Design-by-Contract for *Flexible* Multiparty Session Protocols — Extended Version

Lorenzo Gheri
Imperial College London, UK

Ivan Lanese
Focus Team, University of Bologna/INRIA (Italy)

Neil Sayers
Imperial College London, UK
Coveo Solutions Inc., Canada

Emilio Tuosto
Gran Sasso Science Institute, Italy

Nobuko Yoshida
Imperial College London, UK

Abstract

Choreographic models support a correctness-by-construction principle in distributed programming. Also, they enable the automatic generation of correct message-based communication patterns from a global specification of the desired system behaviour. In this paper we extend the theory of choreography automata, a choreographic model based on finite-state automata, with two key features. First, we allow participants to act only in some of the scenarios described by the choreography automaton. While this seems natural, many choreographic approaches in the literature, and choreography automata in particular, forbid this behaviour. Second, we equip communications with assertions constraining the values that can be communicated, enabling a design-by-contract approach. We provide a toolchain allowing to exploit the theory above to generate APIs for TypeScript web programming. Programs communicating via the generated APIs follow, by construction, the prescribed communication pattern and are free from communication errors such as deadlocks.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Formal software verification

Keywords and phrases Choreography automata, design by contract, deadlock freedom, Communicating Finite State Machines, TypeScript programming

Digital Object Identifier 10.4230/LIPIcs...

Supplementary Material Software: ECOOP 2022 Artifact Evaluation approved artifact, also available at `https://github.com/Tooni/CAScript-Artifact`

Funding Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233. Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems). Lanese and Tuosto are partially supported by INdAM as members of GNCS (Gruppo Nazionale per il Calcolo Scientifico). The work is partially supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS.

Acknowledgements We thank the anonymous reviewers for their useful comments and suggestions. We thank Fangyi Zhou for their help with building our artifact on top of their software, νScr.

1 Introduction

The development of communicating systems is notoriously a challenging endeavour. In this application domain, both researchers and practitioners consider choreographies a valid

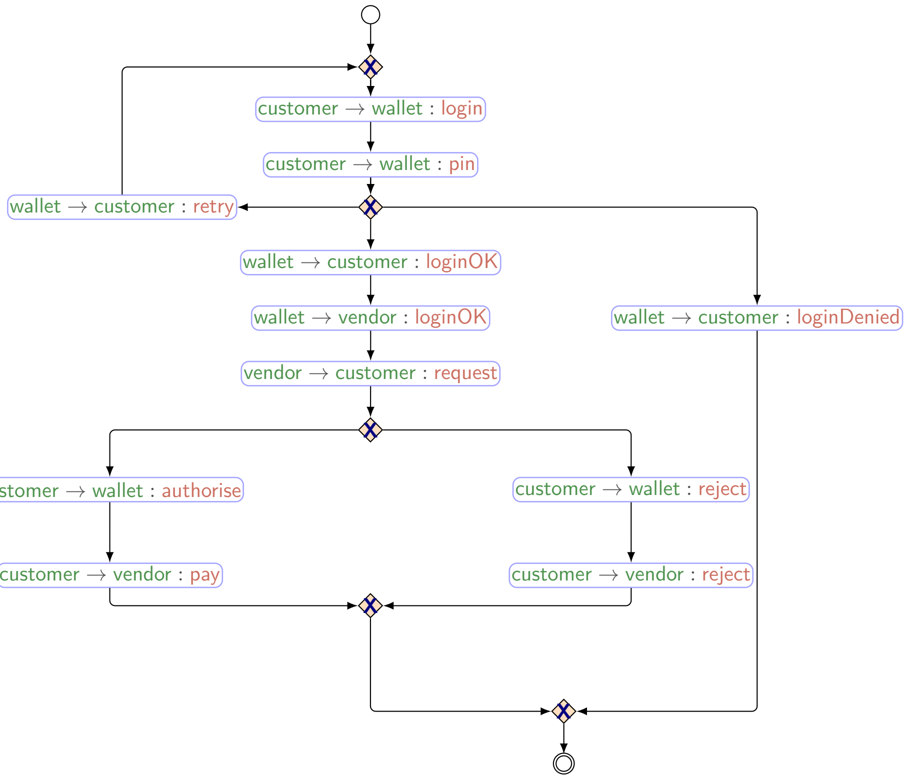


Figure 1 BPMN-like description of the Online Wallet

approach to tackle software development (e.g. [27, 39, 1, 4, 15]). Besides being naturally geared toward scalability (due to the lack of central components), choreographic models have been specifically conceived to support a *correctness-by-construction* [27] principle hinging on the interplay between *global* and *local views*. The former is a description of the interactions among (the *role* of) participants. We illustrate this through an OnLineWallet (OLW) service, adapted from [38] and akin to PayPal, used by vendors to process from customers. Fig. 1 describes our OLW protocol in terms of a BPMN-like diagram (diamonds denote choices or merges). The three participants involved are customer, wallet, and vendor. The former tries first to login into its account on the wallet server. In case of failure, wallet may ask for a retry, or may decide to deny access. A successful authentication is communicated by wallet to customer and vendor through the loginOK message; the vendor then sends a request for payment to customer, who can authorise or reject the transaction.

A natural question to ask is "can the OLW protocol be faithfully realised by distributed components?" The answer to this question requires a careful formalisation which we carry out in the next sections. For the moment, we appeal to intuition and interpret *realisation* as the existence of a set of components that coordinate with each other exclusively by message-passing and *faithful* as the fact that components execute all and only the communications prescribed by the global view without incurring in communication errors such as deadlocks. Local views specify the behaviour of each participant "in isolation". For instance, the local view of vendor above consists of an artefact which, after having received the notification from wallet, sends a request message to customer, and then waits for either a payment or a rejection message from customer. Note that vendor is "oblivious" of the interactions between customer and wallet. Also, observe that, if customer fails to authenticate to wallet (e.g., by typing a wrong password), then no payment request can be made. In this case, it does not make sense to involve vendor in the protocol. We call the ability to involve a participant only in some branches of a protocol *selective participation*.

Rather than an exception, selective participation is a norm in distributed applications, e.g., for data validation, prevention of server overload, or access control. Consider, e.g., services giving public access to some resources while requiring authentication to grant access to others. Often, the authentication phase is outsourced to external services (e.g., providing OAuth2.0 [17] and Kerberos [28] authentication). In this case, accesses to public resources should be oblivious to authentication services while protected resources are not involved in the communication until the authentication phase is cleared (as for vendor in our example). Other examples of selective participation emerge from smart contracts for online money transations (e.g., crowdfunding services as [29]), where participants take part to some stages of the communication only in case of a positive outcome of some financial operation.

A paramount element for the correctness-by-construction principle is the notion of *well-formedness*, namely sufficient conditions guaranteeing the faithful realisation of a protocol. Actually, choreographies advocate the algorithmic derivation, by *projection*, of faithful realisations from well-formed global views [27]. In fact, the so-called *top-down* choreographic approach to development consists of (a) the definition of a well-formed global view of the protocol, (b) the projection of the global view onto local ones, (c) the verification that each implemented component complies with a local view.

Usually, global views abstract away from local computations; for instance, the diagram of OLW in Fig. 1 does not specify how wallet takes the decision of letting customer retry the authentication or the strategy of customer to authorise or not the payment. Both these (and the other local) computations are blurred away because they require to specify the data dependencies that local computations should enforce. As pioneered in [3] in the context of global types [22], assertion methods can abstractly handle those dependencies by suitably constraining the payloads of interactions. Roughly, this transfers design-by-contract [34] methods to message-passing applications by imposing rely-guarantee relations on interactions. As shown in [3], this poses several challenges due to two main reasons. Firstly, pre-conditions ensuring the feasibility of some interactions depend on information scattered across distributed participants. Hence, it is necessary that data flow to participants so that all the information necessary for a participant to guarantee some assertion is available when needed. This requires to restrict to *history sensitive* [3] protocols, namely specifications have to be such that participants required to guarantee an assertion are aware of the information needed to satisfy it. Secondly, a careless use of such assertions may lead to inconsistent specifications so to eventually spoil the realisability of the protocol. This requires to restrict to *temporally satisfiable* [3] protocols, where no assertion ever becomes inconsistent during the execution.

