From Communicating Machines to Graphical Choreographies

Julien Lange
Imperial College London, UK
j.lange@imperial.ac.uk

Emilio Tuosto
University of Leicester, UK
emilio@le.ac.uk

Nobuko Yoshida
Imperial College London, UK
n.yoshida@imperial.ac.uk

Abstract

Graphical choreographies, or global graphs, are general multiparty session specifications featuring expressive constructs such as forking, merging, and joining for representing application-level protocols. Global graphs can be directly translated into modelling notations such as BPMN and UML. This paper presents an algorithm whereby a global graph can be constructed from asynchronous interactions represented by communicating finite-state machines (CFSMs). Our results include: a sound and complete characterisation of a subset of safe CFSMs from which global graphs can be constructed; an algorithm to translate CFSMs to global graphs; a time complexity analysis; and an implementation of our theory, as well as an experimental evaluation.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Process Models

Keywords multiparty session types, choreography, communicating finite-state machines, global graphs, theory of regions

1. Introduction

Context Choreographies, models of interactions among software components from a global point of view, have been advocated as a conceptual and practical tool to tackle the complexity of designing, analysing, and implementing modern applications (see e.g., [3, 12, 18, 27]). As noted in [18], besides yielding a global perspective of the coordination of applications supporting the development and verification of single components, a global specification can also be projected so to obtain the local behaviour of components. The software engineering methodology associated with choreographies is usually a uni-directional (top-down) approach to software development life cycle (SDLC). Such a methodology appeals to industry [3, 4, 18] since it allows developers to check components against the corresponding projections of the choreography. However, choreography-based approaches do not fully support SDLC. For example, the 'conform direction' of *testable architectures* [3] lacks algorithms to obtain global models when modifying local projections.

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676964

To address this limitation we propose an algorithm to construct choreographies from a set of behavioural specifications of components interacting through asynchronous message passing. We consider the following two scenarios to motivate the practical applicability of our algorithm.

- Distributed service architectures envisage software as a provision made available (through a public interface that hides implementation details) to be dynamically searched by and composed. The choreography of such systems cannot therefore be designed in advance and has to be established and checked at binding-time to attain automatic composition.
- A frequent problem practitioners have to face is the integration of newly developed software with legacy code. Typically, the latter often do not come with a global specification and changes with time. Therefore, it is difficult to assess how modifications to newly developed components fit within the system.

Relying on a modelling notations used in industry, our algorithm enables a bi-directional (top-down and bottom-up) choreography-driven SDLC: a developer can visualise a global viewpoint; thus, when an unexpected choreography emerges, either existing components or the global specification may be refined. Modified choreographies can be projected again so to be compared with the original projections.

Our approach We adopt *communicating finite-state machines* (CFSMs) as suitable behavioural specifications of distributed components from which a choreography can be built. CFSMs are a conceptually simple model, based on asynchronous FIFO message-passing communication, and are well-established for analysing properties of distributed systems. They are also widely used in industry tools and can be seen as end-point specifications.

We define an algorithm that, given a set of CFSMs, yields a choreography expressed as a *global graph* [20], a graphical model closely related to BPMN 2.0 Choreography, advocated as a suitable notation for services [1]. The system S_{re} in Figure 1 will be the running example to illustrate our approach; S_{re} consists of four CFSMs, each having *three* buffers to communicate with the other participants, that realise a protocol of a fictive game where:

1. `Alice` (`A`) sends either *bwin* to `Bob` (`B`) or *cwin* to `Carol` (`C`) to decide who wins the game. In the former case, `A` fires the transition `AB`!*bwin* whereby the message *bwin* is put in the FIFO buffer `AB` from `A` to `B`, and likewise in the latter case.
2. If `B` wins (that is the message *bwin* is on top of the queue `AB` and `B` consumes it by taking the transition `AB`?*bwin*), then he sends a notification (*close*) to `C` to notify her that she has lost. Symmetrically, `C` notifies `B` of her victory (*blose*).
3. During the game, `C` notifies `Dave` (`D`) that she is *busy*.
4. After `B` and `C` have been notified of the outcome of the game, `B` sends a signal (*sig*) to `A`, while `C` sends a message (*msg*) to `A`.

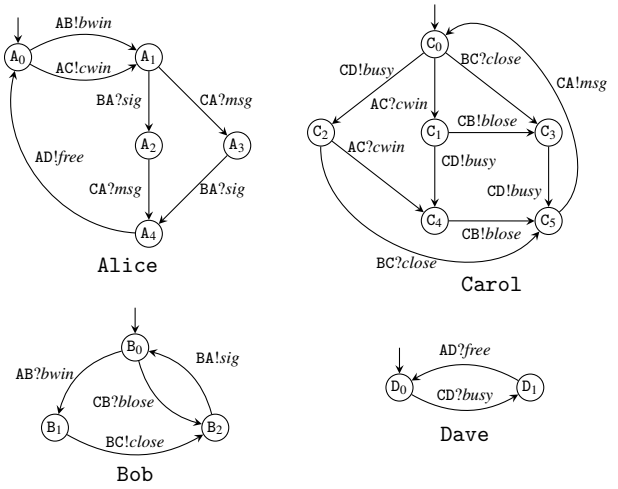


Figure 1. Communicating System S_{re}

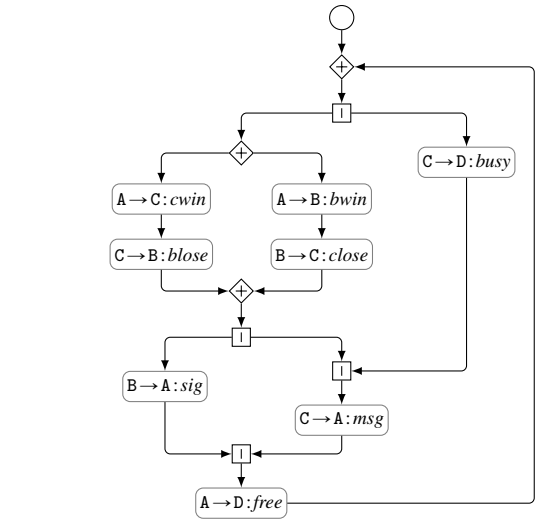


Figure 2. Global Graph G_{re}

5. Once the result is sent, A notifies D that C is now *free* and a new round starts.

The underlying protocol of S_{re} shows that CFSMs capture many coordination constructs: in 1, A (non-deterministically) chooses the winner; in 2, B has a sequential behaviour; in 3, the parallel behaviour of C is rendered with the interleaving of transition C!*busy*; in 4 and 5, threads join and finally the protocol loops.

Understanding the global model of S_{re} is not easy. A much clearer specification is given by the global graph G_{re} (constructed by our algorithm) in Figure 2. There, the choreography of the four components is explicit and it is possible to identify sequentially ordered, independent, or exclusive interactions. For instance, from G_{re}, it is evident that interaction A→B:*bwin* must precede B→C:*close*, while interaction C→D:*busy* is independent of the former two. On the other hand, A→B:*bwin* and A→C:*cwin* are exclusive, i.e., only one of them may be executed in each round of the game.

Establishing properties of CFSMs such as

> *Is S_{re} deadlock-free? will any sent message be eventually consumed? will each participant eventually receive any message s/he is waiting for?*

is generally undecidable [14] or computationally hard, and not immediate even for the simple scenario in Figure 1. We give a decidable condition, called *generalised multiparty compatibility* (GMC) that characterises a set of systems for which the questions above can be decided. Our algorithm can produce a global graph from any set of generalised multiparty compatible CFSMs. The global graph is constructed through a transformation of the CFSMs into a safe Petri net, using the algorithm in [19]. The transformation preserves the original CFSMs, which can be recovered by projecting the global graph. Noteworthy, most of the systems we found in the literature enjoy GMC and very few of them do not (cf. § 5).

Contributions To the best of our knowledge, this is the first work to build graphical choreographies from CFSMs and to characterise the set of CFSMs from which such choreographies can be built. Our theory is supported by a tool (which we evaluated against protocols from the literature).

Recently the construction of syntactic (non-graphical) multiparty session types [24] from local specifications has been studied in [21, 25] for a less general framework with no support for *local concurrency*; for instance in Figure 1, C can send message *busy* while concurrently receiving either *cwin* or *close* (similarly A can execute input actions CA?*msg* and BA?*sig* in parallel). We argue that catering for a general form of local concurrency (which is in fact supported by threads in many programming languages) is crucial for modelling real-world systems.

In [11, 12] conditions for communicating systems to be safe are given; however, they do not address the problem of constructing choreographies and consider a form of local concurrency more restrictive than ours due to a *single* receiving buffer per participant. We use two uni-directional queues for each couple of participants so that a component can concurrently communicate with many other components accessing different FIFO queues (as, e.g., supported in the TCP protocol suite).

Synopsis § 2 reviews CFSMs. § 3 defines generalised multiparty compatibility, analyses its complexity (Proposition 3.1 and Proposition 3.2), and its soundness (Theorem 3.1). § 3.3 discusses how our condition can be used to suggest amendments to fix non-GMC systems. The construction algorithm, its complexity (Proposition 4.1), and its completeness (Theorem 4.1) are in § 4. The tool and experimental evaluation are in § 5. We conclude and discuss future work in § 7, after discussing more related work in § 6. The full version of this paper [5] include full proofs of our results and benchmark protocols; our tool is available online [6].

2. Communicating Finite-State Machines

This section reviews definitions and properties of CFSMs. Throughout the paper we use the following sets and notations. Fix a finite set $\mathcal{P}$ of *participants* (ranged over by p, q, r, s, etc.) and a finite alphabet $\mathbb{A}$. The set of *channels* is $C \stackrel{\text{def}}{=} \{\mathtt{pq} \mid \mathtt{p},\mathtt{q} \in \mathcal{P} \text{ and } \mathtt{p} \neq \mathtt{q}\}$ while $Act \stackrel{\text{def}}{=} C \times \{!,?\} \times \mathbb{A}$ is the set of *actions* (ranged over by ℓ), $\mathbb{A}^*$ (resp. Act^*, ranged over by φ) is the set of finite words on $\mathbb{A}$ (resp. Act). Also, ε ($\notin \mathbb{A} \cup Act$) is the empty word, $|\varphi|$ denotes the length of φ, and $\varphi\varphi'$ is the concatenation of φ and φ' (we overload these notations for words over $\mathbb{A}$).

Definition 2.1 (CFSM)**.** A *communicating finite-state machine* is a finite transition system given by a 4-tuple $M = (Q, q_0, \mathbb{A}, \delta)$ where Q is a finite set of *states*, $q_0 \in Q$ is the initial state, and $\delta \subseteq Q \times Act \times Q$ is a set of *transitions*. ◇

The transitions of a CFSM are labelled by actions; label $\mathtt{sr}!a$ represents the *sending* of message a from machine $\mathtt{s}$ to $\mathtt{r}$ and, dually, $\mathtt{sr}?a$ represents the *reception* of a by $\mathtt{r}$. We write $\mathcal{L}(M) \subseteq Act^*$ for the language on Act accepted by the automaton corresponding to machine M where each state of M is an accepting state. A state $q \in Q$ with no outgoing transition is *final*; q is a *sending* (resp. *receiving*) state if all its outgoing transitions are labelled with sending (resp. receiving) actions, and q is a *mixed* state otherwise.

A CFSM $M = (Q, q_0, \mathbb{A}, \delta)$ is *deterministic* if for all states $q \in Q$ and all actions $\ell \in Act$, if $(q,\ell,q'),(q,\ell,q'') \in \delta$ then $q' = q''$.[1] A CFSM M is *minimal* if there is no machine M' with fewer states and transitions than M such that $\mathcal{L}(M) = \mathcal{L}(M')$. Hereafter, we only consider deterministic and minimal CFSMs.

