FP7 612985 (UP- SCALE), COST Actions IC1201 (BETTY).

$$\frac{j \in I \quad A_j = \{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\} \quad \nu \models \delta_{\mathtt{O}} \quad \nu' = [\lambda_{\mathtt{O}} \mapsto 0]\nu}{(\nu, \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{T}_i\rangle\{A_i\}.\underline{G}_i\}_{i\in I}) \xrightarrow{\mathtt{AB!a}_j\langle \mathtt{T}_j\rangle} (\nu', \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}_j\langle \mathtt{T}_j\rangle\{A_j\}.\underline{G}_j)} \lfloor \textsc{select} \rfloor$$

$$\frac{\nu \models \delta_{\mathtt{I}} \quad \nu' = [\lambda_{\mathtt{I}} \mapsto 0]\nu}{(\nu, \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{T}\rangle\{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}.\underline{G}) \xrightarrow{\mathtt{AB?a}\langle \mathtt{T}\rangle} (\nu', \underline{G})} \quad \frac{(\nu, \underline{G}[\mu \mathbf{t}.\underline{G}/\mathbf{t}]) \xrightarrow{\ell} (\nu', \underline{G}')}{(\nu, \mu \mathbf{t}.\underline{G}) \xrightarrow{\ell} (\nu', \underline{G}')} \lfloor \textsc{branch} \rfloor / \lfloor \textsc{rec} \rfloor$$

$$\frac{\forall k \in I \quad (\nu, \underline{G}_k) \xrightarrow{\ell} (\nu', \underline{G}'_k) \quad \mathtt{A}, \mathtt{B} \notin subj(\ell) \quad \ell \neq t}{(\nu, \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{T}_i\rangle\{A_i\}.\underline{G}_i\}_{i\in I}) \xrightarrow{\ell} (\nu', \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{T}_i\rangle\{A_i\}.\underline{G}'_i\}_{i\in I})} \lfloor \textsc{async1} \rfloor$$

$$\frac{(\nu, \underline{G}) \xrightarrow{\ell} (\nu', \underline{G}') \quad \mathtt{B} \notin subj(\ell)}{(\nu, \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{T}\rangle\{A\}.G) \xrightarrow{\ell} (\nu', \mathtt{A} \rightsquigarrow \mathtt{B} : \mathtt{a}\langle \mathtt{T}\rangle\{A\}.G')} \quad \frac{\nu + t \models^* \mathtt{rdy}(\underline{G})}{(\nu, \underline{G}) \xrightarrow{t} (\nu + t, \underline{G})} \lfloor \textsc{async2} \rfloor / \lfloor \textsc{time} \rfloor$$

**Fig. 25.** Labelled transitions for global types in the framework of timed MPSTs (adapted from [BYY14a])

## A. Correspondence between scribble and timed-MPST

The syntax of global types, in the framework of timed-MPSTs given in [BYY14a, BYY14b], is presented below:

$$\begin{array}{lll} \underline{G} & ::= & \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_i\langle \mathtt{T}_i\rangle\{A_i\}.\underline{G}_i\}_{i\in I} \mid \mu\, \mathbf{t}.\underline{G} \mid \mathbf{t} \mid \mathtt{end} \\ A & ::= & \{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\} \end{array}$$

The syntax of global types is very similar to the syntax of Scribble global protocols in Sect. 2.1 except: (1) Scribble does not cater for delegation and higher order protocols whereas Timed Global Session Types do; and (2) the choice and interaction protocols are two separated constructs in Scribble while they are modelled as a unique construct in Timed Global Session types. These differences, especially (2) and are consequence of the specific focus of Scribble as a protocol design language directed at partitioners that are familiar with e.g., Java notation, who proved to find this notation friendlier [SCR, HMB+11, YHNN13, HHN+14].

**Definition A.1** (*Encoding*) The encoding $E$ from (Scribble) global protocols to (timed-MPSTs) global types is given below:

$$\begin{array}{l} E([@\mathtt{A} : \delta_{\mathtt{O}}, \mathtt{reset}(\lambda_{\mathtt{O}})][@\mathtt{B} : \delta_{\mathtt{I}}, \mathtt{reset}(\lambda_{\mathtt{I}})]\ \mathtt{a}(\mathtt{T})\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\ G) = \\ \qquad \mathtt{A} \to \mathtt{B} : \{\mathtt{a}\langle \mathtt{T}\rangle\{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}.E(G)\} \\ E(\mathtt{choice\ at\ A}\ \{G^b_j\}_{j\in\{1,\ldots,n\}}) = \\ \qquad \mathtt{A} \to \mathtt{B} : \{\mathtt{a}_j\langle \mathtt{T}_j\rangle\{\delta_{\mathtt{O}j}, \lambda_{\mathtt{O}j}, \delta_{\mathtt{I}j}, \lambda_{\mathtt{I}j}\}.E(G_j)\}_{j\in\{1,\ldots,n\}} \\ \text{with (up to unfoldings) } G^b_j = [@\mathtt{A} : \delta_{\mathtt{O}j}, \mathtt{reset}(\lambda_{\mathtt{O}j})][@\mathtt{B} : \delta_{\mathtt{I}j}, \mathtt{reset}(\lambda_{\mathtt{I}j})]\ \mathtt{a}_j(\mathtt{T}_j)\ \mathtt{from}\ \mathtt{A}\ \mathtt{to}\ \mathtt{B};\ G_j \\ E(\mathtt{rec}\ \mathbf{t}\ \{G\}) = \mu\, \mathbf{t}.E(G) \\ E(\mathtt{continue}\ \mathbf{t}) = \mathbf{t} \\ E(\mathtt{end}) = \mathtt{end} \end{array}$$

For convenience we have recalled the semantics of global types in Fig. 25. The semantics of global protocols and global types are similar except that the one for timed-MPSTs from [BYY14a, BYY14b] have no rule $\lfloor \textsc{choice} \rfloor$ as choice is handled directly in the rule for send/selection and brach/receive. It is straightforward by induction on the proof that a Scribble timed global protocol $G$ is trace equivalent to its encoding $E(G)$.

**Lemma A.2** (*Correspondence-global*) Let $G$ be a Scribble timed global protocol, then $G \approx E(G)$.

Similarly we can define the following encodings:

- $E_{LTS}$ which is the encoding from (timed-MPSTs) local types (without delegation, noting that delegation is never introduced in the encoding from global protocols/types) to (Scribble) local protocols;
- $E_{LST}$ which is the encoding from (Scribble) local protocols to (timed-MPSTs) local types;
- $E_{CTS}$ which is the encoding between configurations of (timed-MPSTs) local types and configurations of (Scribble) local protocols.
- $E_{CST}$ which is the encoding between configurations of (Scribble) local protocols and (timed-MPSTs) local types.

It is straightforward by induction on the depth of the transition rule that a Scribble timed global protocol $G$ is trace equivalent to its encoding $E(G)$. Similarly since the semantics of local protocols/types and their corresponding configurations is identical, it is straightforward to prove the following correspondence.

**Lemma A.3** (*Correspondence*) Let $G$ be a Scribble timed global protocol, then $G \approx E(G)$. Similarly let $\underline{L}$ be a (timed-MPSTs) local type and $T$ be a (Scribble) local protocol. Then $\underline{L} \approx E_{LTS}(\underline{L})$; $T \approx E_{LTS}(T)$; $(\underline{L}_1, \ldots, \underline{L}_n, \vec{w}) \approx (E_{CTS}(\underline{L}_1), \ldots, E_{CTS}(\underline{L}_n), \vec{w})$; and $(T_1, \ldots, T_n, \vec{w}) \approx (E_{CST}(T_1), \ldots, E_{CST}(T_n), \vec{w})$.

## References


[AEY05] Alur R, Etessami K, Yannakakis M (2005) Realizability and verification of {MSC} graphs. Theor Comput Sci 331(1):97–114

[AFK87] Apt KR, Francez N, Katz S (1987) Appraising fairness in distributed languages. In: POPL, pp 189–198. ACM

[AGMK10] Akshay S, Gastin P, Mukund M, Narayan Kumar K (2010) Model checking time-constrained scenario-based specifications. In: FSTTCS, vol 8 of LIPIcs, pp 204–215

[AHJ15] Abdallah R, Hélouët L, Jard C (2015) Distributed implementation of message sequence charts. Softw Syst Model 14(2):1029–1048

[AMQ] Advanced Message Queuing protocols (AMQP) homepage. http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol.