Models and results based on the top-down approach to choreography abound in the literature (see, e.g., the survey [25]). This paper builds on *choreography automata* (c-automata) [2]; intuitively, a c-automaton is a finite-state machine whose transitions are labelled by interactions such as those in Fig. 1. The use of automata brings several benefits. On the one hand, automata models are well-known to both academics and industrial computer scientists and engineers. On the other hand, they allow one to exploit the well-developed theory of automata. Furthermore, automata do not have syntactic constraints imposed by algebraic models such as multiparty session types (see, e.g., [21, 43, 7]). Indeed, as noted in [2], c-automata seem to be more flexible than "syntax"-based formalisms such as global graphs [45] or multiparty session types. This is due to the fact that, in the latter family, well-formedness is attained via syntactic restrictions that rule out unrealisable protocols. Indeed,

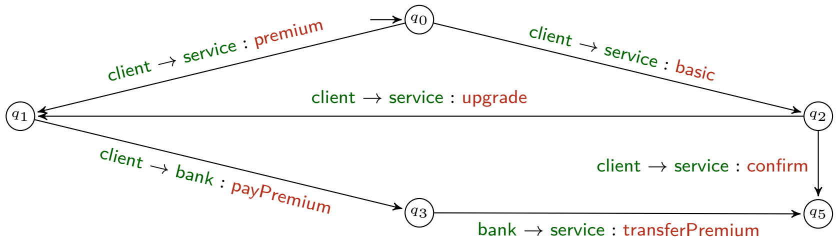


Figure 2 Non-well-structured choreography

a distinguished feature of c-automata is that they admit *non-well-structured* interactions. Let us explain this with the c-automaton in Fig. 2, modelling a choreography where a client registers to a service according to two options. If client opts for the basic level, then no payment is due, while the premium option requires a bank payment. Thus, we have a choice at q_0 between the basic and premium service levels. Then, in state q_2 of Fig. 2, client either confirms the choice or decides to upgrade. (Selective participation is required since the bank only acts in the "left" run.) In a structured model, the "left" and the "right" runs from q_0 to q_5 must be different branches of a choice. But those models cannot encode the upgrade transition that intuitively allows one to move from one branch to the other, before the end of the choice construct.

Contribution and structure. We provide two main contributions to the theory of c-automata, as well as an implementation in the setting of TypeScript programming.

First, we extend c-automata with selective participation, which, although natural as seen above, is actually forbidden in many choreographic models (e.g., [21, 43, 7]) including c-automata [2]. For instance, we will use the OLW protocol, where vendor's involvement occurs only on successful authentication, as our running example.

Our second contribution is the definition of *asserted c-automata*, that is a design-by-contract framework for c-automata. More precisely, we equip transitions with assertions constraining the exchanged messages, allowing one to specify such policies. For example, we can specify that the authentication of OLW customer can fail at most 3 times. At a glance, asserted c-automata mimick the constructions introduced in [3]. However, the generalisation of c-automata to selective participation (not featured in [3]) and the greater flexibility introduced by non well-structured interactions require to address non-trivial technical challenges that we discuss in § 4.

The last contribution is a toolchain, dubbed CAScr, based on the theory of c-automata with selective participation developed in this paper. More precisely, CAScr allows one to specify a protocol using the Scribble framework [20, 37, 47] and to check its well-formedness relying on our theory. Finally, CAScr generates TypeScript APIs to implement the roles of the original protocol. To the best of our knowledge, CAScr is the first toolchain that integrates Scribble with the flexibility of the theory of c-automata.

Our paper is structured as follows. § 2 introduces notions on finite state automata, and in particular on communicating finite state machines, to model participants, and on c-automata.

§ 3 develops the theory of c-automata. The main novelty w.r.t. [2] is to allow for selective participation. The resulting framework is more flexible with respect to [2], e.g., it allows one to prove that the OLW protocol can be faithfully projected. Even in this more general setting we can prove standard results: the implementation has the same behaviour as the original specification (Corollary 3.16) and is free from deadlocks (Thm. 3.19). Also, when focusing on one of the participants the projected system is lock free (Thm. 3.23).

§ 4 develops our second contribution, namely design-by-contract in the setting of c-automata. More precisely, c-automata are extended with assertions (Def. 4.6) and the related theory is extended accordingly. Also in this setting the implemented system faithfully executes its specification (Corollary 4.24) and it is deadlock free (Thm. 4.25).

§ 5 presents CAScr, a novel, full toolchain—from the Scribble [20, 37, 47] specification of the communication protocol, to the generation of APIs—providing support for distributed web development in TypeScript and relying on flexible c-automata with selective participation.

Finally, § 6 discusses related work, while § 7 draws some conclusions, and sketches future directions. A possible extension of our implementation is discussed in the appendix.

2 Choreography Automata and Communicating Systems

This section recalls basic notions about automata in general and about choreography automata (c-automata) [2] and systems of Communicating Finite State Machines (CFSMs) [5] in particular. Following [2], global views, rendered as c-automata, are projected into systems of local descriptions modelled as CFSMs. We start by surveying finite-state automata (FSA).

▶ **Definition 2.1** (FSA). *A* labelled transition system *(LTS) is a tuple* $(Q, q_0, \mathcal{L}, \mathcal{T})$ *where*

- Q *is a set of states (ranged over by* $s, q, \ldots$*) and* $q_0 \in Q$ *is the* initial state*;*
- $\mathcal{L}$ *is a finite set of labels (ranged over by* $\ell, \ldots$*);*
- $\mathcal{T} \subseteq Q \times (\mathcal{L} \cup \{\varepsilon\}) \times Q$ *is a set of transitions where* $\varepsilon \notin \mathcal{L}$ *is a distinguished label.*

A finite-state automaton *(FSA) is an LTS whose set of states is finite.*

When the LTS $A = (Q, q_0, \mathcal{L}, \mathcal{T})$ is understood we use the usual notations $s_1 \xrightarrow{\ell} s_2$ for the transition $(s_1, \ell, s_2) \in \mathcal{T}$ and $s_1 \longrightarrow s_2$ when there exists ℓ such that $s_1 \xrightarrow{\ell} s_2$, as well as $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$. We denote as $\mathit{out}(\mathsf{CA}, q)$ the set of transitions from q in A. We occasionally write $q \in A$ and $(q, \alpha, q') \in A$ instead of, respectively, $q \in Q$ and $(q, \alpha, q') \in \mathcal{T}$, and likewise for $\_ \subseteq \_$. We recall standard notions on LTSs.

▶ **Definition 2.2** (Traces and trace equivalence). *A* run *of an LTS* $A = \langle \mathbb{S}, s_0, \mathcal{L}, \mathcal{T} \rangle$ *is a (possibly empty) finite or infinite sequence* $\pi = (s_i \xrightarrow{\ell_i} s_{i+1})_{0 \le i \le n}$ *of consecutive transitions starting at* s_0 *(assume* $n = \infty$ *if the run is infinite). The* trace *(or* word*) of* π *is the concatenation of the labels* $\mathsf{trace}(\pi)$ *of the run* π*, namely* $\mathsf{trace}(\pi) = \ell_0 \cdot \ell_1 \cdots \ell_n$*. As usual,* ε *denotes the identity element of concatenation and the trace of an empty run is* ε*. Function* $\mathsf{trace}(\cdot)$ *extends homomorphically to sets of runs. Also, s-runs and s-traces of* A *are, respectively, runs and traces of* $\langle \mathbb{S}, s, \mathcal{L}, \mathcal{T} \rangle$*. The* language of A *is* $\mathsf{L}(A) = \{\mathsf{trace}(\pi) \mid \pi \text{ is a run of } A\}$*;* A accepts w *if* $w \in \mathsf{L}(A)$ *and* A accepts w from s *if* $w \in \mathsf{L}(\langle \mathbb{S}, s, \mathcal{L}, \mathcal{T} \rangle)$*. LTSs* A *and* B *are* trace equivalent *if* $\mathsf{L}(A) = \mathsf{L}(B)$*.*

Bisimilarity [41] is an equivalence relation on LTSs simpler to prove than trace equivalence which is implied by bisimilarity, and coincides with it for deterministic LTSs.

▶ **Definition 2.3** (Bisimulation). *Let* $A = \langle \mathbb{S}_A, s_{0A}, \mathcal{L}, \mathcal{T}_A \rangle$ *and* $B = \langle \mathbb{S}_B, s_{0B}, \mathcal{L}, \mathcal{T}_B \rangle$ *be two LTSs. A relation* $\mathcal{R} \subseteq (\mathbb{S}_A \times \mathbb{S}_B) \cup (\mathbb{S}_B \times \mathbb{S}_A)$ *is a* (strong) bisimulation *if it is symmetric,* $(s_{0A}, s_{0B}) \in \mathcal{R}$*, and for every pair of states* $(p, q) \in \mathcal{R}$ *and all labels* ℓ*:*

$$\text{if } p \xrightarrow{\ell} p' \text{ then there is } q \xrightarrow{\ell} q' \text{ such that } (p', q') \in \mathcal{R}$$

Relation $\mathcal{R}$ *is a* weak *bisimulation if it is symmetric,* $(s_{0A}, s_{0B}) \in \mathcal{R}$*, and for every pair of states* $(p, q) \in \mathcal{R}$ *and all labels* ℓ*:*

- *if* $p \xrightarrow{\ell} p'$ *with* $\ell \neq \varepsilon$ *then there is a run* $q \xrightarrow{\varepsilon}^{\star} \xrightarrow{\ell} \xrightarrow{\varepsilon}^{\star} q'$ *such that* $(p', q') \in \mathcal{R}$ *and*
- *if* $p \xrightarrow{\varepsilon} p'$ *then there is a run* $q \xrightarrow{\varepsilon}^{\star} q'$ *such that* $(p', q') \in \mathcal{R}$.

If two LTSs are bisimilar then they are also trace equivalent.