Definition 2.2 (Communicating systems)**.** Given a CFSM $M_\mathtt{p} = (Q_\mathtt{p}, q_{0\mathtt{p}}, \mathbb{A}, \delta_\mathtt{p})$ for each $\mathtt{p} \in \mathcal{P}$, the tuple $S = (M_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$ is a *communicating system* (*CS*). A *configuration* of S is a pair $s = (\vec{q};\vec{w})$ where $\vec{q} = (q_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$ with $q_\mathtt{p} \in Q_\mathtt{p}$ and where $\vec{w} = (w_\mathtt{pq})_{\mathtt{pq}\in C}$ with $w_\mathtt{pq} \in \mathbb{A}^*$; component $\vec{q}$ is the *control state* and $q_\mathtt{p} \in Q_\mathtt{p}$ is the *local state* of machine $M_\mathtt{p}$. The *initial configuration* of S is $s_0 = (\vec{q}_0;\vec{\varepsilon})$ with $\vec{q}_0 = (q_{0\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$. ◇

Hereafter, we fix a machine $M_\mathtt{p} = (Q_\mathtt{p}, q_{0\mathtt{p}}, \mathbb{A}, \delta_\mathtt{p})$ for each participant $\mathtt{p} \in \mathcal{P}$ and let $S = (M_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$ be the corresponding system.

Definition 2.3 (Reachable states and configurations)**.** A configuration $s' = (\vec{q}';\vec{w}')$ is *reachable* from another configuration $s = (\vec{q};\vec{w})$ by *firing transition* ℓ, written $s \xrightarrow{\ell} s'$ (or $s \to s'$ if the label is immaterial), if there is $a \in \mathbb{A}$ such that either:

1. $\ell = \mathtt{sr}!a$ and $(q_\mathtt{s}, \ell, q'_\mathtt{s}) \in \delta_\mathtt{s}$ and
   (a) $q'_\mathtt{p} = q_\mathtt{p}$ for all $\mathtt{p} \neq \mathtt{s}$, and
   (b) $w'_\mathtt{sr} = w_\mathtt{sr}.a$ and $w'_\mathtt{pq} = w_\mathtt{pq}$ for all $\mathtt{pq} \neq \mathtt{sr}$; or
2. $\ell = \mathtt{sr}?a$ and $(q_\mathtt{r}, \ell, q'_\mathtt{r}) \in \delta_\mathtt{r}$ and
   (a) $q'_\mathtt{p} = q_\mathtt{p}$ for all $\mathtt{p} \neq \mathtt{r}$, and
   (b) $w_\mathtt{sr} = a.w'_\mathtt{sr}$ and $w'_\mathtt{pq} = w_\mathtt{pq}$ for all $\mathtt{pq} \neq \mathtt{sr}$.

The reflexive and transitive closure of $\to$ is $\to^*$. We write $s_1 \xrightarrow{\ell_1\cdots\ell_m} s_{m+1}$ when, for some $s_2,\ldots,s_m$, $s_1 \xrightarrow{\ell_1} s_2 \cdots s_m \xrightarrow{\ell_m} s_{m+1}$. A sequence of transitions is *k-bounded* if no channel of any intermediate configuration on the sequence contains more than k messages. The set of *reachable configurations of* S is $RS(S) = \{s \mid s_0 \to^* s\}$. The *k-reachability set of* S is the largest subset $RS_k(S)$ of $RS(S)$ within which each configuration s can be reached by a k-bounded execution from s_0. ◇

Condition (1b) in Definition 2.3 puts a on channel $\mathtt{sr}$, while (2b) gets a from channel $\mathtt{sr}$. Note that, for every integer k, the set $RS_k(S)$ is finite and computable.

We now recall several definitions about communicating systems S and their configurations $s = (\vec{q};\vec{w})$. We say that s is a *deadlock configuration* [17, Def. 12] if $\vec{w} = \vec{\varepsilon}$, there is $\mathtt{r} \in \mathcal{P}$ such that $(q_\mathtt{r}, \mathtt{sr}?a, q'_\mathtt{r}) \in \delta_\mathtt{r}$, and for every $\mathtt{p} \in \mathcal{P}$, $q_\mathtt{p}$ is a receiving or final state, i.e., all the buffers are empty, there is at least one machine waiting for a message, and all the other machines are either in a final or receiving state. Configuration s is an *orphan message configuration* if all $q_\mathtt{p} \in \vec{q}$ are final but $\vec{w} \neq \vec{\varepsilon}$, i.e., there is at least a non-empty buffer and each machine is in a final state. Finally, s is an *unspecified reception configuration* [17, Def. 12] if there exists $\mathtt{r} \in \mathcal{P}$ such that $q_\mathtt{r}$ is a receiving state, and $(q_\mathtt{r}, \mathtt{sr}?a, q'_\mathtt{r}) \in \delta_\mathtt{r}$ implies that $|w_\mathtt{sr}| > 0$ and $w_\mathtt{sr} \notin a\mathbb{A}^*$, i.e., $q_\mathtt{r}$ is prevented from receiving any message from any of its buffers.

[1] Sometimes, a CFSM is considered deterministic when $(q,\mathtt{sr}!a,q') \in \delta$ and $(q,\mathtt{sr}!a',q'') \in \delta$ then $a = a'$ and $q' = q''$. Here, we follow a different definition [17] in order to represent branching type constructs.

Definition 2.4 (Safe CS)**.** System S is *safe* if for each $s \in RS(S)$, s is not a deadlock, an orphan message, nor an unspecified reception configuration. ◇

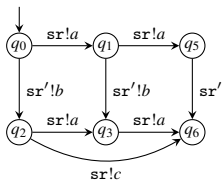
The following definitions are new and instrumental for § 3 where we characterise a subset of safe CS from which a global graph can be constructed. A key point to give our condition for a CS to be safe is to identify sets of concurrent actions. Below, we define an equivalence relation on transitions of a CFSM. Given $q,q' \in Q$, let $act(q,q') \stackrel{\text{def}}{=} \{\ell \mid (q,\ell,q') \in \delta\}$ and define $\diamond, \blacklozenge \subseteq \delta \times \delta$ as the smallest equivalence relations that respectively contain the relations $\underline{\diamond}$ and $\underline{\blacklozenge}$ where

- $(q_1,\ell,q_2)\underline{\diamond}(q'_1,\ell,q'_2)$ iff $\ell \notin act(q_1,q'_1) = act(q_2,q'_2) \neq \varnothing$
- $(q_1,\ell,q_2)\underline{\blacklozenge}(q'_1,\ell,q'_2)$ iff $(q_1,\ell,q_2)\underline{\diamond}(q'_1,\ell,q'_2)$ and for all $(q,\ell,q') \in [(q_1,\ell,q_2)]^\diamond$, $act(q_1,q) = act(q_2,q') \wedge act(q'_1,q) = act(q'_2,q')$

where $[(q,\ell,q')]^\diamond$ denotes the equivalence class of (q,ℓ,q') wrt $\diamond$.

Intuitively, two transitions are $\blacklozenge$-related if they refer to the same action up-to interleaving.

Example 2.1. Consider the CFSM below.



$$(q_0,\mathtt{sr}!a,q_1)\underline{\diamond}(q_2,\mathtt{sr}!a,q_3) \quad (1)$$
$$(q_0,\mathtt{sr}!a,q_1)\blacklozenge(q_2,\mathtt{sr}!a,q_3) \quad (2)$$
$$\neg((q_0,\mathtt{sr}!a,q_1)\underline{\diamond}(q_1,\mathtt{sr}!a,q_5)) \quad (3)$$
$$(q_0,\mathtt{sr}'!b,q_2)\underline{\diamond}(q_1,\mathtt{sr}'!b,q_3) \quad (4)$$
$$\neg((q_0,\mathtt{sr}'!b,q_2)\blacklozenge(q_1,\mathtt{sr}'!b,q_3)) \quad (5)$$

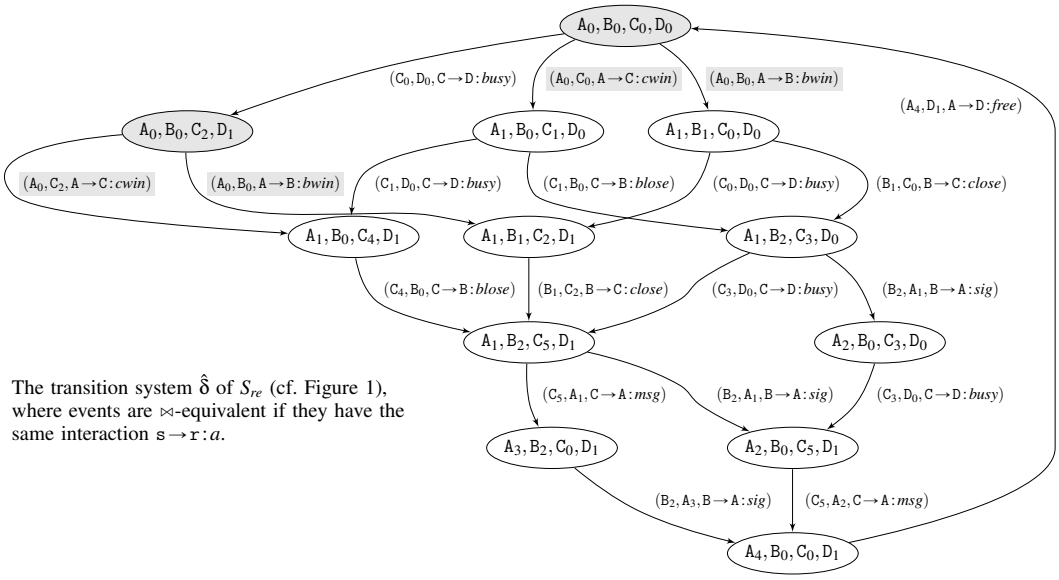
The relations in (1-2) hold since both transitions are interleaved with $\mathtt{sr}'!b$. The relation in (3) does not hold since the transition between the source of one (q_0) and the source of the other (q_1) passes through $\mathtt{sr}!a$ itself. The two transitions in (3) are *sequential* rather than concurrent. The relation in (4) holds, but the relation in (5) does not because there is $(q_5,\mathtt{sr}'!b,q_6)$ in the $\diamond$-equivalence classes of $(q_0,\mathtt{sr}'!b,q_2)$ for which the condition does not hold (due to the transition with label $\mathtt{sr}!c$).

In Figure 1, $(\mathtt{C}_0,\mathtt{AC}?cwin,\mathtt{C}_1)\blacklozenge(\mathtt{C}_2,\mathtt{AC}?cwin,\mathtt{C}_4)$ since both transitions represent the same action interleaved with $\mathtt{CD}!busy$. In each machine in Figure 1, a set of transitions (q,ℓ,q') with the same label ℓ forms a $\blacklozenge$-equivalence class, e.g., in $\mathtt{Alice}$, $\{(\mathtt{A}_1,\mathtt{CA}?msg,\mathtt{A}_3),\ (\mathtt{A}_2,\mathtt{CA}?msg,\mathtt{A}_4)\}$ is a $\blacklozenge$-equivalence class labelled by $\mathtt{CA}?msg$.

3. CFSMs Characterisation of Global Graphs

3.1 Synchronous transition system

Systems amenable to be transformed into global graphs are identified through *their synchronous transition system* (cf. Definition 3.2) where nodes consist of a vector of local states and transitions are labelled by elements in the set of *events* $\mathcal{E} \stackrel{\text{def}}{=} \bigcup_{\mathtt{s},\mathtt{r}\in\mathcal{P}} Q_\mathtt{s} \times Q_\mathtt{r} \times \{(\mathtt{s},\mathtt{r})\} \times \mathbb{A}\}$. Intuitively, an event $(q_\mathtt{s},q_\mathtt{r},\mathtt{s},\mathtt{r},a) \in \mathcal{E}$, written $(q_\mathtt{s},q_\mathtt{r},\mathtt{s}\to\mathtt{r}:a)$ for short, indicates that machines $\mathtt{s}$ and $\mathtt{r}$ can exchange message a when they are respectively in state $q_\mathtt{s}$ and $q_\mathtt{r}$. Indexing events with the local states of the machines permits to distinguish two occurrences of the same communication at two different points in a global graph. To single out parallelism at the machine level, we introduce an equivalence relation over events that identifies events whose underlying local transitions are $\blacklozenge$-equivalent.