[BCD+13] Bocchi L, Chen T-C, Demangeon R, Honda K, Yoshida N (2013) Monitoring networks through multiparty session types. In: FORTE, vol 7892 of LNCS, pp 50–65

[BDY12] Bocchi L, Demangeon R, Yoshida N (2012) A multiparty multi-session logic. In: TGC, vol 8191 of LNCS, Springer, Berlin, pp 97–111

[BFM98] Bowman H, Faconti GP, Massink M (1998) Specification and verification of media constraints using UPAAL. In: Design, specification and verification of interactive systems'98, proceedings of the fifth international eurographics workshop, 1998, Abingdon, Springer, UK, pp 261–277

[BHTY10] Bocchi L, Honda K, Tuosto E, Yoshida N (2010) A theory of design-by-contract for distributed multiparty interactions. In: CONCUR, vol 6269 of LNCS, pp 162–176

[BLY15] Bocchi L, Lange J, Yoshida N (2015) Meeting deadlines together. In: 26th International conference on concurrency theory, CONCUR 2015, Madrid, Spain, Sept 1.4, 2015, vol 42 of LIPIcs, pp 283–296. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

[BY07] Berger M, Yoshida N (2007) Timed, distributed, probabilistic, typed processes. In: APLAS, vol 4807 of LNCS, pp 158–174

[BYY14a] Bocchi L, Yang W, Yoshida N (2014) Timed multiparty session types. In: CONCUR, vol 8704 of LNCS, Springer, Berlin, pp 419–434

[BYY14b] Bocchi L, Yang W, Yoshida N (2014) Timed multiparty session types. Technical Report 2014/3, Department of Computing, Imperial College London

[C+11] Cambronero M-E et al (2011) Validation and verification of web services choreographies by using timed automata. J Log Algebr Program 80(1):25–49

[CDCYP15] Coppo M, Dezani-Ciancaglini M, Yoshida N, Padovani L (2015) Global progress for dynamically interleaved multiparty sessions. MSCS 760:1–65

[CKGJ13] Cheikhrouhou S, Kallel S, Guermouche N, Jmaiel M (2013) A survey on time-aware business process modeling. In: ICEIS (3), pp 236–242. SciTePress

[CPS09] Colombo C, Pace GJ, Schneider G (2009) Larva—safer monitoring of real-time java programs (tool paper). In: SEFM, pp 33–37

[CR07] Chen F, Rosu G (2007) Mop: an efficient and generic runtime verification framework. In: OOPSLA, pp 569–588

[dBdGJ+14] de Boer FS, de Gouw S, Johnsen EB, Kohn A, Wong PYH (2014) Run-time assertion checking of data- and protocol-oriented properties of Java programs: an industrial case study. Trans Aspect-Oriented Softw Dev 11:1–26

[DHH+15] Demangeon R, Honda K, Hu R, Neykova R, Yoshida N (2015) Practical interruptible conversations: Distributed dynamic verication with multiparty session types and python. FMSD, pp 1–29

[DY13] Deniélou P-M, Yoshida N (2013) Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In: Automata, languages, and programming—40th international colloquium, ICALP 2013, Riga, Latvia, July 8–12, 2013, Proceedings, Part II, volume 7966 of Lecture Notes in Computer Science, Springer, Berlin, pp 174–186

[GBE07] Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. SIGPLAN Not 42(10):57–76

[GDZ12] Guermouche N, Dal-Zilio S (2012) Towards timed requirement verification for service choreographies. In: CollaborateCom, pp 117–126. IEEE

[GMNK09] Gastin P, Mukund M, Kumar KN (2009) Reachability and boundedness in time-constrained MSC graphs. In: Lodaya K, Mukund M, Ramanujam R (eds) Perspectives in concurrency theory, pp 157–183. Universities Press

[GR+97] Gregorio-Rodrguez C et al (1997) Testing semantics for a probabilistic-timed process algebra. In: Transformation-based reactive systems development, vol 1231 of LNCS, pp 353–367

[HHN+14] Honda K, Hu R, Neykova R, Chen T-C, Demangeon R, Denilou P-M, Yoshida N (2014) Structuring communication with session types. In: COB 2014, vol 8665 of LNCS, Springer, Berlin, pp 105–127

[HJ00] Hlout L, Jard C (2000) Conditions for synthesis of communicating automata from hmscs. In: 5th International workshop on formal methods for industrial Cr itical systems (FMICS), Berlin, GMD FOKUS

[HMB+11] Honda K, Mukhamedov A, Brown G, Chen T-C, Yoshida N (2011) Scribbling interactions with a formal foundation. In: ICDCIT 2011, vol 6536 of LNCS. Springer, Berlin