A main role in our models is played by *interactions* built on the alphabet:

$$\mathcal{L}_{\text{int}} \triangleq \{\, \mathsf{p} \to \mathsf{q} : \mathsf{m} \mid \mathsf{p} \neq \mathsf{q} \in \mathfrak{P} \quad \text{and} \quad \mathsf{m} \in \mathcal{M} \,\} \quad \text{ranged over by lowercase Greek letters}$$

where $\mathfrak{P}$ and $\mathcal{M}$ are, respectively, sets of participants and of messages. We assume $\mathfrak{P} \cap \mathcal{M} = \emptyset$. An interaction $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ specifies that participant p sends a message (of type) m to participant q and participant q receives m. Hence, by construction, each send is paired with a unique receive and vice versa. In most choreographic models, this forbids to specify message losses, races, and deadlocks. Adopting the terminology of the session type community (see, e.g., [25]),

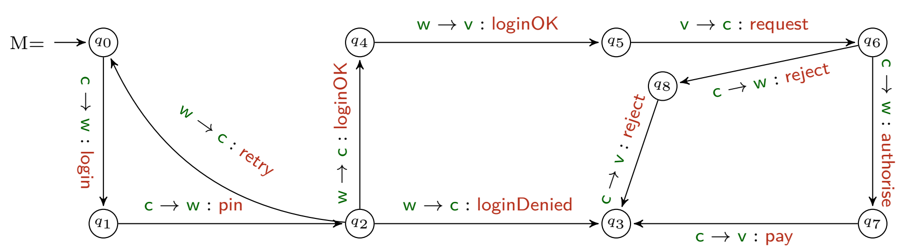
- with *message loss* we mean a send that cannot be matched by a receive; this cannot happen in interactions since $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ specifies both the send and the receive together;
- with *race* we mean a configuration where a receiver non-deterministically interacts with either of two senders (or a sender with either of two receivers), depending on the relative speed of their execution; this cannot happen since an interaction specifies which send is supposed to interact with which receive and vice versa (notably, concurrency can take place without message races, e.g., if participant p sends to participant q and at the same time c sends to d there is no race);
- with *deadlock* we mean a configuration where two or more participants are blocked waiting for one another forming cyclic dependencies (e.g., p is waiting for q which waits for c, which in turns waits for p); this cannot happen either since an interaction specifies which participant has to send and which one has to receive.

All these properties hold by construction in most choreographic models. However, care is needed to ensure that these properties are preserved when moving from the choreographic specification to a distributed implementation. Such analysis has been performed for many choreographic models in the literature (see [25]).

▶ **Definition 2.4** (Choreography automata)**.** *A* choreography automaton *(c-automaton) is an FSA on the alphabet* $\mathcal{L}_{int}$. *Elements of* $\mathcal{L}_{int}^{\star} \cup \mathcal{L}_{int}^{\omega}$ *are* choreography words*, subsets of* $\mathcal{L}_{int}^{\star} \cup \mathcal{L}_{int}^{\omega}$ *are* choreography languages.

The set of participants of a c-automaton is finite; we denote with $\mathcal{P}_{\mathsf{CA}}$ (or simply $\mathcal{P}$ if CA is understood) the set of participants of c-automaton CA. Given $\mathsf{p} \to \mathsf{q} : \mathsf{m} \in \mathcal{L}_{\text{int}}$, we define $\mathsf{ptp}(\mathsf{p} \to \mathsf{q} : \mathsf{m}) \triangleq \{\mathsf{p}, \mathsf{q}\}$ and extend it homorphically to (sets of) transitions. We say that $\alpha, \beta \in \mathcal{L}_{\text{int}}$ are *independent*, written $\alpha \parallel \beta$, if $\mathsf{ptp}(\alpha) \cap \mathsf{ptp}(\beta) = \emptyset$.

▶ **Example 2.5** (OLW's c-automaton)**.** The c-automaton



models the OLW example in § 1. ◇

We now survey communicating systems [5], our formal model of local views.

▶ **Definition 2.6** (Communicating system). *A* communicating finite-state machine *(CFSM) is an FSA on the set* $\mathcal{L}_{act} \triangleq \{\mathsf{pq!m}, \mathsf{pq?m} \mid \mathsf{p}, \mathsf{q} \in \mathfrak{P} \quad and \quad \mathsf{m} \in \mathcal{M}\}$ *of* actions*.*

Action $\mathsf{pq!m}$ *is the send of message* m *from* p *to* q*, while action* $\mathsf{pq?m}$ *is the corresponding receive. The* subjects *of an output and an input action, say* $\mathsf{pq!m}$ *and* $\mathsf{pq?m}$*, are respectively* p *and* q*. A CFSM is* p-local *if all its transitions have labels with subject* p*. A* (communicating) system *is a map* $\mathsf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ *assigning a* p*-local CFSM* M_{p} *to each participant* $\mathsf{p} \in \mathcal{P}$*. We require that* $\mathcal{P} \subseteq \mathfrak{P}$ *is finite and that any participant occurring in a transition of* M_{p} *is in* $\mathcal{P}$*.*

We now introduce the notion of projection from c-automata to systems of CFSMs. Intuitively, projection builds a system aimed at implementing the projected c-automaton. Similar notions in the literature often take the name of endpoint projection (see, e.g., [22, 7]).

▶ **Definition 2.7** (Automata projection). *The* projection $\alpha\!\downarrow_{\mathsf{p}}$ of an interaction α on $\mathsf{p} \in \mathfrak{P}$ *is*

$$(\mathsf{p} \to \mathsf{q} : \mathsf{m})\!\downarrow_{\mathsf{p}} = \mathsf{pq!m}, \ (\mathsf{q} \to \mathsf{p} : \mathsf{m})\!\downarrow_{\mathsf{p}} = \mathsf{qp?m}, \qquad and \qquad \alpha\!\downarrow_{\mathsf{p}} = \varepsilon \text{ for any other label } \alpha$$

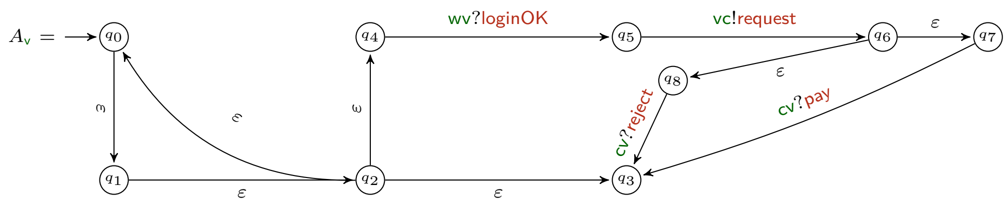
Function $\_\!\downarrow_{\mathsf{p}}$ *extends homomorphically to transitions, runs, and choreography words.*

The projection $\mathsf{CA}\!\downarrow_{\mathsf{p}}$ *of a c-automaton* $\mathsf{CA} = \langle \mathbb{S}, q_0, \mathcal{L}_{int}, \mathcal{T}\rangle$ *on a participant* $\mathsf{p} \in \mathcal{P}$ *is obtained by determinising and minimising up-to language equivalence the* intermediate *CFSM*

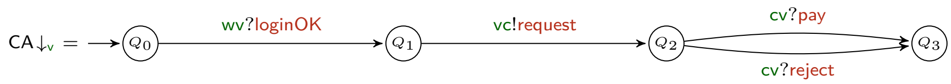
$$A_{\mathsf{p}} = \left\langle \mathbb{S}, q_0, \mathcal{L}_{act}, \left\{ (q \xrightarrow{\alpha\downarrow_{\mathsf{p}}} q') \mid q \xrightarrow{\alpha} q' \in \mathcal{T} \right\} \right\rangle$$

The projection of CA*, written* $\mathsf{CA}\!\downarrow$*, is the communicating system* $(\mathsf{CA}\!\downarrow_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$*.*

▶ **Example 2.8** (Projecting OLW). We instantiate here projection on the c-automaton for the OLW protocol described in Ex. 2.5. In particular, the intermediate CFSM A_{v} is



the determinisation of which yields the following CFSM $\mathsf{CA}\!\downarrow_{\mathsf{v}}$ for the vendor participant



Noteworthy, due to determinisation, states of the projection correspond to (not necessarily disjoint) sets of states of the starting c-automaton. Indeed, in $\mathsf{CA}\!\downarrow_{\mathsf{v}}$ we have $Q_0 = \{q_0, q_1, q_2, q_3, q_4\}$, $Q_1 = \{q_5\}$, $Q_2 = \{q_6, q_7, q_8\}$, and $Q_2 = \{q_3\}$. ◇

We present below the semantics of communicating systems. We consider a synchronous semantics. Essentially, a system can execute an interaction $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ if two of its participants can provide complementary actions $\mathsf{pq!m}$ and $\mathsf{pq?m}$ (while the others do not move), and can take an ϵ action if one of its participant can do it (while the others do not move).