The transition system $\hat{\delta}$ of S_{re} (cf. Figure 1), where events are $\bowtie$-equivalent if they have the same interaction $\mathtt{s} \to \mathtt{r} : a$.

Figure 3. Transition graph of $\hat{\delta}$ and $TS(S_{re})$

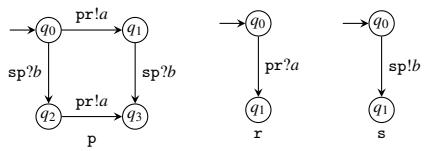
Definition 3.1 ($\mathcal{E}$-equivalence)**.** The *event equivalence* is the relation $\bowtie \stackrel{\text{def}}{=} \bowtie_{\mathtt{s}} \cap \bowtie_{\mathtt{r}} \subseteq \mathcal{E} \times \mathcal{E}$ where

$$(q_1, q_2, \mathtt{s} \to \mathtt{r} : a) \bowtie_{\mathtt{s}} (q_1', q_2', \mathtt{s} \to \mathtt{r} : a) \iff \forall (q_1, \mathtt{sr}!a, q_3), (q_1', \mathtt{sr}!a, q_3') \in \delta_{\mathtt{s}} : \ (q_1, \mathtt{sr}!a, q_3) \blacklozenge (q_1', \mathtt{sr}!a, q_3')$$

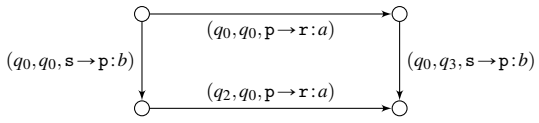
$$(q_1, q_2, \mathtt{s} \to \mathtt{r} : a) \bowtie_{\mathtt{r}} (q_1', q_2', \mathtt{s} \to \mathtt{r} : a) \iff \forall (q_2, \mathtt{sr}?a, q_4), (q_2', \mathtt{sr}?a, q_4') \in \delta_{\mathtt{r}} : \ (q_2, \mathtt{sr}?a, q_4) \blacklozenge (q_2', \mathtt{sr}?a, q_4')$$

We let $[e]$ denote the $\bowtie$-equivalence class of event e. ◇

Example 3.1. Consider the communicating system below:



Its synchronous transition system (cf. Definition 3.2 below) is the labelled transition system:



We have $(q_0, q_0, \mathtt{p} \to \mathtt{r} : a) \bowtie (q_2, q_0, \mathtt{p} \to \mathtt{r} : a)$ and $(q_0, q_0, \mathtt{s} \to \mathtt{p} : b) \bowtie (q_0, q_3, \mathtt{s} \to \mathtt{p} : b)$. Considering equivalent the events on the "vertical" transitions and those on the "horizontal" ones equivalent allows us to identify a pair of concurrent interactions; while still differentiating them from other occurences of communications $\mathtt{p} \to \mathtt{r} : a$ and $\mathtt{s} \to \mathtt{p} : b$.

In our running example (cf. Figure 1), we have $(\mathtt{C}_5, \mathtt{A}_2, \mathtt{C} \to \mathtt{A} : msg) \bowtie (\mathtt{C}_5, \mathtt{A}_1, \mathtt{C} \to \mathtt{A} : msg)$ since the underlying transitions of $\mathtt{A}$ are $\blacklozenge$-equivalent, i.e., $(\mathtt{A}_1, \mathtt{CA}?msg, \mathtt{A}_3) \blacklozenge (\mathtt{A}_2, \mathtt{CA}?msg, \mathtt{A}_4)$, and the underlying transition of $\mathtt{C}$ is the same for both events, i.e., $(\mathtt{C}_5, \mathtt{CA}!msg, \mathtt{C}_0)$.

Hereafter, we let $n, n', \ldots$ denote vectors of local states and $n[\mathtt{p}]$ denote the state of $\mathtt{p} \in \mathcal{P}$ in n.

Definition 3.2 (Synchronous transition system)**.** Given a system $S = (M_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$, let $N \stackrel{\text{def}}{=} \{\vec{q} \mid (\vec{q}; \vec{\varepsilon}) \in RS_1(S)\}$,

$$\hat{\delta} \stackrel{\text{def}}{=} \left\{ (n, e, n') \mid (n; \vec{\varepsilon}) \xrightarrow{\mathtt{sr}!a} \xrightarrow{\mathtt{sr}?a} (n'; \vec{\varepsilon}) \wedge e = (n[\mathtt{s}], n[\mathtt{r}], \mathtt{s} \to \mathtt{r} : a) \right\}$$

and $E \stackrel{\text{def}}{=} \{e \mid \exists n, n' \in N : \ (n, e, n') \in \hat{\delta}\} \subseteq \mathcal{E}$.
The *synchronous transition system of* S is $TS(S) = (N, n_0, E/\bowtie, \Rightarrow)$ where $n_0 = \vec{q}_0$ is the initial state, and $n \stackrel{[e]}{\Rightarrow} n' \iff (n, e, n') \in \hat{\delta}$. We fix a set $\hat{E}$ of representative elements of each $\bowtie$-equivalence class (i.e., $\hat{E} \subseteq E$ and $\forall e \in E \ \exists! e' \in \hat{E} : \ e' \in [e]$) and write $n \stackrel{e'}{\Rightarrow} n'$ for $n \stackrel{[e]}{\Rightarrow} n'$ when $e' \in [e] \cap \hat{E}$. Sequences of events are ranged over by π and we extend the notation on $\to$ in Definition 2.3 to $\Rightarrow$ (e.g., if $\pi = e_1 \cdots e_k$, $n_1 \stackrel{\pi}{\Rightarrow} n_{k+1}$ iff $n_1 \stackrel{e_1}{\Rightarrow} n_2 \stackrel{e_2}{\Rightarrow} \cdots \stackrel{e_k}{\Rightarrow} n_{k+1}$). ◇

$TS(S)$ represents all the possible synchronous executions of system S; and each transition is labelled by an event e, taken up-to $\blacklozenge$-equivalence so to distinguish different occurrences of a same communication, while preserving the parallelism of local machines. The synchronous transition system for our running example is given in Figure 3.

Definition 3.3 (Projections)**.** The projection of an event e onto participant $\mathtt{p}$, denoted by $e\!\upharpoonright_{\mathtt{p}}$, is defined as follows:

$$(q_{\mathtt{s}}, q_{\mathtt{r}}, \mathtt{s} \to \mathtt{r} : a)\!\upharpoonright_{\mathtt{p}} \stackrel{\text{def}}{=} \begin{cases} \mathtt{pr}!a & \text{if } \mathtt{s} = \mathtt{p} \\ \mathtt{sp}?a & \text{if } \mathtt{r} = \mathtt{p} \\ \varepsilon & \text{otherwise} \end{cases}$$

Projection is defined on sequences of events in the obvious way. The projection of $TS(S) = (N, n_0, \hat{E}, \Rightarrow)$ on participant $\mathtt{p}$, written $TS(S)\!\upharpoonright_{\mathtt{p}}$, is the automaton $(Q, q_0, \mathbb{A}, \delta)$ where $Q = N$, $q_0 = n_0$, and $\delta \subseteq Q \times Act \cup \{\varepsilon\} \times Q$ is s.t. $(n_1, e\!\upharpoonright_{\mathtt{p}}, n_2) \in \delta \iff n_1 \stackrel{e}{\Rightarrow} n_2$. ◇

3.2 Generalised multiparty compatibility

We introduce *generalised multiparty compatibility* (GMC) as a sound and complete condition for constructing global graphs. Hereafter, we fix a system $S = (M_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$ with $TS(S) = (N, n_0, \hat{E}, \Rightarrow)$. Essentially, GMC relies on two conditions, (1) *representability* (cf.

Definition 3.4): for each machine, each trace and each choice are represented in $TS(S)$; and (2) *branching property* (Definition 3.5): whenever there is a choice in $TS(S)$, a unique machine takes the decision and each of the other participants is either made aware of which branch was chosen or not involved in the choice. Representability guarantees that $TS(S)$ contains enough information to decide safety properties of any (asynchronous) execution of S; and the branching property ensures that, if a branching in $TS(S)$ represents a choice, then this choice is "well-formed".

For a language $\mathcal{L}$, $hd(\mathcal{L})$ returns the first actions of $\mathcal{L}$ (if any).

$$hd(\mathcal{L}) \stackrel{\text{def}}{=} \{\ell \mid \exists \varphi \in Act^* : \ \ell \cdot \varphi \in \mathcal{L}\} \qquad hd(\{\varepsilon\}) \stackrel{\text{def}}{=} \{\varepsilon\}$$

Given $n \in N$, let $TS(S)\langle n\rangle$ be the transition system $TS(S)$ where the initial state n_0 is replaced by n. We write $LT(S,n,\mathtt{p})$ for $\mathcal{L}(TS(S)\langle n\rangle\!\downarrow_{\mathtt{p}})$; that is $LT(S,n,\mathtt{p})$ is the language obtained by setting the initial node of $TS(S)$ to n and then projecting this new transition system onto $\mathtt{p}$.

Definition 3.4 (Representability)**.** System S is *representable* if

1. $\mathcal{L}(M_{\mathtt{p}}) = LT(S,n_0,\mathtt{p})$ and
2. $\forall q \in Q_{\mathtt{p}}\ \exists n \in N :\ n[\mathtt{p}] = q \wedge \bigcup_{(q,\ell,q')\in\delta_{\mathtt{p}}} \{\ell\} \subseteq hd(LT(S,n,\mathtt{p}))$.

for all $\mathtt{p} \in \mathcal{P}$. ◇

Condition (1) in Definition 3.4 is needed to ensure that each trace of each machine is represented in $TS(S)$; while condition (2) is necessary to ensure that every choice in each machine is represented in $TS(S)$.

Proposition 3.1. *Given a system $S = (M_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$, checking whether S satisfies the representability condition is computable in*

$$O(\sum_{\mathtt{p}\in\mathcal{P}} 2^{|N|+|Q_{\mathtt{p}}|}) \text{ time, with } |N| = \prod_{\mathtt{p}\in\mathcal{P}} |Q_{\mathtt{p}}|$$

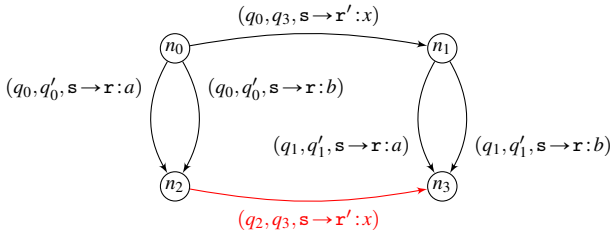
In the worst case, the time complexity of checking the representability of S is exponential. This is solely due to the language equivalence check (condition (1) in Definition 3.4) between each machine and its projection from $TS(S)$. However, as observed in [13], in practice algorithms for language equivalence behave very efficiently. In addition, we can remove some states from the projection of $TS(S)$, e.g., those that are on chains of ε-transitions only, while preserving its language, thus reducing the exponent $|N|$.