[HNY+13] Hu R, Neykova R, Yoshida N, Demangeon R, Honda K (2013) Practical interruptible conversations: distributed dynamic verification with session types and python. In: RV, vol 8174 of LNCS, pp 130–148
[HY16] Hu R, Yoshida N (2016) Hybrid session verification through endpoint api generation. In: FASE 2016, LNCS. Springer, Berlin
[HYC08] Honda K, Yoshida N, Carbone M (2008) Multiparty asynchronous session types. In: POPL, pp 273–284. ACM
[Int98] International Telecommunication Union. Recommendation Z.120: Message sequence chart (1998)
[KCD+09] Kallel S, Charfi A, Dinkelaker T, Mezini M, Jmaiel M (2009) Specifying and monitoring temporal properties in web services compositions. In: Seventh IEEE European Conference on Web Services (ECOWS 2009), 9–11 Nov 2009, Eindhoven, The Netherlands, pp 148–157
[KY06a] Krcal P, Yi W (2006) Communicating timed automata: the more synchronous, the more difficult to verify. In: Computer aided verification, vol 4144 of LNCS, Springer, Berlin, pp 249–262
[KY06b] Krcal P, Yi W (2006) Communicating timed automata: the more synchronous, the more difficult to verify. In: CAV, vol 4144 of LNCS, pp 243–257
[LDD06] Liang H, Dingel J, Diskin Z (2006) A comparative survey of scenario-based to state-based model synthesis approaches. In: International workshop on scenarios and state machines: models, algorithms, and tools, SCESM '06. New York, pp 5–12. ACM
[Loh03] Lohrey M (2003) Realizability of high-level message sequence charts: closing the gaps. Theor Comput Sci 309(1):529–554
[LP12] López HA, Pérez JA (2012) Time and exceptional behavior in multiparty structured interactions. In: WS-FM, vol 7176 of LNCS, pp 48–63
[LPT07] Lapadula A, Pugliese R, Tiezzi F (2007) Cows: a timed service-oriented calculus. In: ICTAC, vol 4711 of LNCS, pp 275–290
[LZ02] Lee JY, Zic J (2002) On modeling real-time mobile processes. Aust Comput Sci Commun 24(1):139–147
[LZ05] Laneve C, Zavattaro G (2005) Foundations of web transactions. In: FOSSACS, vol 3411 of LNCS, pp 282–298
[MU00] Minsky NH, Ungureanu V (2000) Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. TOSEM 9:273–305
[NYH13] Neykova R, Yoshida N, Hu R (2013) SPY: local verification of global protocols. In: RV, vol 8174, Springer, Berlin, pp 358–363
[OOI] Ocean Observatories Initiative (OOI) http://oceanobservatories.org/
[pyt] Timed Conversation API in Python. http://www.doc.ic.ac.uk/~rn710/TimeApp.html
[SCR] Scribble Project homepage. www.scribble.org
[SG13] Saeedloei N, Gupta G (2013) Timed $\pi$-calculus. In: TGC, vol 8358 of LNCS. Springer, Berlin, pp 119–135
[Ski08] Skiena SS (2008) The algorithm design manual, 2nd edn. Springer, Berlin
[SMT] The Simple Mail Transfer Protocol. http://tools.ietf.org/html/rfc5321
[Tri99] Tripakis S (1999) Verifying progress in timed systems. In: Formal methods for real-time and probabilistic systems, vol 1601 of LNCS, Springer, Berlin, pp 299–314
[UPP] UPPAAL tool website. http://www.uppaal.org/
[WIH11] Kenji W, Ishikawa F, Hiraishi K (2011) Formal verification of business processes with temporal and resource constraints. In: SMC, pp 1173–1180. IEEE
[YDBH10] Yoshida N, Deniélou P-M, Bejleri A, Hu R (2010) Parameterised multiparty session types. In: FoSSaCs'10, vol 6014 of LNCS, Springer, Berlin, pp 128–145
[YHE02] Ye W, Heidemann J, Estrin D (2002) An energy-efficient mac protocol for wireless sensor networks. In: INFOCOM 2002, vol 3, pp 1567–1576. IEEE
[YHNN13] Yoshida N, Hu R, Neykova R, Ng N (2013) The scribble protocol language. In: TGC 2013, vol 8358 of LNCS, Springer, Berlin, pp 22–41
[Z3C] Z3 smt solver. http://z3.codeplex.com/

*Received 1 Mar 2015*
*Accepted in revised form 20 January 2017 by Thomas Hildebrandt, Joachim Parrow and Marco Carbone*
*Published online 22 February 2017*