▶ **Definition 2.9** (Semantics of communicating systems). *Let* $\mathsf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ *be a communicating system where* $M_{\mathsf{p}} = \langle \mathbb{S}_{\mathsf{p}}, q_{0\mathsf{p}}, \mathcal{L}_{act}, \mathcal{T}_{\mathsf{p}}\rangle$ *for each participant* $\mathsf{p} \in \mathcal{P}$*.*

A configuration of S *is a map* $s = (q_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ *assigning a* local state $q_{\mathsf{p}} \in \mathbb{S}_{\mathsf{p}}$ *to each* $\mathsf{p} \in \mathcal{P}$*. The* semantics *of* S *is the c-automaton* $[\![\mathsf{S}]\!] = \langle \mathbb{S}, s_0, \mathcal{L}_{int}, \mathcal{T}\rangle$ *where*

- $\mathbb{S}$ *is the set of configurations of* S*, as defined above, and* $s_0 : \mathsf{p} \mapsto q_{0\mathsf{p}}$ *for each* $\mathsf{p} \in \mathcal{P}$ *is the* initial *configuration of* $\mathbb{S}$
- $\mathcal{T}$ *is the set of transitions*
  - $s_1 \xrightarrow{\mathsf{p} \to \mathsf{q}:\mathsf{m}} s_2$ *such that*
    - $s_1(\mathsf{p}) \xrightarrow{\mathsf{pq!m}} s_2(\mathsf{p}) \in \mathcal{T}_{\mathsf{p}}$ *and* $s_1(\mathsf{q}) \xrightarrow{\mathsf{pq?m}} s_2(\mathsf{q}) \in \mathcal{T}_{\mathsf{q}}$*, and*
    - *for all* $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}, \mathsf{q}\}$, $s_1(\mathsf{x}) = s_2(\mathsf{x})$
  - $s_1 \xrightarrow{\varepsilon} s_2$ *such that* $s_1(\mathsf{p}) \xrightarrow{\varepsilon} s_2(\mathsf{p}) \in \mathcal{T}_{\mathsf{p}}$*, and for all* $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}\}$, $s_1(\mathsf{x}) = s_2(\mathsf{x})$.

3 Flexible Choreography Automata

We now introduce a theory of c-automata enabling faithful realisations, which is formalised as language equivalence between a c-automaton and the semantics of its projection as proved in Corollary 3.16. However, not all c-automata can be faithfully realised, hence we need to restrict to *well-formed* c-automata. Well-formedness is defined as the conjunction of two properties, *well-sequencedness* and *well-branchedness*. Both these properties are inspired from [2]. However, well-branchedness is generalised to allow participants to act on some of the scenarios specified by the c-automaton only upon request from other participants. We call this feature *selective participation*, since a participant may act on a branch only if selected to be involved by some other participant. This is disallowed in many choreographic formalisms (e.g., [21, 43, 7]), including choreography automata [2]. On the other hand, well-sequencedness is strengthened since the formulation in [2] is not enough to ensure faithful realisations. We start by defining concurrent transitions, exploited in the definition of well-sequencedness.

▶ **Definition 3.1** (Concurrent transitions). *Two consecutive transitions* $q \xrightarrow{\alpha} q' \xrightarrow{\beta} q''$ *are* concurrent *if there is* q''' *such that* $q \xrightarrow{\beta} q''' \xrightarrow{\alpha} q''$.

Essentially, two transitions are concurrent if they give rise to a commuting diamond.

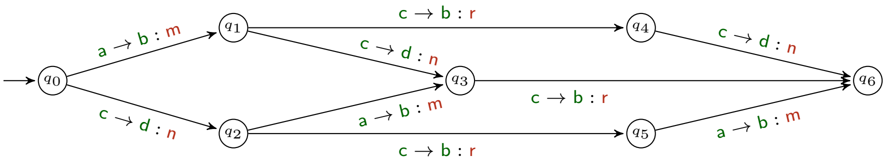
▶ **Definition 3.2** (Well-sequencedness). *A c-automaton* CA *is* well-sequenced *if for each two consecutive transitions* $q \xrightarrow{\alpha} q' \xrightarrow{\beta} q''$ *either*

(a) $\alpha \not\parallel \beta$*, i.e.,* α *and* β *are not independent (hence* $\mathsf{ptp}(\alpha) \cap \mathsf{ptp}(\beta) \neq \emptyset$*), or*

(b) *there is* q''' *such that* $q \xrightarrow{\beta} q''' \xrightarrow{\alpha} q''$ *(i.e., the transitions are concurrent); furthermore for each transition* $q''' \xrightarrow{\gamma} q''''$, $\gamma \parallel \alpha$ *and* $\gamma \parallel \beta$.

Intuitively, well-sequencedness forces the explicit representation of concurrency among interactions with disjoint sets of participants as commuting diamonds. The second part of clause (b) in Def. 3.2 rules out the entanglement of choices with commuting diamonds, while enabling to compose an arbitrary number of independent actions. This condition, absent in [2], does not allow them to enforce faithful realisations as shown in the next example.

▶ **Example 3.3.** Consider the c-automaton below.

$\mathsf{CA} =$ 

In $\mathsf{CA}\!\downarrow$, participant c can immediately send r to b, since it is not involved in transition $q_0 \xrightarrow{\mathsf{a}\to\mathsf{b}:\mathsf{m}} q_1$. Similarly, b can immediately receive r from c, since it is not involved in transition $q_0 \xrightarrow{\mathsf{c}\to\mathsf{d}:\mathsf{n}} q_2$. Thus, a transition with label $\mathsf{c} \to \mathsf{b} : \mathsf{r}$ is enabled in the initial configuration of the semantics of $\mathsf{CA}{\downarrow}$. However, no transition with the same label is enabled in the initial state of CA, hence the implementation is not faithful. ◇

The following auxiliary concepts are instrumental in the definition of well-branchedness (cf. Def. 3.7). Given a word w, $\mathsf{pref}(w)$ denotes the set of its prefixes.

▶ **Definition 3.4** (Full awareness). *Let (π_1, π_2) be a pair of q-runs of a c-automaton* CA. *Participant $\mathsf{p} \in \mathit{ptp}(\pi_1) \cap \mathit{ptp}(\pi_2)$ is* fully aware *of (π_1, π_2) if there are $\alpha_1 \neq \alpha_2 \in \mathcal{L}_{int}$ such that $\mathsf{p} \in \mathit{ptp}(\alpha_1) \cap \mathit{ptp}(\alpha_2)$ and*

1. *either α_h is the first interaction in $\mathsf{L}(\pi_h)$ for $h = 1, 2$*
2. *or for $h \in \{1, 2\}$ there is a proper prefix $\hat{\pi}_i$ of π_i such that $\mathit{trace}(\hat{\pi}_1{\downarrow_\mathsf{p}}) = \mathit{trace}(\hat{\pi}_2{\downarrow_\mathsf{p}})$, the partners of p in α_h are fully aware of $(\hat{\pi}_1, \hat{\pi}_2)$, $\mathit{trace}(\hat{\pi}_h)\alpha_h \in \mathit{pref}(\mathit{trace}(\pi_h))$, and α_h does not occur on π_{3-h}.*

Intuitively, a participant p is fully aware of two q-runs when able to ascertain which branch has been taken. This happens either when p itself chooses (1), or when p is informed of the choice by interacting with some other participant already fully aware of the q-runs (2).

▶ **Example 3.5** (Full awareness in OLW). Let us consider the runs $\pi_1 = q_2 \xrightarrow{\mathsf{w}\to\mathsf{c}:\mathsf{loginOK}} q_4 \xrightarrow{\mathsf{w}\to\mathsf{v}:\mathsf{loginOK}} q_5$ and $\pi_2 = q_2 \xrightarrow{\mathsf{w}\to\mathsf{c}:\mathsf{loginDenied}} q_3$ of the OLW c-automaton M in Ex. 2.5. Both w and c are fully-aware of (π_1, π_2) since they occur in the first interaction in both the runs (Def. 3.4(1)). Participant v is not fully-aware of (π_1, π_2) since it occurs on π_1 only.

Take now the runs $\pi_3 = q_6 \xrightarrow{\mathsf{c}\to\mathsf{w}:\mathsf{reject}} q_8 \xrightarrow{\mathsf{c}\to\mathsf{v}:\mathsf{reject}} q_3$ and $\pi_4 = q_6 \xrightarrow{\mathsf{c}\to\mathsf{w}:\mathsf{authorise}} q_7 \xrightarrow{\mathsf{c}\to\mathsf{v}:\mathsf{pay}} q_3$ in M. As before, both participants w and c are fully-aware of (π_3, π_4) since they occur in the first interaction in both the runs. Participant v is fully-aware of (π_3, π_4) as well, since its partner c is fully-aware of $(q_6 \xrightarrow{\mathsf{c}\to\mathsf{w}:\mathsf{reject}} q_8, q_6 \xrightarrow{\mathsf{c}\to\mathsf{w}:\mathsf{authorise}} q_7)$. ◇

To establish well-branchedness of a c-automaton we have to ensure that for each choice, namely for each state with (at least) two non-independent outgoing transitions, and each participant p, if p has to take different actions in the branches starting from the two transitions, then p is fully-aware of the taken branch. In principle, such a condition should be checked on all pairs of coinitial paths. However, this would lead to redundant checks, hence below we borrow from [2] the notion of q-spans, namely pairs of paths from q on which we will perform the check. Essentially, we have to handle choices with loops on some branches and we have to consider "long-enough" branches. More precisely, a q-run in a c-automaton CA is a *pre-candidate q-branch* if each of its cycles has at most one occurrence within the whole run (i.e., if π' is a q'-run included in π and ending in q', then π' has exactly one occurrence in π); a *candidate q-branch* is a maximal pre-candidate q-branch with respect to the prefix order.