We give a few auxiliary definitions before formalising the branching property. For $n \neq n' \in N$, we define $n \prec n'$ iff $n \Rightarrow^* n'$ and for all paths $n_0 \Rightarrow n_1 \Rightarrow \ldots \Rightarrow n_{k-1} \Rightarrow n_k = n$ in $TS(S)$ such that $n_0,\ldots,n_k$ are pairwise distinct, $n' \neq n_h$ for all $0 \leqslant h \leqslant k$. Intuitively, $n \prec n'$ holds if n' is reachable from n and no simple path from n_0 to n goes through n'; note that $\prec$ is not a preorder in general. The *last nodes* reachable from $n \in N$ with $e_1 \neq e_2 \in \hat{E}$ are

$$ln(n,e_1,e_2) \stackrel{\text{def}}{=} \left\{ (n_1,n_2) \;\middle|\; \begin{array}{l} \exists n' \in N : \forall i \in \{1,2\} : n \Rightarrow^* n' \stackrel{e_i}{\Rightarrow} n_i \\ \quad \wedge\ \forall n'' \in N : n' \Rightarrow n'' \\ \qquad \Longrightarrow\ \forall j \in \{1,2\} :\ \neg(n' \prec n'' \stackrel{e_j}{\Rightarrow}) \end{array} \right\}$$

If $(n_1,n_2) \in ln(n,e_1,e_2)$, then n_i is a $\stackrel{e_i}{\Rightarrow}$-successor ($i = 1,2$) of a node n' on a path from n whose successors are either not able to fire both e_1 and e_2 or not $\prec$-related to n'.

Example 3.2. Consider the synchronous transition system below.



If $q_0 = q_1$ and $q_0' = q_1'$, we have $ln(n_0,(q_0,q_0',\mathtt{s}\to\mathtt{r}:a),(q_0,q_0',\mathtt{s}\to\mathtt{r}:b)) = \{(n_3,n_3)\}$. In this case, both branches on a and b from nodes n_0 and n_1 are considered equivalent (they are only interleaved with the exchange of message x). However, if the edge from n_2 to n_3 is removed and $q_0 \neq q_1$ and $q_0' \neq q_1'$, then $ln(n_0,(q_0,q_0',\mathtt{s}\to\mathtt{r}:a),(q_0,q_0',\mathtt{s}\to\mathtt{r}:b)) = \{(n_2,n_2)\}$. In this case the two branches are not equivalent since one of them prevents x to be ever exchanged.

In our running example (cf. Figure 3), we have:

$$ln((\mathtt{A}_0,\mathtt{B}_0,\mathtt{C}_0,\mathtt{D}_0),(\mathtt{A}_0,\mathtt{B}_0,\mathtt{A}\to\mathtt{B}:bwin),(\mathtt{A}_0,\mathtt{C}_0,\mathtt{A}\to\mathtt{C}:cwin)) = \{((\mathtt{A}_1,\mathtt{B}_1,\mathtt{C}_2,\mathtt{D}_1),(\mathtt{A}_1,\mathtt{B}_0,\mathtt{C}_4,\mathtt{D}_1))\}$$

Recall that $(\mathtt{A}_0,\mathtt{C}_2,\mathtt{A}\to\mathtt{C}:cwin) \bowtie (\mathtt{A}_0,\mathtt{C}_0,\mathtt{A}\to\mathtt{C}:cwin)$; i.e., the pair of events can be fired from both $(\mathtt{A}_0,\mathtt{B}_0,\mathtt{C}_0,\mathtt{D}_0)$ and $(\mathtt{A}_0,\mathtt{B}_0,\mathtt{C}_2,\mathtt{D}_1)$.

For an event $e = (q_{\mathtt{s}},q_{\mathtt{r}},\mathtt{s}\to\mathtt{r}:a) \in \mathcal{E}$, let $\iota(e) = \mathtt{s}\to\mathtt{r}:a$ and define a dependency relation $\lhd \subseteq \mathcal{E}\times\mathcal{E}$ on events:

$$e \lhd e' \iff \iota(e) = \mathtt{s}\to\mathtt{r}:a \wedge (\iota(e') = \mathtt{s}\to\mathtt{r}:a' \vee \iota(e') = \mathtt{r}\to\mathtt{r}':a')$$

Intuitively, e and e' are $\lhd$-related if there exists a dependency relation between the two interactions, from the point of view of the receiver. We define a relation $e \blacktriangleleft e'$ in π if there is a $\lhd$-relation between e and e' in π, i.e.,

$$e \blacktriangleleft e' \text{ in } \pi \iff \begin{cases} (e \lhd e'' \wedge e'' \blacktriangleleft e' \text{ in } \pi') \vee e \blacktriangleleft e' \text{ in } \pi' & \text{if } \pi = e''\cdot\pi' \\ e \lhd e' & \text{otherwise} \end{cases}$$

also, $dep(\iota(e),\pi,\iota(e'))$ iff

$$\big(\pi = \pi_1\cdot e\cdot\pi_2\cdot e'\cdot\pi' \ \wedge\ (\_,\_,\iota(e)) \notin \pi_1 \ \wedge\ (\_,\_,\iota(e')) \notin \pi_2\big) \Longrightarrow\ e \blacktriangleleft e' \text{ in } \pi_2$$

which checks whether there is a dependency between two interactions on a path π (if these interactions do appear in π). Below we give the second condition for GMC, which ensures that each "global choice" is made by exactly one participant and that all the other participants are either made aware of the choice or not involved in/affected by the choice.

Definition 3.5 (Branching property)**.** System S has the *branching property* if for all $n \in N$ and for all $e_1 \neq e_2 \in \hat{E}$ such that $n \stackrel{e_1}{\Rightarrow} n_1$ and $n \stackrel{e_2}{\Rightarrow} n_2$, then we have that

1. either there is $n' \in N$ such that $n_1 \stackrel{e_2}{\Rightarrow} n'$ and $n_2 \stackrel{e_1}{\Rightarrow} n'$, or
2. for each $(n_1',n_2') \in ln(n,e_1,e_2)$, letting

$$L^i_{\mathtt{p}} \stackrel{\text{def}}{=} hd\left(\{e_i\!\downarrow_{\mathtt{p}}\cdot\varphi \mid \varphi \in LT(S,n_i',\mathtt{p})\}\right) \text{ with } i\in\{1,2\} \text{ and } \mathtt{p}\in\mathcal{P},$$

conditions (2a), (2b), and (2c) below hold.

(a) *choice-awareness*: $\forall \mathtt{p} \in \mathcal{P}$: either
   i. $L^1_{\mathtt{p}} \cap L^2_{\mathtt{p}} \subseteq \{\varepsilon\}$ and $\varepsilon \in L^1_{\mathtt{p}} \iff \varepsilon \in L^2_{\mathtt{p}}$, or
   ii. $\exists n' \in N, \pi_1, \pi_2$:
   $n_1' \stackrel{\pi_1}{\Rightarrow} n' \wedge n_2' \stackrel{\pi_2}{\Rightarrow} n' \wedge (e_1\cdot\pi_1)\!\downarrow_{\mathtt{p}} = (e_2\cdot\pi_2)\!\downarrow_{\mathtt{p}} = \varepsilon$

(b) *unique selector*: $\exists!\,\mathtt{s} \in \mathcal{P} :\ L^1_{\mathtt{s}} \cap L^2_{\mathtt{s}} = \varnothing \wedge \exists\, \mathtt{sr}!a \in L^1_{\mathtt{s}} \cup L^2_{\mathtt{s}}$

(c) *no race*: $\forall \mathtt{r} \in \mathcal{P} :\ L^1_{\mathtt{r}} \cap L^2_{\mathtt{r}} = \varnothing$

$\implies \forall \mathtt{s}_1\mathtt{r}?a_1 \in L^1_{\mathtt{r}}, \forall \mathtt{s}_2\mathtt{r}?a_2 \in L^2_{\mathtt{r}} : \ \forall i \neq j \in \{1,2\} : \ n'_i \overset{\pi_i}{\Rightarrow}$
$\implies dep(\mathtt{s}_i \to \mathtt{r} : a_i,\ e_i \cdot \pi_i,\ \mathtt{s}_j \to \mathtt{r} : a_j)$ ◇

Definition 3.5 ensures that every branching either is (1) the concurrent execution of two events; or, for each participant $\mathtt{p}$, (2(a)i) if $\mathtt{p}$ does not terminates before n, then the first actions of $\mathtt{p}$ in two different branches are disjoint; or (2(a)ii) $\mathtt{p}$ is not involved in the choice, i.e., the branches merge before $\mathtt{p}$ does any action; (2b) there is a unique participant $\mathtt{s}$ making the decision; and (2c) for each participant $\mathtt{r}$ involved in the choice, there cannot be a race condition between the messages that $\mathtt{r}$ can receive. The *no race* condition notably ensures that in any (asynchronous) execution of S, if a machine has more than one non-empty buffers, then it can read from them in any order (interleaving is possible). Note that if a machine $\mathtt{r}$ receives all its messages from a same sender, then there is a $\lhd$-relation between all its actions.

In system S_{re}, case (1) of Definition 3.5 applies to all branching nodes except $n_0 = (\mathtt{A}_0, \mathtt{B}_0, \mathtt{C}_0, \mathtt{D}_0)$ and $n = (\mathtt{A}_0, \mathtt{B}_0, \mathtt{C}_2, \mathtt{D}_1)$, highlighted in Figure 3, for which case (2) applies. For $e_1 = (\mathtt{A}_0, \mathtt{B}_0, \mathtt{A} \to \mathtt{B} : bwin)$ and $e_2 = (\mathtt{A}_0, \mathtt{C}_0, \mathtt{A} \to \mathtt{C} : cwin)$, we have $ln(n_0, e_1, e_2) = ln(n, e_1, e_2) = \{((\mathtt{A}_1, \mathtt{B}_1, \mathtt{C}_2, \mathtt{D}_1), (\mathtt{A}_1, \mathtt{B}_0, \mathtt{C}_4, \mathtt{D}_1))\}$. Hence, case (2a) holds for n_0 iff it holds for n. Following (2a), we check that every participant satisfies either (2(a)i) or (2(a)ii):

- $\mathtt{A}$ executes different (sending) actions in both branches ($\mathtt{AB}!bwin$ and $\mathtt{AC}!cwin$),
- $\mathtt{B}$ executes different (receiving) actions ($\mathtt{AB}?bwin$ and $\mathtt{CB}?blose$),
- $\mathtt{C}$ executes different (receiving) actions ($\mathtt{AC}?cwin$ and $\mathtt{BC}?close$),

hence case (2(a)i) applies to $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{C}$. While case (2(a)ii) applies to $\mathtt{D}$ since there is a node $n' = (\mathtt{A}_1, \mathtt{B}_2, \mathtt{C}_5, \mathtt{D}_1)$ such that $\mathtt{D}$ does not execute any action on either path from n to n' (through nodes $(\mathtt{A}_1, \mathtt{B}_1, \mathtt{C}_2, \mathtt{D}_1)$ and $(\mathtt{A}_1, \mathtt{B}_0, \mathtt{C}_4, \mathtt{D}_1)$, respectively). Also, condition (2b) is satisfied since $\mathtt{A}$ is the unique sender that executes different actions in both branches e_1 and e_2.

Condition (2c) is satisfied for $\mathtt{B}$ and $\mathtt{C}$ due to the existence of dependency chains from $\mathtt{AB}?bwin$ to $\mathtt{CB}?blose$ (and vice versa) and from $\mathtt{AC}?cwin$ to $\mathtt{BC}?close$ (and vice versa). For instance, the dependency chain $\mathtt{B} \to \mathtt{C} : close \lhd \mathtt{C} \to \mathtt{A} : msg \lhd \mathtt{A} \to \mathtt{C} : cwin$ prevents $\mathtt{C}$ to delay the reception of *close* (sent by $\mathtt{B}$) until she can receive message *cwin* (sent by $\mathtt{A}$); $\mathtt{C}$ must send a message *msg* (to $\mathtt{A}$) before she can receive the outcome of a new round of the game.

Finally, note that $ln(n_0, e_1, e_2)$ ensures that checking the branching between e_1 and e_2 at node n_0 is delayed until the interaction $\mathtt{C} \to \mathtt{D} : busy$ does not interfere with the choice. Hence, the behaviours of $\mathtt{C}$ and $\mathtt{D}$ are checked only once they have exchanged the *busy* message.