▶ **Definition 3.6** (q-span). *A pair (π, π') of pre-candidate q-branches of* CA *is a* q-span *if*

1. *either π and π' are cofinal, with no common node but q and the last one;*
2. *or π and π' are candidate q-branches with no common node but q;*
3. *or π and π' are a candidate q-branch and a loop on q with no other common nodes.*

We can now introduce well-branchedness.

▶ **Definition 3.7** (Well-branchedness). *A c-automaton* CA *is* well-branched *if it is deterministic and for each of its states q there is a partition $T_1, \ldots, T_k$ of $\mathit{out}(\mathsf{CA}, q)$ such that*

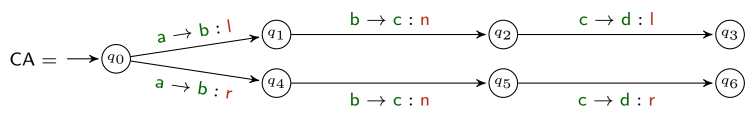
- *for all $1 \leq i \neq j \leq k$, $\mathit{ptp}(T_i) \cap \mathit{ptp}(T_j) = \emptyset$ and for each $q \xrightarrow{\alpha_i} q_i \in T_i$, $q \xrightarrow{\alpha_j} q_j \in T_j$ there exists q' such that $q_i \xrightarrow{\alpha_j} q'$ and $q_j \xrightarrow{\alpha_i} q'$*

- *for all* $1 \leq i \leq k$, $\bigcap_{t \in T_i} \mathit{ptp}(t) \neq \emptyset$ *and for all* $\mathsf{p} \in \mathit{ptp}(\mathsf{CA}) \setminus \bigcap_{t \in T_i} \mathit{ptp}(t)$ *and q-span* (π_1, π_2) *starting from transitions in* T_i*, if* $\pi_1 \downarrow_{\mathsf{p}} \neq \pi_2 \downarrow_{\mathsf{p}}$ *then either* p *is fully aware of* (π_1, π_2) *or there is* $i \in \{1, 2\}$ *such that* $\mathsf{p} \notin \mathit{ptp}(\pi_i)$ *and*
  1. *the first transition in* π_{3-i} *involving* p *is with a fully aware participant of* (π_1, π_2) *and*
  2. *for all runs* π' *such that* $\pi_i \pi'$ *is a candidate q-branch of* CA *the first transition in* π' *involving* p *is with a participant which is fully aware of* (π_1, π_2)*.*

Intuitively, a c-automaton is well-branched if for any state with multiple outgoing transitions (both clauses in Def. 3.7 trivially hold when $\mathit{out}(\mathsf{CA}, q)$ is empty or a singleton) , we can group them in equivalence classes. Transitions in different classes are concurrent, hence they give rise to commuting diamonds. Transitions in the same class are choices: one participant, belonging to all the (initial) transitions, makes the choice, and any other participant p is either fully aware of the q-runs or it is inactive in some branch π_i (condition $\mathsf{p} \notin \mathsf{ptp}(\pi_i)$). In the last case, p has to interact with a fully aware partner (i) on each continuation π' (if any) of π_i as well as (ii) inside the other branch, π_{3-i}. Intuitively, (i) is necessary to make p aware of when the choice is fully completed and (ii) on whether the branch on which p needs to act has been taken. At the price of increasing the technical complexity, the second clause in Def. 3.7 can be relaxed. Indeed, right now it requires a participant p, occurring in one branch only, to interact (both in the branch where it occurs and in the continuations after the merge of the two branches) with a fully-aware participant. We could instead allow p to interact with a chain of other participants occurring only in the same branch, and such that the last participant in the chain interacts with a fully-aware participant.

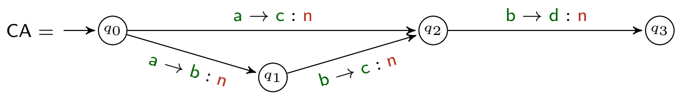
▶ **Example 3.8** (OLW is well-branched)**.** Let us show that the c-automaton in Ex. 2.5 is well-branched. The only states for which well-branchedness is not trivial are q_2 and q_6 (the others have at most one outgoing transition). In both the cases we have a single equivalence class where w and c are in all the first transitions; hence they are both fully-aware in all the possible spans. Let us check the condition for v. Let us consider q_6. There is one q_6-span, with branches with states q_6, q_8, q_3 and q_6, q_7, q_3, which fits case 1 in Def. 3.6. As discussed in Ex. 3.5, in this q_6-span v is fully-aware, hence the condition is satisfied. Let us now consider q_2. Here we have a loop with states q_2, q_0, q_1, q_2, a candidate q_2-branch with states q_2, q_3, and two candidate q_2-branches with a common prefix (states q_2, q_4, q_5, q_6) and two continuations (states q_6, q_8, q_3 and q_6, q_7, q_3). Any combination of the self-loop with the candidate q_2-branches fit in case 3 in Def. 3.6, while the pairings of the first candidate q_2-branch with any of the others fit in case 1 in Def. 3.6. In the q_2-spans above v occurs only in the one with two continuations. Since there it interacts with c which is fully-aware, condition 1 in Def. 3.7 holds. Condition 2 holds trivially, since the branches join only in state q_3 which has no outgoing transitions. ◇

▶ **Example 3.9** (Non well-branched c-automata)**.** Consider the c-automaton below.



Here, c is not fully-aware since it interacts with b (which is fully-aware) receiving the same message on both the branches. Hence, its first different interactions are with d, which is not fully-aware. Indeed, d gets different messages, but from c which is not fully aware either. Thus, c and d can decide, e.g., to take the lower branch even if a and b took the upper one, thus producing a trace $\mathsf{a} \to \mathsf{b} : \mathsf{l} \cdot \mathsf{b} \to \mathsf{c} : \mathsf{n} \cdot \mathsf{c} \to \mathsf{d} : \mathsf{r}$ not part of the language of CA. ◇

▶ **Example 3.10** (Non well-branchedness with selective participation)**.** Consider the c-automaton:



Here, b occurs in the bottom branch only, interacting with a which is fully-aware, as required. However, after the merge of the two branches, b interacts with d which is not fully aware, thus violating condition 2 in Def. 3.7. Indeed the interaction $\mathsf{b} \to \mathsf{d} : \mathsf{n}$ is enabled since the initial configuration, against the prescription of CA. ◇

▶ **Definition 3.11** (Well-formedness)**.** *A c-automaton* CA *is* well-formed *if it is both well-sequenced and well-branched.*

Well-formed c-automata enjoy relevant properties. First, for each well-formed c-automaton the semantics of the projection is bisimilar to the starting c-automaton.

▶ **Lemma 3.12.** *Let* CA *be a well-formed c-automaton and A_{p} and A_{q} be the intermediate CFSM for two participants* p *and* q *of* CA *(cf. Def. 2.7). If*

$$q \xrightarrow{\varepsilon}{}^{\star} q_{\mathsf{p}} \xrightarrow{\mathsf{pq!m}} q'_{\mathsf{p}} \text{ in } A_{\mathsf{p}} \qquad \text{and} \qquad q \xrightarrow{\varepsilon}{}^{\star} q_{\mathsf{q}} \xrightarrow{\mathsf{pq?m}} q'_{\mathsf{q}} \text{ in } A_{\mathsf{q}}$$

then the state q of CA *has an outgoing transition with label $\mathsf{p} \to \mathsf{q} : \mathsf{m}$.*

Proof. Let the two runs for p and q in the intermediate CFSMs be the projections of runs π_{p} and π_{q} in CA. We have two cases, depending on whether $\pi_{\mathsf{p}} = \pi_{\mathsf{q}}$.

If $\pi_{\mathsf{p}} = \pi_{\mathsf{q}}$ then the last transition of π_{p} is concurrent to all the previous ones. Indeed, the previous transitions are projected to ε both on p and on q, hence neither p nor q can occur in the label. Thus, by well-sequencedness, the state q has a transition with label $\mathsf{p} \to \mathsf{q} : \mathsf{m}$.

Assume $\pi_{\mathsf{p}} \neq \pi_{\mathsf{q}}$. By construction,

$$\pi_{\mathsf{p}} = \pi \; t_{\mathsf{p}} \; \pi'_{\mathsf{p}} \; q_{\mathsf{p}} \xrightarrow{\mathsf{p \to q:m}} q'_{\mathsf{p}} \qquad \text{and} \qquad \pi_{\mathsf{q}} = \pi \; t_{\mathsf{q}} \; \pi'_{\mathsf{q}} \; q_{\mathsf{q}} \xrightarrow{\mathsf{p \to q:m}} q'_{\mathsf{q}}$$

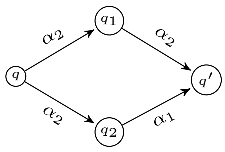
for a q-run π, two transitions t_{p} and t_{q}, two runs π'_{p} and π'_{q}; observe that the ending state of π, say $\hat{q}$, is also the source state of t_{p} and t_{q} while π'_{p} (resp. π'_{q}) ends in q_{p} (resp. q_{q}).