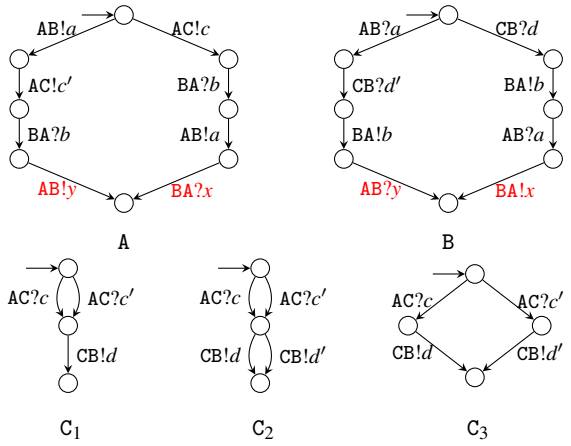
Proposition 3.2. *Given a system $S = (M_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$, checking whether S satisfies the branching property is computable in*

$$O\left(|\Rightarrow|^2 \times |\Rightarrow|! \times \sum_{\mathtt{r} \in \mathcal{P}} \left(|\delta_{\mathtt{r}}|^2 \right) \right) \text{ time.}$$

Checking the branching property is factorial in the size of $TS(S)$ because it requires the enumeration of paths of $TS(S)$ (cf. (2c) of Definition 3.5). We remark that the above is a rather coarse approximation obtained under worst case assumptions oblivious of the typical structure of $TS(S)$; our experiments show good performances (cf. § 5). Finally, we observe that $TS(S)$ is generally much smaller than, e.g., the one-bounded transition system of S (where each queue may contain at most one message).

Definition 3.6 (Generalised multiparty compatibility)**.** A system S is *generalised multiparty compatible* (*GMC*) if it is representable and has the branching property. ◇

Example 3.3. We show the interplay between the representability and branching conditions by exhibiting unsafe systems satisfying only one of the properties. Consider the following machines:



(1) System $S_1 = (\mathtt{A}, \mathtt{B}, \mathtt{C}_1)$ with $d = d'$ is not safe: whenever the left-hand side branch of $\mathtt{A}$ and the right-hand side branch of $\mathtt{B}$ are taken in a same execution, S_1 will reach an orphan message configuration where messages x and y are never consumed. In fact, S_1 is not GMC because there is a branching node from which $\mathtt{B}$ can execute, as first actions, either $\mathtt{AB}?a$ or $\mathtt{CB}?d$, and there is no dependency between the reception of a and that of d' (with $d = d'$) in the left-hand side branch, i.e., $\neg(\mathtt{A} \to \mathtt{B} : a \lhd \mathtt{A} \to \mathtt{C} : c' \lhd \mathtt{C} \to \mathtt{B} : d')$. Thus the branching property does not hold.
(2) System $S_2 = (\mathtt{A}, \mathtt{B}, \mathtt{C}_2)$ with $d \neq d'$ is not safe: as before, whenever the left-hand side branch of $\mathtt{A}$ and the right-hand side branch of $\mathtt{B}$ are taken in a same execution this system reaches an orphan message configuration. These two branches are not mutually exclusive since $\mathtt{C}_2$ can receive c' then send d. This system is not GMC since there is no node in $TS(S_2)$ such that actions $\mathtt{CB}!d$ and $\mathtt{CB}!d'$ are the first actions executed by $\mathtt{C}$. Hence the representability condition does not hold.
(3) System $S_3 = (\mathtt{A}, \mathtt{B}, \mathtt{C}_3)$ with $d \neq d'$ is safe and is GMC. In S_3, the left-hand side branch of $\mathtt{A}$ and the right-hand side branch of $\mathtt{B}$ are always mutually exclusive, while in S_1 and S_2 they are only mutually exclusive in synchronous executions.

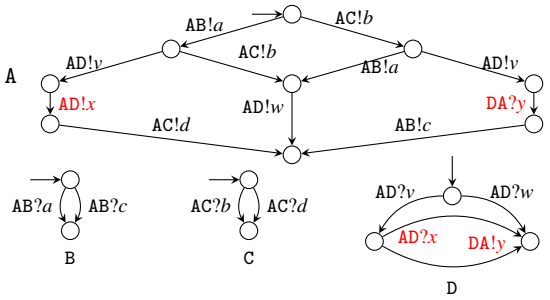
We remark that systems S_1 and S_2 may be easily changed so that they are "safe" in any k-bounded execution but not safe in a k+1-bounded execution. This may be done by making $\mathtt{A}$ and $\mathtt{B}$ exchange k+1 messages consecutively, e.g., by replacing every $\mathtt{AB}!a$ (resp. $\mathtt{AB}?a$) transitions in $\mathtt{A}$ (resp. $\mathtt{B}$) by a sequence of k+1 transitions $\mathtt{AB}!a_i$ (resp. $\mathtt{AB}?a_i$), for $1 \leqslant i \leqslant k+1$.

Theorem 3.1 (Soundness)**.** *If S is GMC, then it is* safe *(no orphan message, deadlock, and unspecified reception configurations).*

Theorem 3.1 says that no (asynchronous) execution of S will result in an orphan message, deadlock or unspecified reception configurations. Relying on representability (every transition and branching in each machine is represented in $TS(S)$), the proof shows that, for each branching node n, the function $ln(n, e_1, e_2)$ allows enough branches to be verified against the branching property. Then, it shows that any sent message is eventually received and that a machine in a receiving state eventually receives a message it expected, by Definition 3.5.

Example 3.4. The *unsafe* system below has the branching property and validates condition (1) of Definition 3.4, but *not* condition (2). This system can reach an orphan message configuration, where

messages x and y are never received.



This example illustrates the importance of condition (2) of Definition 3.4 to ensure safety. In the *TS* of this system (isomorphic to machine `A`), the branches corresponding to `DA!`y and `AD?`x of machine `D` are not checked against each other for the branching property.

3.3 Amending communicating systems

When a system is not GMC, our algorithm can be used to suggest different ways of transforming it, so to validate the condition. By Definition 3.6, we first note:

Proposition 3.3. *If S satisfies all but* (1) *in Definition 3.4, then the system consisting of the (minimised) projections of $TS(S)$ is GMC.*

This means that, in such a case, a new *safe* system may be automatically obtained from the projections of *TS*. For instance, system S_2 in Example 3.3 is not GMC because (1) in Definition 3.4 does not hold. However, the system corresponding to the projections of $TS(S_2)$ is exactly system S_3, which is GMC.

In case the projections of $TS(S)$ do not provide a viable alternative, then the language equivalence check allows to highlight which transitions (or paths) of each machine are not represented in $TS(S)$. Similarly, local states and transitions violating it can be singled out, according to condition (2) in Definition 3.4. For instance, in Example 3.4, we can highlight all transitions over x and y, as well as the states where they are enabled.

When the branching property (Definition 3.5) is violated, then our analysis permits to give precise information on where the problem occurs. First, we can give the vector of local states and the two branching events for which the problem occurs as well as a witnessing execution that leads to the offending configuration.

- If the choice-awareness condition (2a) is violated, then we can list the machines for which the condition is not satisfied. If a machine has a first same receiving action in both branches, then it may be corrected by simply renaming some messages. These renamings can be automatically suggested while checking for the branching property. If the condition fails because a machine terminates in one branch but not in the other, then we can suggest to add a new label and a transition to the final state in the terminated branch; as well as a dual transition in a sending machine.
- If condition (2b) is violated, we can highlight the set of machines sending messages at this branching node. A solution may be found by identifying the genuine selecting machine and add communications from this machine to the others.
- If condition (2c) is violated, then we can highlight, for each machine violating the condition, on which messages a race condition may occur; and suggest to add an acknowledgement message between the two corresponding actions.

Note that since CFSMs are specification or abstraction of programs, it is generally not desirable to automatically repair non-GMC systems. Indeed, some corrections may not be reflected easily in the program or might have side effects in the corresponding implementation, analogously to concurrent programming where automatic corrections are not generally appealing, even if some deadlocks may be detected automatically (at run or compile time).

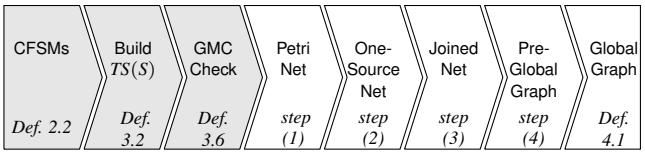


Figure 4. Work-flow of the construction

4. Building Global Graphs

In § 3, we construct the synchronous transition system $TS(S)$ of a communicating system S, and check whether it is GMC. We now describe the construction algorithm and its properties; Figure 4 summarises the work-flow of the transformations.

The algorithm to construct a global graph G from a synchronous transition system $TS(S)$ consists of the following steps:

(1) we apply the algorithm of Cortadella et al. [19] to *derive* a Petri net $\mathbb{N}$ from $TS(S)$;

(2) we transform $\mathbb{N}$ so that its initial marking consists of exactly one place;

(3) we join transitions whenever possible, so to make explicit join and fork points of the work-flow;

(4) we transform the net of (3) into a pre-global graph; finally, we "clean-up" the pre-global graph of unnecessary vertexes so to obtain a global graph.

For the sake of the presentation and because the transformations are rather mechanical, we explain them through our running example. The formal definitions of the transformations and additional results are given in Appendix B.

For (1), it is enough for the reader to know that the algorithm in Cortadella et al. [19] is based on the theory of regions [8] and transforms a transition system into a safe and extended free-choice *labelled* Petri net, whose reachability graph is bisimilar to the original transition system. Basically, this algorithm transforms events of $TS(S)$ into transitions of $\mathbb{N}$ while the places are built out of *regions*, i.e., sets of states having a uniform behaviour wrt events. We assume in this section that each $TS(S)$ is *self-loop free*[2], i.e., $\forall n, n' \in N : n \Rightarrow n' \implies n \neq n'$. The algorithm of [19] is applicable on a self-loop free $TS(S)$, since every event $e \in \hat{E}$ has an occurrence in $TS(S)$ by construction and every state n is reachable from n_0, as stated in Lemma 4.1 below. The Petri net obtained from $TS(S_{re})$ in Figure 3 is given in Figure 5 (left).

Lemma 4.1. *If S is GMC and $TS(S) = (N, n_0, \hat{E}, \Rightarrow)$, then $\forall n \in N : n_0 \Rightarrow^* n$.*

In step (2), we transform a Petri net obtained from Cortadella's algorithm into a Petri net whose initial marking consists of exactly one place. This allows us to construct a global graph that has a unique starting point. In our running example, the Petri net on the left of Figure 5 is transformed by adding a fresh place (p_0), initially

[2] In $TS(S)$, if an event e self-loops, then any transition labelled by e is a self-loop. Hence, we can easily lift the self-loop free assumption by decomposing each self-loop into two (pointed) transitions in $TS(S)$ and recompose them once the global graph is constructed.