Note that $\mathsf{p} \notin \mathsf{ptp}(\pi \; t_{\mathsf{p}} \; \pi'_{\mathsf{p}})$ and $\mathsf{q} \notin \mathsf{ptp}(\pi \; t_{\mathsf{q}} \; \pi'_{\mathsf{q}})$. If $\mathsf{q} \notin \mathsf{ptp}(\pi \; t_{\mathsf{p}} \; \pi'_{\mathsf{p}})$ or $\mathsf{p} \notin \mathsf{ptp}(\pi \; t_{\mathsf{q}} \; \pi'_{\mathsf{q}})$ then the thesis immediately follows by well-sequencedness as before. Therefore we can assume $\mathsf{p} \in \mathsf{ptp}(\pi \; t_{\mathsf{q}} \; \pi'_{\mathsf{q}})$ ($\mathsf{q} \in \mathsf{ptp}(\pi \; t_{\mathsf{p}} \; \pi'_{\mathsf{p}})$ could be assumed as well).

Consider now the two $\hat{q}$-runs $\pi_1 = t_{\mathsf{p}} \; \pi'_{\mathsf{p}}$ and $\pi_2 = t_{\mathsf{q}} \; \pi'_{\mathsf{q}}$. By well-branchedness, there is a partition of $out(\mathsf{CA}, \hat{q})$ satisfying the conditions of Def. 3.7. Then t_{p} and t_{q} cannot belong to the same equivalence class of such partition since neither p nor q are fully aware of (π_1, π_2) and the first interaction of p on π_1 is with q. Hence, $t_{\mathsf{p}} = \hat{q} \xrightarrow{\alpha_1} q_1$ and $t_{\mathsf{q}} = \hat{q} \xrightarrow{\alpha_2} q_2$ necessarily belong to different equivalence classes. Therefore, by Def. 3.7, there is a state $\hat{q}'$ such that $q_1 \xrightarrow{\alpha_2} \hat{q}'$ and $q_2 \xrightarrow{\alpha_1} \hat{q}'$. Hence, the transitions of π'_{p} and those of π'_{q} form commuting diamonds and therefore there is a q-run in CA where q does not occur and all the transitions involving p in π'_{q} follow a transition with label $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ (easily by induction on the length of π'_{p} and π'_{q}). The thesis then follows since, as before, the transition labelled by $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ commutes with any preceding transition by well-sequencedness. ◀

▶ **Lemma 3.13.** *Let* CA *be a well-formed c-automaton and, for $i \in \{1, 2\}$, $t_i = q \xrightarrow{\alpha_i} q'_i$ the first two transitions of a q-span (π_1, π_2) in* CA *such that t_1 and t_2 are concurrent. Then either α_1 occurs on $\mathsf{trace}(\pi_2)$ or for each run $\pi_2 \; \pi$ in* CA*, α_1 occurs in $\mathsf{trace}(\pi)$.*

Proof. The proof is by coinduction. By well-branchedness there is a state q' such that



If q' occurs in π_2 we are done. Otherwise, by well-sequencedness (cf. Def. 3.2(b)), $q_1 \xrightarrow{\alpha_2} q'$ is concurrent with the first transition of π_2 and the thesis follows by the coinductive hypothesis. ◀

▶ **Proposition 3.14.** *Let* CA *be a well-formed c-automaton,* $\mathsf{p}, \mathsf{q} \in \mathcal{P}_{\mathsf{CA}}$*, and* $\alpha \in \mathcal{L}_{int}$*. If* $w_{\mathsf{p}}\alpha$*,* $w_{\mathsf{q}}\alpha$*,* $w \in \mathsf{L}(\mathsf{CA})$ *are three words such that* $w_{\mathsf{p}} \downarrow_{\mathsf{p}} = w \downarrow_{\mathsf{p}}$ *and* $w_{\mathsf{q}} \downarrow_{\mathsf{q}} = w \downarrow_{\mathsf{q}}$ *then* $w\alpha \in \mathsf{L}(\mathsf{CA})$*.*

Proof. The proof is by case analysis on the form of the words.

If $w = w_{\mathsf{p}} = w_{\mathsf{q}} = \varepsilon$ then the thesis follows trivially. Otherwise at least one among those words is not empty. Let $\alpha = \mathsf{p} \to \mathsf{q} : \mathsf{m}$ and, towards a contradiction, suppose $w\alpha \notin \mathsf{L}(\mathsf{CA})$.

Let π, π_{p}, and π_{q} be runs of CA such that

$$w = \mathsf{trace}(\pi), \qquad w_{\mathsf{p}}\ \alpha = \mathsf{trace}(\pi_{\mathsf{p}}), \qquad \text{and} \qquad w_{\mathsf{q}}\ \alpha = \mathsf{trace}(\pi_{\mathsf{q}}) \tag{1}$$

respectively; and let $\hat{q}$ be the state from where at least two of the three runs in (1) start to become different.

We first consider the case where one of the words is empty.

- If $w = \varepsilon$, we show that $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ is enabled in the initial state contradicting our assumption that $w\ \alpha = \mathsf{p} \to \mathsf{q} : \mathsf{m} \notin \mathsf{L}(\mathsf{CA})$. From the hypothesis, p does not occur in π_{p} and q does not occur in π_{q}. If p also does not occur in π_{q} then we can commute $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ and the thesis follows. Otherwise, either p or q should be fully aware, but this is not possible since each of them occurs in one of the runs only.
- If $w_{\mathsf{p}} = \varepsilon$ then there is a transition labelled with the interaction $\alpha = \mathsf{p} \to \mathsf{q} : \mathsf{m}$ from the initial state of CA, say q_0. Such transition cannot be concurrent with the first transition of π (otherwise, by Lemma 3.13, it would occur on all continuations of π contradicting our assumption that $w\ \alpha \notin \mathsf{L}(\mathsf{CA})$). Hence, by well-branchedness, the initial transitions of π and π_{p} must belong to a same partition of $out(\mathsf{CA}, q_0)$. Again, by well-branchedness, q must be fully aware of (π_{p}, π). The only possibility is that q occurs in the initial transitions of π and π_{q}. The condition $w{\downarrow_{\mathsf{q}}} = w_{\mathsf{q}}{\downarrow_{\mathsf{q}}}$ implies that π and π_{q} must have the same initial transition and therefore they share a non-empty prefix; let $\hat{q}'$ be the last state of the longest of such prefixes and $t \neq t_{\mathsf{q}}$ be the first transitions from $\hat{q}'$ on π and π_q respectively.
  - If t and t_{q} are concurrent then by well-branchedness the last transition of π_{q}, say t_{p} (note that the label of t_{p} is $\alpha = \mathsf{p} \to \mathsf{q} : \mathsf{m}$) must be concurrent with the last transition of π, hence there is a run $\pi\ t_{\mathsf{p}}$ in CA, which violates our assumption that $w\ \alpha \in \mathsf{L}(\mathsf{CA})$.
  - Otherwise q is not fully aware of the choice because otherwise $t = t_{\mathsf{q}}$. Hence by well-branchedness p occurs on π_{p} contrary to the assumption that $w{\downarrow_{\mathsf{p}}} = \varepsilon$.

  In all cases we derive a contradiction, hence $w\ \alpha \in \mathsf{L}(\mathsf{CA})$.
- If w_{q} is empty then the proof is as in the previous case.

For the case that none of the words is empty (i.e., $w_{\mathsf{p}} \neq \varepsilon$, $w_{\mathsf{q}} \neq \varepsilon$, and $w \neq \varepsilon$) we analyse how the runs branch.

Let $\hat{q}$ be the first state on π after which π and π_{p} start to diverge along two different transitions t of π and t_{p} of π_{p}.

Then t and t_{p} must be in the same equivalence class of the partition of $out(\mathsf{CA}, \hat{q})$ otherwise the last transition of π_{p}, say t' (which is labelled with α) would commute with all the transitions of π; hence $\pi\ t'$ would be a run in CA contrary to our assumption that $w\ \alpha \notin \mathsf{L}(\mathsf{CA})$. (Such partition should also include a transition t_{q} on π_{q} for the same reason.)

The projection on q of π and π_{q} differ. Assume the number of interactions involving p in w, which is the same as those in w_{p}, is not 0. Then, p should be fully aware of (π, π_{p}). However, if t and t_{p} involve p then $t = t_{\mathsf{p}}$ by the hypothesis that $w \downarrow_{\mathsf{p}} = w_{\mathsf{p}} \downarrow_{\mathsf{p}}$. Hence condition (1) of Def. 3.4 does not apply. Condition (2) of Def. 3.4 does not apply either since there is an action on one side only after equal traces. Hence we have a contradiction if the number of interactions involving p is not 0.