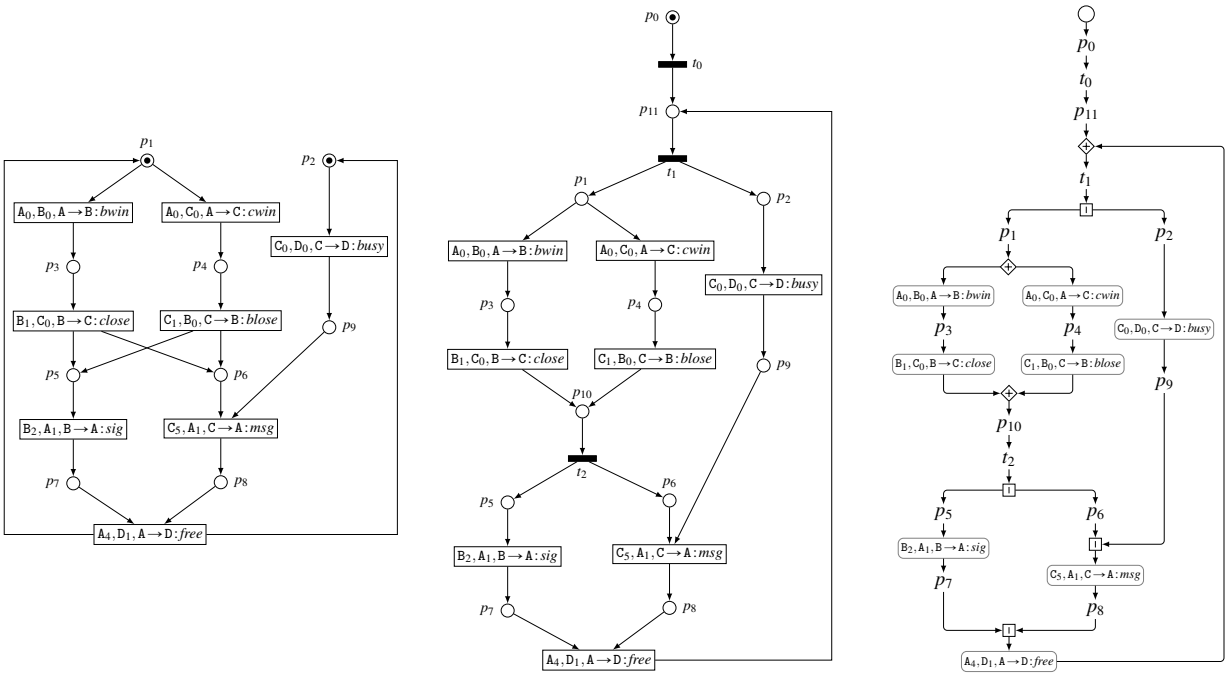


Figure 5. Derived net (left), net after transformations (middle), and pre-global graph (right)

marked, and a fresh (silent) transition (t_0) connected to places p_1 and p_2 (this simple transformation is not illustrated in Figure 5).

In step (3), a transformation ensures that parallel gates are used "as much as possible" in the graph (instead of mixing choice and parallel gates). In fact, the transformation joins sets of places that have the same preset or postset to minimise the number of choice gates. The Petri net in the middle of Figure 5 is the net obtained from the left-hand side net after applying step (2) and (3). In the second transformation, we add (*i*) t_1 and p_{11} so to join p_1 and p_2 which have the same preset, i.e., t_0 and the transition with label $(\mathtt{A}_4, \mathtt{D}_1, \mathtt{A} \to \mathtt{D} : \mathit{free})$; and (*ii*) we add t_2 and p_{10} so to join p_5 and p_6 which have the same preset, i.e., the transitions with labels $(\mathtt{C}_1, \mathtt{B}_0, \mathtt{C} \to \mathtt{B} : \mathit{blose})$ and $(\mathtt{B}_1, \mathtt{C}_0, \mathtt{B} \to \mathtt{C} : \mathit{close})$. Both t_1 and t_2 are silent transitions.

Let $\approx$ be the weak bisimilarity relation on reachability graphs (i.e., $\approx$ is the bisimilarity up-to silent transitions, cf. Appendix A).

Lemma 4.2. *Let $\mathbb{N}_1$ be the Petri net obtained after step (1), let $\mathbb{N}_2$ (resp. $\mathbb{N}_3$) be obtained by applying step (2) (resp. (3)) to $\mathbb{N}_1$ (resp. $\mathbb{N}_2$). If T_i is the reachability graph of $\mathbb{N}_i$ (for $i = 1,2,3$) then $T_1 \approx T_2 \approx T_3$.*

We now define global graphs (a superclass of the generalised global types of [20] that allows each gate to be connected to more than two predecessors or successors).

Definition 4.1 (Global graph)**.** A *global graph* (over $\mathcal{P}$ and $\mathbb{A}$) is a labelled graph $\langle V, A, \Lambda \rangle$ with set of vertexes V, set of edges $A \subseteq V \times V$, and labelling function Λ from V to $\{\bigcirc, ◎, ⟐, ⊡\} \cup \{\mathtt{s} \to \mathtt{r} : a \mid \mathtt{s}, \mathtt{r} \in \mathcal{P} \wedge a \in \mathbb{A}\}$ such that, $\Lambda^{-1}(\bigcirc)$ is a singleton, and for each $v \in V$, if $\Lambda(v)$ is of the form $\mathtt{s} \to \mathtt{r} : a$ then v has unique incoming and unique outgoing edges, and if $\Lambda(v) \in \{⟐, ⊡\}$, v has at least one incoming and one outgoing edge while v has no outgoing edges if $\Lambda(v) = ◎$. ◇

Label $\mathtt{s} \to \mathtt{r} : a$ represents an interaction where $\mathtt{s}$ sends a message a to $\mathtt{r}$. A vertex with label $\bigcirc$ represents the source of the global graph, ◎ represents the termination of a branch or of a thread, ⊡ indicates forking or joining threads, and ⟐ marks vertexes corresponding to branch or merge points, or to entry points of loops.

In step (4), a *pre-global graph* is obtained from the Petri net obtained after step (3) via a transformation which consists in, firstly, creating a vertex in the global graph for each place, transition, and element of the flow relation. Then these vertexes are connected via gates: a source vertex is connected to a vertex without predecessor, a sink vertex is connected to any vertex without successors, while transitions (resp. places) are connected to a ⊡-gate (resp. ⟐-gate) if they have more than one predecessors or successors. Finally, each component of the graph is connected by merging "ports" corresponding to elements of the flow relation. The pre-global graph for S_{re} (Figure 1) is given in Figure 5 (right).

A global graph is obtained from a pre-global graph by removing all unnecessary nodes (i.e., former places and transitions such as p_0 and t_0 in Figure 5) and relabelling events into interactions (e is replaced by $\iota(e)$); e.g., the pre-global graph in Figure 5 becomes the global graph in Figure 2.

Proposition 4.1. *Steps (2) to (4) are computable in polynomial time in the size of $\mathbb{N}$.*

We give the main result regarding the construction of a global graph from CFSMs. In Theorem 4.1 below, we formalise the relationship between the machines from which a global graph is constructed and its projections. Projecting a global graph G can be done in two ways: (*i*) G can be transformed into a Petri net whose reachability graph may be projected, similarly to the projection of $TS(S)$ (cf. Definition 3.3); or (*ii*) G can be transformed into an automaton whose states are the nodes of G and each transition is labelled by $(\mathtt{s} \to \mathtt{r} : a)\restriction_{\mathtt{p}}$ if the source state corresponds to a vertex with label $\mathtt{s} \to \mathtt{r} : a$, and by ε otherwise. In order to recover local concurrency, we take the parallel composition of the automata resulting of the projection of each successor of a ⊡-gate. Finally, the resulting automaton is minimised wrt. language equivalence.

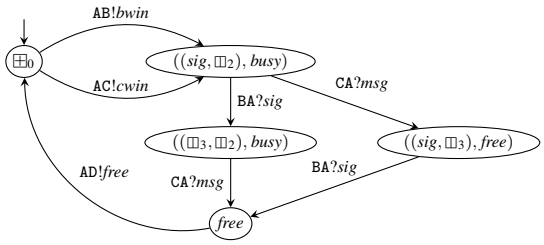


Figure 6. Projection of G_{re} onto A

We write $G\downarrow_{\mathtt{p}}$ for the projection of G onto p, and give the formal definition in [5]. As an example, Figure 6 shows the minimised projection of G_{re} (cf. Figure 2) onto A.

Theorem 4.1 (Completeness)**.** *Given a GMC system $S = (M_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$, let G be the global graph built from S and let $TS(S) = (N, n_0, \hat{E}, \Longrightarrow)$. If $TS(S)$ is self-loop free (i.e. $\forall n, n' \in N : n \Longrightarrow n' \implies n \neq n'$), then S is isomorphic to $(G\downarrow_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$, the system made of the projection of G.*

The proof of Theorem 4.1 (cf. [5]) relies on the fact that each machine is preserved during the construction, i.e., (1) the projection of $TS(S)$ onto each p is language equivalent with $M_{\mathtt{p}}$, (2) the net obtained from $TS(S)$ via the algorithm in [19] is bisimilar to $TS(S)$, (3) each transformation preserves (weak) bisimilarity with the derived net, cf. Lemma 4.2, and (4) the transformation to a global graph is sound since the net is extended free choice.

5. Implementation and Experimental Evaluation

In order to assess the applicability of our work and to estimate the effectiveness of checking for the GMC condition as well as constructing a global graph, we have developed a prototype tool supporting our theory [6]. The tool (implemented in Haskell) takes as input a textual representation of a communicating system S, then builds $TS(S)$ on which the representability condition and branching property are concurrently checked for (using HKC [13] to check for language equivalence). Then the tool constructs a global graph from $TS(S)$ relying on Petrify [2] (to derive a Petri net from $TS(S)$), and Graphviz (to render global graphs).

Table 1 summarises the results of experiments conducted on a few real-world protocols mainly taken from the literature. For each protocol, the table reports the number of machines, the number of nodes and transitions in $TS(S)$, whether it validates the GMC condition, the size of the constructed global graph, and the time it takes to check the condition *and* render its global graph (executions were on a 3.40GHz Intel i7 CPU with 16GB of RAM).

On most of the protocols the execution takes only a few seconds. To generate larger interesting examples, we tested systems consisting of the parallel composition of two protocols, e.g., *Running Example* $\times 2$ is the parallel composition of two instances of the running example. Graphical representations of these protocols are in [5]. Observe that in general the size of the constructed global graph (i.e., the number of vertices) is significantly smaller than the size of $TS(S)$, see *Running Example* $\times 2$ for instance. We note that it is slightly more expensive to check the *Running Example* and the *Logistic* protocols. This is due to the fact that each of these protocols features at least one participant for which checking condition (2c) of Definition 3.5 is not trivial, because they receive information about a choice from different participants, e.g., Carol in S_{re}. On the other hand, checking the *Alternating 3-bit* protocol is more time consuming due to larger $\blacklozenge$-equivalence classes.

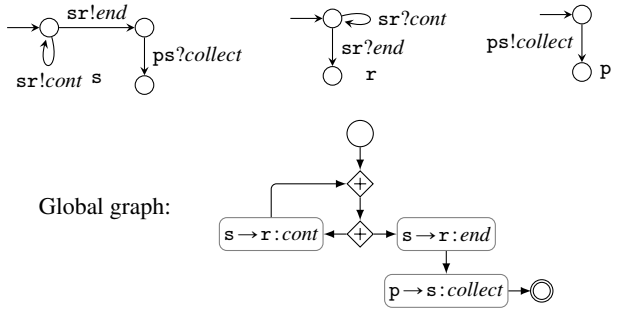
S	$\lvert\mathcal{P}\rvert$	$\lvert N\rvert$	$\lvert\Longrightarrow\rvert$	GMC	$\lvert G\rvert$	Time (s)
Running Example	4	12	19	✓	16	0.184
Running Example × 2	8	144	456	✓	32	22.307
Bargain	3	4	4	✓	8	0.103
Bargain × 2	6	16	32	✓	16	0.161
Alternating 2-bit [20]	2	8	12	✓	13	0.161
Alternating 2-bit × 2	4	64	192	✓	24	0.355
Alternating 3-bit [20]	2	24	48	✓	18	3.164
Alternating 3-bit × 2	4	576	2304	✓	34	12.069
TPMContract v2 [23]	2	5	8	✓	15	0.142
TPMContract v2 × 2	4	25	80	✓	30	0.362
Sanitary Agency [31]	4	17	21	✓	22	0.241
Sanitary Agency × 2	8	196	476	✓	44	3.165
Health System [15]	6	10	11	✓	14	0.17
Health System × 2	12	100	220	✓	28	1.702
Filter Collaboration [33]	2	3	5	✓	10	0.118
Filter Collaboration × 2	4	9	30	✓	20	0.178
Logistic [1]	4	13	17	✓	27	0.276
Logistic × 2	8	169	442	✓	54	2.155
Cloud System v4 [22]	4	7	8	✓	12	0.14
Cloud System v4 × 2	8	49	112	✓	24	0.432

Table 1. Experiment results; $\lvert\mathcal{P}\rvert$ is the number of machines, $\lvert N\rvert$ (resp. $\lvert\Longrightarrow\rvert$) is the number of nodes (resp. transitions) in $TS(S)$, and $\lvert G\rvert$ is the number of *vertices* in G.