Now, let the number of interactions involving p be 0. Hence, p should be fully aware of (π, π_{p}). Using the same reasoning on (w, w_{p}), however, the number of occurrences of p in w is 0, hence it cannot be fully aware of $(w, w_{\mathsf{q}}\ \mathsf{p} \to \mathsf{q} : \mathsf{m})$. Again we have a contradiction, hence this case can never happen. ◀

▶ **Theorem 3.15.** CA *is bisimilar to* $[\![\mathsf{CA}\downarrow]\!]$ *for any well-formed c-automaton* CA.

Proof. Let $\mathsf{CA} = \langle Q, q_0, \mathcal{L}_{\text{int}}, \mathcal{T} \rangle$ and let $\mathbb{S}$ be the set of configurations of $[\![\mathsf{CA}\downarrow]\!]$. We show by coinduction that the relation

$$\mathcal{R} = \{(q, s) \in Q \times \mathbb{S} \mid q \in s(\mathsf{p}) \text{ for each } \mathsf{p} \in \mathcal{P}_{\mathsf{CA}}\}$$

is a bisimulation. (Recall that, due to determinisation and minimisation, for each $s \in \mathbb{S}$ and each participant p, $s(\mathsf{p})$ is a subset of Q). Since bisimulation implies trace equivalence, we also have that corresponding elements are reachable via the same trace.

Let $(q, s) \in \mathcal{R}$ and consider a challenge from $[\![\mathsf{CA}\downarrow]\!]$, namely $s \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} s'$. By definition of synchronous semantics, $s(\mathsf{p}) \xrightarrow{\mathsf{pq!m}} s'(\mathsf{p})$, $s(\mathsf{q}) \xrightarrow{\mathsf{pq?m}} s'(\mathsf{q})$ and $s(\mathsf{x}) = s'(\mathsf{x})$ for each $\mathsf{x} \notin \{\mathsf{p}, \mathsf{q}\}$. By definition of determinisation, there are $\tilde{q}_{\mathsf{p}} \in s(\mathsf{p})$ and $\tilde{q}_{\mathsf{q}} \in s(\mathsf{q})$ such that

$$\tilde{q}_{\mathsf{p}} \xrightarrow{\varepsilon}{}^{\star} q_{\mathsf{p}} \xrightarrow{\mathsf{pq!m}} q'_{\mathsf{p}} \text{ in } A_{\mathsf{p}} \qquad \text{and} \qquad \tilde{q}_{\mathsf{q}} \xrightarrow{\varepsilon}{}^{\star} q_{\mathsf{q}} \xrightarrow{\mathsf{pq?m}} q'_{\mathsf{q}} \text{ in } A_{\mathsf{q}}$$

If $q = \tilde{q}_{\mathsf{p}} = \tilde{q}_{\mathsf{q}}$ then $q \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'$ by Lemma 3.12. Otherwise, since $(q, s) \in \mathcal{R}$, each run in $[\![\mathsf{CA}\downarrow]\!]$ that reaches s has, for all participants $\mathsf{x} \in \mathcal{P}$, a corresponding run that reaches q in A_{x}. Now consider a word w that reaches q and matching words w_{p} and w_{q} obtained by lifting to CA runs reaching $\tilde{q}_{\mathsf{p}}$ and $\tilde{q}_{\mathsf{q}}$ in the respective auxiliary CFSMs. By construction, they are in the hypothesis of Prop. 3.14, hence $q \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'$ also in this case. We now show that $q' \in s'(\mathsf{x})$ for each x. By definition, for each x, in the intermediate CFSM A_{x} we have $q_{\mathsf{x}} \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}\downarrow_{\mathsf{x}}} q'_{\mathsf{x}}$ for some $q_{\mathsf{x}} \in s(\mathsf{x})$ and some q'_{x}.

- If $\mathsf{x} \notin \{\mathsf{p}, \mathsf{q}\}$ then $\mathsf{p} \to \mathsf{q} : \mathsf{m}\downarrow_{\mathsf{x}} = \varepsilon$ hence $q_{\mathsf{x}} \in s(\mathsf{x})$ implies $q'_{\mathsf{x}} \in s'(\mathsf{x})$ as required since $s(\mathsf{x})$ contains the ε-closure of its elements by construction.
- If $\mathsf{x} = \mathsf{p}$ then $\mathsf{p} \to \mathsf{q} : \mathsf{m}\downarrow_{\mathsf{x}} = \mathsf{pq!m}$. Thus $q_{\mathsf{x}} \xrightarrow{\mathsf{pq!m}} q'_{\mathsf{x}}$ and, as shown above, $s(\mathsf{p}) \xrightarrow{\mathsf{pq!m}} s'(\mathsf{p})$. Since $q_{\mathsf{x}} \in s(\mathsf{p})$ and CA is deterministic then $q'_{\mathsf{x}} \in s'(\mathsf{x})$ since $s'(\mathsf{x})$ is the ε-closure of q'_{x} by construction.
- If $\mathsf{x} = \mathsf{q}$ then the reasoning is analogous to the previous case.

Let us now consider a challenge from CA, namely $q \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'$. By definition of projection and ε-closure, $s(\mathsf{p}) \xrightarrow{\mathsf{pq!m}} s'(\mathsf{p})$, $s(\mathsf{q}) \xrightarrow{\mathsf{pq?m}} s'(\mathsf{q})$, and $s(\mathsf{x}) = s'(\mathsf{x})$ for each $\mathsf{x} \notin \{\mathsf{p}, \mathsf{q}\}$. By definition of synchronous semantics $s \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} s'$ as desired. For each participant y from $q \in s(\mathsf{y})$ we get $q' \in s'(\mathsf{y})$, hence the thesis follows. ◀

An immediate consequence of Thm. 3.15 is that the language of a well-formed c-automaton coincides with the language of the semantics of its projection.

▶ **Corollary 3.16.** $L(\mathsf{CA}) = L([\![\mathsf{CA}\!\downarrow]\!])$ *for any well-formed c-automaton* CA.

Proof. From Thm. 3.15 given that bisimulation implies trace equivalence. ◀

We now show that projections of well-formed c-automata do not deadlock. To this end, we need to extend CFSMs with a concept of final state. Intuitively, a state is final in the projection on some participant p of a given c-automaton CA iff one of the corresponding states of CA (remember that states of the projection are sets of states of CA) has an outgoing maximal path along with p is not involved. Formally:

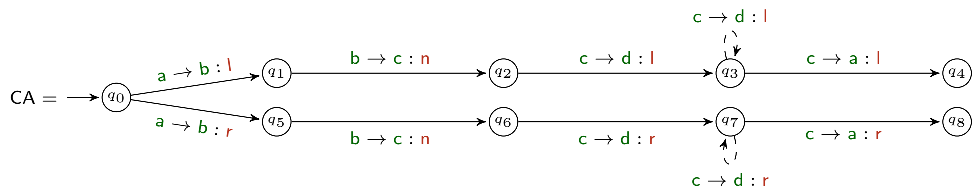
▶ **Definition 3.17** (Final states in projected CFSMs). *Let* CA *be a c-automaton and* p *one of its participants. A state* Q *of* $\mathsf{CA}\!\downarrow_{\mathsf{p}}$ *is* final *if in* CA *there is* $q \in Q$ *and a candidate* q*-branch such that* $\mathsf{p} \notin \mathit{ptp}(\pi)$*.*

▶ **Definition 3.18** (Deadlock freedom). *The projection of a c-automaton is* deadlock-free *if for each of its reachable configurations* s *either* s *has an outgoing transition or, for each participant* p*,* $s(\mathsf{p})$ *is final.*

▶ **Theorem 3.19** (Projections of well-formed c-automata are deadlock-free). *Let* CA *be a well-formed c-automaton. Then* $\mathsf{CA}\!\downarrow$ *is deadlock-free.*

Proof. Let us assume, towards a contradiction, that $\mathsf{CA}\!\downarrow$ is not deadlock-free. Then there is a reachable configuration s in $[\![\mathsf{CA}\!\downarrow]\!]$ with no outgoing transition and there exists a participant p such that $s(\mathsf{p})$ is not final. Then, by Def. 3.17, for each $q \in s(\mathsf{p})$ and each candidate q-branch π in CA, $\mathsf{p} \in \mathsf{ptp}(\pi)$. From the proof of Thm. 3.15, s is bisimilar to one such q; hence, s should answer the challenge from the first action of π, hence it has an outgoing transition against the hypothesis. ◀

▶ **Example 3.20** (C-automaton with deadlock). Consider the c-automaton



obtained by adding the transitions from states q_3 and q_7 to the one in Ex. 3.9. Disregard the dashed transitions. If, as discussed in Ex. 3.9, c and d decide to take the bottommost branch while a and b take the uppermost one, we can reach a configuration s where c wants to send r to a, but a is only willing to take l. Hence, no transition is possible and we have a deadlock. Due to Thm. 3.19 this is possible only since the c-automaton is not well-formed. ◇

We can refine the result above by focusing on a single participant.