6. Related Work

Session Types In the context of multiparty session types, [28] first suggested a construction of a global protocol from a set of local session types, up to asynchronous sub-typing. A typing system which infers a global type [24] from a set of session types is given in [25]. Recursive constructions are restricted in this work, due to an inherently syntax-driven typing system, and multi-threaded participants are not supported (i.e., in terms of CFSMs, this means that mixed states are *not* allowed).

Example 6.1. Consider the GMC system of three machines below. Machine s chooses to either continue interacting with machine r (sending *cont*), or notify r that it wants to terminate (sending *end*), before collecting some information from machine p (*collect*).



This system is not accepted by the typing system in [25] because machine p is not involved in the recursion (cf. rules $[\mu]$ and $[\mathbf{x}]$ in [25]).

In [21], the authors study the *synthesis* of global types from *basic* CFSMs, that is deterministic, non-mixed (each state is either sending or receiving), and directed (for each state, its outgoing transitions are all labelled by an action sending to, or receiving from, the same participant). Basic CFSMs do not allow to model general concurrency at the local level, since a machine cannot have mixed states. Note that machines A, B, and C in Figure 1 are not directed. The present work covers a much larger set of global protocols than [21, 25, 28]: we support mixed and non-directed states (hence, multi-threaded participants are allowed), recursive

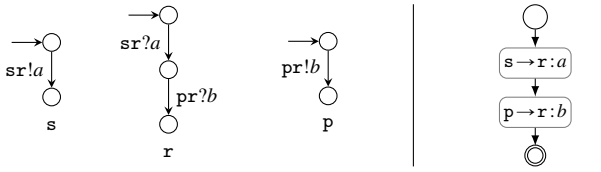
protocols are no longer restricted by a syntax oriented formalism, and explicit fork/join control points may be constructed.

The first translation from generalised global types into CFSMs was given in [20], where only sound properties were presented. The generalised global types of [20] are strictly included in GMC systems (Definition 3.6). The complete characterisation of global graphs and a construction algorithm were left as open problems. This paper solves these problems.

Choreographies Other recent works [10–12, 16, 23] study the relationship between global and local specifications, but *do not* consider the problem of building global specifications from local ones. Namely, in [11], *synchronisable* systems are shown to preserve some reachability properties regardless the communication being asynchronous or synchronous. Essentially, systems are synchronisable if their synchronous behaviour is equivalent to their one-bounded asynchronous behaviour (considering send actions only). In [12], the authors tackle the problem of determining whether a choreography is realisable. Essentially, a choreography is realisable if "it is possible to build a distributed system that communicates exactly as the choreography specifies". Choreographies in their work take the form of *conversation protocols*, that are finite state machines specifying the allowable sequence of interactions. A conversation protocol is akin to a global graph but without explicit construct for concurrent interactions, i.e., concurrent interactions must be specified by interleaving them.

We observe that both synchronisability and realisability conditions require strong properties on message ordering. In comparison, the GMC condition requires (*i*) the existence of a synchronous execution that encompasses all paths in each machine, and (*ii*) that each machine is either made aware every time a choice occurs or is not involved in the choice. In addition, a subtle difference between our machines and the machines in [11, 12] is that each of the latter machines has a unique buffer from which it can receive messages. Namely, their model is not suitable to reason about a CS as the interleaving of several multiparty sessions (where each participant has different receiving buffers in each session). In particular, their model cannot be used to represent programs which communicate via point-to-point communications, such as TCP connections between pairs of participants. We discuss a few examples that illustrate the main differences between the two communication models.

Example 6.2. Consider the GMC system below.



In our model, machine `r` receives messages from `s` and `p` from two different buffers; therefore, this system is safe (since machine `r` is always able to read message *a* and then consume *b* regardless of the order in which the messages from `s` and `p` arrive). In a model where machine `r` has only one FIFO buffer to receive both messages from `s` and `p`, machine `r` will end up in an unspecified reception configuration if message *b* reaches the queue before *a*. The system above is *not synchronisable*, since its synchronous execution differs from its one-bounded asynchronous execution (considering send actions only). Symmetrically, its choreography is *not realisable*.

Observe that the system $(\mathtt{A},\mathtt{B},\mathtt{C}_1)$, from Example 3.3, is unsafe in our communication model, but safe in theirs (where it is *synchronisable*). In that model, safety follows from the fact that machine `B` would have only one buffer. Hence, if `A` chooses the left-hand side branch, message *a* will be in `B`'s queue before, thus `B` must execute its left-hand side branch; while if `A` chooses the right-hand side branch, *d* will appear on `B`'s queue first and the latter will then execute its right-hand side branch. Finally, note that the GMC system (resp. choreography) in Example 6.1 is not synchronisable (resp. realisable) either due to the "race" between the send actions from machines `s` and `p`.

Automata & MSC The term *synthesis* of CFSMs has been used to describe the reduction of CS to a more manageable (and decidable) model, e.g., with partial order approaches (see [30] for a summary of recent results). The acceptation of the term *synthesis* in this context is to identify a system of CFSMs that realises a protocol described by an incomplete specification (such as in [9, 29]). These approaches do not yield a global specification as instead achieved by our algorithm. In addition, our approach enables the verification of trace-based properties surveyed in [30]. For instance, the closed synthesis of CFSMs can be reduced to the construction from a regular language L of a machine satisfying certain conditions related to buffer boundedness, deadlock-freedom, and words swapping.

In [27] a tool chain is given to synthesise an orchestrator (i.e., a message forwarder) from a set of finite-state machines communicating synchronously. This is transformed into a BPMN diagram via a Petri net transformation based on [19]. The work [32] gives an algorithm to compose several services. Each service is presented as an automaton and a set of automata are composed by a parallel product. The composite automaton is then transformed into a Petri net, using [19]. In both works, no result regarding safety or preservation of the behaviour of the original machines is given.

The work [7] studies whether Message Sequence Charts (MSC) imply unspecified scenarios (where MSCs are implemented by concurrent automata, but do not necessarily feature order-preserving communications). It gives conditions on MSCs for their implementation to be deadlock-free and realisable. MSCs are realisable if no other MSC may be inferable from them. It does not attempt to give an exhaustive global view of a distributed system, but focuses on identifying its possible misbehaviours.

7. Conclusions & Future Work

We have given a complete algorithm whereby one can build a global graph (choreography) from any *generalised multiparty compatible* (GMC) system. GMC systems form a new class of communicating systems, and we have proved that any system in this class is safe and there exist efficient algorithms to check GMC. Our work effectively uses the theory of regions [19], bridging a gap between a set of distributed uncontrolled behaviours (represented by CFSMs) and well-structured graphical session types, while offering a scalable implementation for our framework.

Since the original machines can be recovered by projecting the constructed global graph (by Theorem 4.1), we can use our framework to develop a software development life cycle based on choreographies: a specification written as a choreography is projected onto a set of local models which will then be refined against their implementations. Such an approach can also be used to reverse-engineer existing distributed systems. We are currently collaborating with the Zero Deviation Lifecyle project [4] which proposes a platform to attain "near-zero defect leakage across the various phases of the software development lifecycle". Updating global scenarios against local models plays an important role in different stages of software life cycle in this architecture. Our framework applies naturally to this platform, which notably uses BPMN 2.0 Choreography [1] specifications and tools.

We also plan to investigate a relaxed version of the GMC condition which would allow to build global graphs whose projections are equivalent to the original system, up-to asynchronous order-preserving communication [28].

Acknowledgments

We thank Pierre-Malo Deniélou for initial discussions on this work and the ZDLC team at Cognizant for their stimulating conversations. This work is partially supported by UK EPSRC projects EP/K034413/1, EP/K011715/1, and EP/L00058X/1; and by EU 7FP projects under grant agreements 295261 (MEALS) and 612985 (UPSCALE) and COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY).

References

[1] Business Process Model and Notation. http://www.bpmn.org.

[2] Petrify. http://www.lsi.upc.edu/~jordicf/petrify/.

[3] SAVARA Testable Architecture. http://www.jboss.org/savara.

[4] Zero Deviation Lifecycle. http://www.zdlc.co.

[5] Full version of this paper. http://www.doc.ic.ac.uk/~jlange/papers/lty15.pdf, 2014.

[6] GMC-Synthesis. https://bitbucket.org/julien-lange/gmc-synthesis, 2014.

[7] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Trans. Software Eng.*, 29(7):623–633, 2003.

[8] E. Badouel and P. Darondeau. Theory of regions. In *Petri Nets*, volume 1491 of *LNCS*, pages 529–586. Springer, 1996.

[9] C. Baier, J. Klein, and S. Klüppelholz. Synthesis of reo connectors for strategies and controllers. *Fundam. Inform.*, 130(1):1–20, 2014.

[10] S. Basu and T. Bultan. Choreography conformance via synchronizability. In *WWW*, pages 795–804. ACM, 2011.

[11] S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In *VMCAI*, volume 7148 of *LNCS*. Springer, 2012.

[12] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *POPL*, pages 191–202. ACM, 2012.

[13] F. Bonchi and D. Pous. Checking nfa equivalence with bisimulations up to congruence. In *POPL*, pages 457–468. ACM, 2013.

[14] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, 1983.

[15] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing service composition. In *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE07)*, 2007.

[16] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multi-party session. *LMCS*, 8(1), 2012.

[17] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *I&C*, 202(2):166–190, 2005.

[18] W. W. W. Consortium. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/, 2005.

[19] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets for Finite Transition Systems. *IEEE Trans. Computers*, 47 (8):859–882, 1998.

[20] P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[21] P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186, 2013.

[22] M. Güdemann, G. Salaün, and M. Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In *ATVA*, volume 7561 of *LNCS*, pages 238–253, 2012.

[23] S. Hallé and T. Bultan. Realizability analysis for message-based interactions using shared-state projections. In *SIGSOFT FSE*, pages 27–36. ACM, 2010.

[24] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

[25] J. Lange and E. Tuosto. Synthesising Choreographies from Local Session Types. In *CONCUR*, volume 7454 of *LNCS*, pages 225–239. Springer, 2012.

[26] R. Lanotte, A. Maggiolo-Schettini, and A. Troina. Weak bisimulation for probabilistic timed automata. *Theor. Comput. Sci.*, 411(50):4291–4322, 2010.

[27] S. McIlvenna, M. Dumas, and M. T. Wynn. Synthesis of orchestrators from service choreographies. In *APCCM*, volume 96 of *CRPIT*, pages 129–138. ACS, 2009.

[28] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009.

[29] M. Mukund, K. N. Kumar, and M. A. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *CONCUR*, volume 1877 of *LNCS*, pages 521–535. Springer, 2000.

[30] A. Muscholl. Analysis of communicating automata. In *LATA*, volume 6031 of *LNCS*, pages 50–57. Springer, 2010.

[31] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS*, pages 43–. IEEE Computer Society, 2004.

[32] Y. Wang, A. Nazeem, and R. Swaminathan. On the optimal Petri net representation for service composition. In *ICWS*, pages 235–242. IEEE Computer Society, 2011.

[33] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

A. Equivalences between Petri Nets

We give the formal definitions of the reachability graph of a Petri net and weak-bisimulation, which are used in Section 4.

Definition A.1 (Reachability graph [19])**.** Given $\mathbb{N} = (P, T, F, m_0)$, we say that a transition $t \in T$ is enabled at marking m_1 if all its input places are marked. An enabled transition t may fire, producing a new marking m_2 with one less token in each input place and one more token in each output place. We write $m_1 \xrightarrow{t} m_2$, if m_2 is reachable from m_1 by firing t, and write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$.

The *reachability graph* of $\mathbb{N}$ is the transition system $RG(\mathbb{N}) = (M, m_0, \hat{E}, \rightarrow)$ such that $M = \{m \mid m_0 \rightarrow^* m\}$;

- $\rightarrow = \{(m_1, lab(t), m_2) \mid m_1, m_2 \in M \wedge m_1 \xrightarrow{t} m_2\}$ (where $lab(t) = \varepsilon$ if the label of t is ε, and return the label e of t otherwise); and
- $\hat{E} = \{e \mid \exists (m_1, e, m_2) \in \rightarrow \wedge e \neq \varepsilon\}$;

Let $m \xrightarrow{e} m'$ iff $(m, e, m') \in \rightarrow$ and $m \stackrel{e}{\Rightarrow} m'$ iff $m(\xrightarrow{\varepsilon})^* \xrightarrow{e} (\xrightarrow{\varepsilon})^* m'$, with $e \neq \varepsilon$. ◇

The notion of weak bisimulation between two transition systems is given in Definition A.2 (adapted from [26])

Definition A.2 (Weak bisimulation)**.** Let $T = (M, m_0, \hat{E}, \rightarrow)$ be a transition system. A weak bisimulation on T is an equivalence relation $\mathcal{B} \subseteq M \times M$ s.t. for all $(m_1, m_2) \in \mathcal{B}$, the following holds

- $m_1 \xrightarrow{e} m_1'$ implies that there is m_2' such that $m_2 \stackrel{e}{\Rightarrow} m_2'$ and $(m_1', m_2') \in \mathcal{B}$; and
- $m_2 \xrightarrow{e} m_2'$ implies that there is m_1' such that $m_1 \stackrel{e}{\Rightarrow} m_1'$ and $(m_1', m_2') \in \mathcal{B}$.

Two states m_1 and m_2 are called *weakly bisimilar* on T, written $m_1 \approx_T m_2$, iff $(m_1, m_2) \in \mathcal{B}$ for some weak bisimulation $\mathcal{B}$.

Two transition systems $T_i = (M_i, m_0^i, \hat{E}_i, \rightarrow_i)$, $i \in \{1, 2\}$, such that $M_1 \cap M_2 = \varnothing$, are weakly bisimilar, written $T_1 \approx T_2$, if given

- $M' = M_1 \cup M_2 \cup \{m_0\}$ and $\hat{E}' = \hat{E}_1 \cup \hat{E}_2$,
- $T' = (M', m_0, \hat{E}', \rightarrow_1 \cup \rightarrow_2 \cup \{(m_0, \varepsilon, m_0^1), (m_0, \varepsilon, m_0^2)\})$

$m_0^1 \approx_{T'} m_0^2$ holds. ◇

B. From Petri Nets to Global Graphs

In this section we give the detailed transformations omitted in Section 4. The algorithm to construct a global graph G from a synchronous transition system $TS(S)$ consists of the following steps: (1) using the algorithm of Cortadella et al. [19], we derive a Petri net $\mathbb{N}$ from $TS(S)$; (2) we transform $\mathbb{N}$ so that its initial marking consists of exactly one place (Transformation B.1 below); (3) we join transitions whenever possible, so to make joins and forks explicit (Transformation B.2 below); (4) we transform the net of (3) into a pre-global graph (Transformation B.3 below); finally, we "clean-up" the pre-global graph of any unnecessary vertexes so to obtain a global graph (Transformation B.4 below).

Definition B.1 (Labelled net)**.** A *labelled Petri net*, or net, $\mathbb{N}$ is a quadruple (P,T,F,m_0) with P a set of places (ranged over by p), T a set of transitions (ranged over by t), $F \subseteq (P \times T) \cup (T \times P)$ the flow relation, and m_0 the initial marking. Each transition $t \in T$ is labelled with an event $e \in \hat{E}$, or marker ε (the latter representing a silent transition). We let x range over elements of $P \cup T$. As usual, ${}^\bullet x$ (resp. $x^\bullet$) is the preset (resp. postset) of x. A net is called *safe* if, for all reachable markings, no more than one token can appear in each place; in which case the reachable markings (including m_0) are sets of places. A net is *extended free-choice* if $\forall p \in P, \forall t \in T : \ (p,t) \in F \implies ({}^\bullet t \times p^\bullet) \subseteq F$. ◇

In the second step (2), we transform a Petri net obtained from Cortadella's algorithm into a Petri net whose initial marking consists of exactly one place. This allows us to construct a global graph that has a unique starting point (source).

Transformation B.1 (One-source net)**.** Given a labelled Petri net $\mathbb{N} = (P,T,F,m_0)$, the *one-source net* of $\mathbb{N}$ is $\mathbb{N}' = (P \cup \{p_0\}, T \cup \{t'\}, F', \{p_0\})$ such that $p_0 \notin P$, $t' \notin T$ is labelled by ε, and $F' = F \cup \{(p_0,t')\} \cup \bigcup_{p \in m_0} \{(t',p)\}$.

Proposition B.1. *Transformation B.1 is computable in linear time in the size of* m_0.

We can now state the following result, formalising the soundness of Transformation B.1.

Lemma B.1. *If T is the reachability graph of the Petri net $\mathbb{N}$ obtained from $TS(S)$ via the algorithm in [19], and T' is the reachability graph of the Petri net obtained after applying Transformation B.1, then $T \approx T'$.*

Next, Transformation B.2 ensures that parallel gates are used "as much as possible" in the graph (instead of mixing choice and parallel gates). In fact, Transformation B.2 joins sets of places having the same preset or postset to decrease the number of choice gates.

Transformation B.2 (Joined net)**.** The *joined net* of $\mathbb{N} = (P,T,F,m_0)$ is a net $\mathbb{N}' = (P',T',F',m_0)$ such that the following transformations are applied repeatedly:

1. for all maximal $X \subseteq P$ s.t. $|X| > 1$ and $\forall p_1,p_2 \in X : \ {}^\bullet p_1 = {}^\bullet p_2 \wedge |{}^\bullet p_1| > 1$, $P' = P \cup \{p'\}$ and $T' = T \cup \{t'\}$ with $p' \notin P$ and $t' \notin T$ and labelled by ε; also, chosen $p \in X$, $F' = \{(p',t')\} \cup ({}^\bullet p \times \{p'\}) \cup (\{t'\} \times X) \cup F \setminus \bigcup_{x \in X} {}^\bullet x \times \{x\}$
2. for all maximal $X \subseteq P$ s.t. $|X| > 1$ and $\forall p_1,p_2 \in X : \ p_1{}^\bullet = p_2{}^\bullet \wedge |p_1{}^\bullet| > 1$, $P' = P \cup \{p'\}$ and $T' = T \cup \{t'\}$ with $p' \notin P$ and $t' \notin T$ and labelled by ε; also, chosen $p \in X$, $F' = \{(t',p')\} \cup (\{p'\} \times p^\bullet) \cup (X \times \{t'\}) \cup F \setminus \bigcup_{x \in X} \{x\} \times x^\bullet$.

Note that the definition of F' does not depend on the choice of p.

Proposition B.2. *Transformation B.2 is computable in polynomial time in the size of* $\mathbb{N}$.

Since we are working with *safe* nets, we have the result below.

Lemma B.2. *If T (resp. T') is the reachability graph of the Petri net $\mathbb{N}$ obtained after Transformation B.1 (resp. Transformation B.2), then $T \approx T'$.*

Definition B.2 (Graph composition)**.** Let $\mathbb{N}_i = (P_i,T_i,F_i,m_{0_i})$ with $i \in \{1,2\}$ be two nets and $G_i = \langle V_i,A_i,\Lambda_i \rangle$ two graphs such that $V_i = P_i \cup T_i \cup F_i$, $i \in \{1,2\}$, the *composition of G_1 and G_2*, denoted by $G_1 \uplus G_2$ is a graph $\langle V,A,\Lambda \rangle$ defined as:

- $V = \{v \in V_1 \mid v \in F_1 \implies v \notin V_2\} \cup \{v \in V_2 \mid v \in F_2 \implies v \notin V_1\}$
- $A = ((A_1 \cup A_2) \cap V \times V) \cup \{(v,v') \mid \exists v'' \in F_i : \ (v,v'') \in A_i, (v'',v') \in A_j \wedge i \neq j \in \{1,2\}\}$ ◇

Intuitively, the composition of the graphs consists of (1) the union of the two sets of vertexes, except flow elements (p,t) and (t,p) if they appear in both V_1 and V_2; and (2) the union of the two sets of arcs between vertexes in V, and each pair of arcs of the form $(v,(x,x'))$ or $((x,x'),v')$ is replaced by a single arc (v,v').

Transformation B.3 (Pre-global graph)**.** The *pre-global graph* of $\mathbb{N} = (P,T,F,\{p_0\})$ is a tuple $\langle V,A,\Lambda \rangle$ such that $V = P \cup T \cup F$, Λ is a labelling function such that $\Lambda(v) = v$ if $v \in P \cup F$ or $v \in T$ labelled by ε, and $\Lambda(v) \in \hat{E} \cup \{◎, ○, ⊗, ◫\}$ otherwise; and A is given by:

$$T_g(\mathbb{N}) = \biguplus_{x \in P \cup T} T_i(x) \uplus T_o(x) \qquad \text{where, given } x \in P \cup T :$$

$$T_i(x) \stackrel{\text{def}}{=} \begin{cases} ○ \to x & \text{if } {}^\bullet x = \varnothing \\ (x',x) \to x & \text{if } {}^\bullet x = \{x'\} \\ (x_1,x), \ldots, (x_k,x) \to ⊗ \to x & \text{if } {}^\bullet x = \{x_1,\ldots,x_k\} \end{cases}$$

$$T_o(x) \stackrel{\text{def}}{=} \begin{cases} x \to ◎ & \text{if } x^\bullet = \varnothing \\ x \to (x,x') & \text{if } x^\bullet = \{x'\} \\ x \to ⊗ \to (x,x_1), \ldots, (x,x_k) & \text{if } x^\bullet = \{x_1,\ldots,x_k\} \end{cases}$$

with $k > 1$, $⊗ = ◈$ if $x \in P$, and $⊗ = ◫$ if $x \in T$.

The pre-global graph of S_{re} (Figure 1) is given in Figure 5 (right). Observe that all the vertexes of the form (x,x'), corresponding to an element of the flow relation, are removed as part of the graph composition (Definition B.2).

Proposition B.3. *Transformation B.3 is computable in polynomial time in the size of* $\mathbb{N}$.

We define the final transformation which cleans up a pre-global graph by removing unnecessary vertexes and arcs.

Transformation B.4. A global graph $G = \langle V,A,\Lambda \rangle$ is obtained from a pre-global graph $\langle P \cup T \cup F, A', \Lambda' \rangle$ by applying the following transformation: (1) replace each pair of transition $(x,p),(p,x') \in A'$ by $(x,x') \in A$; (2) replace each pair of transition $(x,t),(t,x') \in A'$, with t labelled by ε, by $(x,x') \in A$; and (3) label each t which is labelled by $(q_{\mathtt{s}}, q_{\mathtt{r}}, \mathtt{s} \to \mathtt{r} : a)$ in $\mathbb{N}$, by $\mathtt{s} \to \mathtt{r} : a$.

Proposition B.4. *Transformation B.4 is computable in polynomial time in the size of* $\mathbb{N}$.