▶ **Definition 3.21** (Lock freedom). *The projection of a c-automaton is* lock-free *if for each of its reachable configurations* s *and each participant* p*, either* $s(\mathsf{p})$ *is final or* s *has at least an outgoing transition and for each candidate* s*-branch* π *we have* $\mathsf{p} \in \mathit{ptp}(\pi)$*.*

Lock freedom is strictly stronger than deadlock freedom. Indeed, each configuration s and a participant p such that $s(\mathsf{p})$ is not final has an outgoing transition, hence it is not a deadlock. However, there are systems which are deadlock-free but not lock-free, as discussed below.

▶ **Example 3.22** (C-automaton with locks (but no deadlock)). Consider again the c-automaton from Ex. 3.20, including the dashed self-loops. There is now no deadlock, since the configuration s has an outgoing transition, namely a self-loop involving c and d. However, s is a lock for a. Indeed, it is not final for a, yet a does not take part in the branch corresponding to the execution of the self-loop.

▶ **Theorem 3.23** (Projections of well-formed c-automata are lock-free). *Let* CA *be a well-formed c-automaton. Then* CA↓ *is lock-free.*

Proof. Let us assume, towards a contradiction, that CA↓ is not lock-free. Then there is a reachable configuration s in $[\![\mathsf{CA}\!\downarrow]\!]$ and a participant p such that $s(\mathsf{p})$ is not final and either there is no outgoing transition or there is a candidate s-branch π with $\mathsf{p} \notin \mathsf{ptp}(\pi)$. In the first case the configuration is a deadlock and we have a contradiction from Thm. 3.19.

Otherwise, by Def. 3.17, for each $q \in s(\mathsf{p})$ and each candidate q-branch π in CA, $\mathsf{p} \in \mathsf{ptp}(\pi)$.

From the proof of Thm. 3.15, s is bisimilar to one such q. Hence, each candidate s-branch matches a candidate q-branch in CA, thus it contains interactions where p participates. ◀

4 Design-by-Contract

We now extend the theory of choreography automata and communicating systems to handle specifications amenable to predicate over data exchanged through a protocol. The basic idea is to frame the design-by-contract theory proposed in [3] for global types in the context of c-automata. This theory advocates *global assertions* to specify and verify contracts among participants of a protocol. Taking inspiration from Design-by-Contract (DbC) [34], widely used in the practice of sequential programming [19, 14], a global assertion is a global type decorated with logical formulae predicating on the payload carried by interactions. Just as in the traditional DbC, the use of logical predicates allows one to specify protocols where the content of messages is somehow constrained.

4.1 Asserted choreography automata

To specify protocols that encompass constraints on payloads, we extend c-automata to *asserted c-automata*. The structure of messages is reshaped to account for sorted data in interactions and predicate over the payload of communications. More precisely, the set of *messages* $\mathcal{M}$ consists of *tagged tuples* $\tau\,\langle \mathtt{V}\rangle$ where τ is a *tag* and $\mathtt{V} = \mathsf{v}_1\ \mathsf{s}_1, \ldots, \mathsf{v}_h\ \mathsf{s}_h$ is a tuple of pairwise distinct sorted variables (namely, $\mathsf{v}_i = \mathsf{v}_j \implies i = j$ for $1 \leq i \leq j \leq h$). The set of variables of $\mathtt{V} = \mathsf{v}_1\ \mathsf{s}_1, \ldots, \mathsf{v}_h\ \mathsf{s}_h$ is $\mathsf{var}(\mathtt{V}) \triangleq \{\mathsf{v}_1, \ldots, \mathsf{v}_h\}$ and, accordingly $\mathsf{var}(\mathsf{m}) \triangleq \mathsf{var}(\mathtt{V})$ and $\mathsf{var}(\mathsf{p} \to \mathsf{q} : \mathsf{m}) \triangleq \mathsf{var}(\mathsf{m})$ are the set of variables of m and of $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ respectively. Intuitively, now an interaction specifies also the sort of the values communicated by the sender and the "local" variables where the receiver "stores" those values.

▶ **Example 4.1** (OLW variable sorts). When asking customer for another login attempt, wallet can send a message retry ⟨msg string⟩ where the payload msg yields an error message. ◇

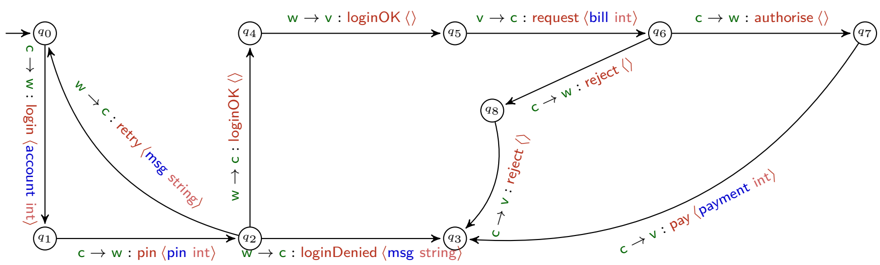
We borrow from [3] (with minor syntactic changes) the first-order logic to specify the constraints on payloads; the set $\mathcal{A}$ of logical formulae are derived from the following grammar

$$\mathsf{A}, \mathsf{B} ::= \top \mid \bot \mid \phi(e_1, \ldots, e_n) \mid \neg \mathsf{A} \mid \mathsf{A} \wedge \mathsf{B} \mid \mathsf{A} \supset \mathsf{B} \mid \exists \mathsf{v}\ \mathsf{s} :\ \mathsf{A} \tag{2}$$

In (2), ϕ ranges over pre-defined atomic predicates with fixed arities and sorts (e.g., bool, int, etc) [33, §2.8] and $e_1, \ldots, e_n$ denote expressions. Instead of fixing a specific language of expressions, we just assume that they encompass usual data types of programming languages and variables v. Also, we assume that sorts of expressions can be inferred (hence, we occasionally omit sorts and tacitly assume that usage of variables is consistent with respect to their sort). For simplicity, we consider only basic sorts (as in [3]). More complex static data structures can be handled similarly, while dynamic data structures (e.g., pointers) require to extend our theory with suitable semantics of value passing (e.g., deep-copy).

Let $\mathsf{var}(e)$ be the set of variables occurring in expression e; likewise $\mathsf{var}(\mathsf{A})$ denotes the set of free variables of predicate $\mathsf{A} \in \mathcal{A}$, while $\mathsf{bvar}(\mathsf{A})$ denotes the bound variables in A (defined in the standard way). Hereafter, assume that $\mathsf{var}(\mathsf{A}) \cap \mathsf{bvar}(\mathsf{A}) = \emptyset$.

▶ **Example 4.2** (OLW payloads). The payloads of the OLW protocol which we will use through the paper are those in the following FSA:



Notice that some messages have empty payloads. ◇

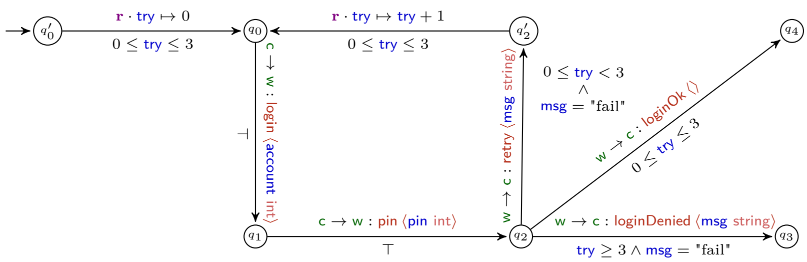
We will consider FSAs where transitions are decorated with *assertions*, namely formulae in $\mathcal{A}$ predicating on variables of the FSAs. The interplay between payloads and assertions requires some care to handle iterative behaviour and the scoping of variables. In fact, we will need to slightly change the FSA above to handle the iteration of the authentication phase.

Iterative computations require a few more ingredients. First we fix a *recursion context* ρ which maps each recursion variable $\mathbf{r}$ to a triplet $(\mathtt{V}, \mathsf{A}, q)$ consisting of

- a set of sorted variables $\mathtt{V}$ which identify the formal parameters of $\mathbf{r}$,
- a predicate $\mathsf{A} \in \mathcal{A}$, the loop invariant to be maintained through the iteration, and
- a state q of the FSA identifying the start of the iteration.

We assume that if $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, q)$ and $\rho(\mathbf{r}') = (\mathtt{V}', \mathsf{A}', q')$ then $\mathbf{r} \neq \mathbf{r}'$ implies $q \neq q'$ and $\mathtt{V} \cap \mathtt{V}' = \emptyset$. Then we use FSAs on the set $\widehat{\mathcal{L}_{\text{int}}}$ (ranged over by λ), defined as the union of $\mathcal{L}_{\text{int}}$ and the set of *recursive calls* which are defined as pairs $\mathbf{r} \cdot \iota$ of a recursive variable and a map assigning expressions to recursive parameters of $\mathbf{r}$.

▶ **Example 4.3** (OLW iteration). Using assertions, the constraint on the authentication phase of the OLW protocol described in § 1 can be specified as follows:



where $\rho(\mathbf{r}) = (\{\mathsf{try}\}, 0 \leq \mathsf{try} \leq 3, q_0)$. The automaton above refines the left part of the c-automaton in Ex. 4.2. In particular, states with the same names do correspond. States

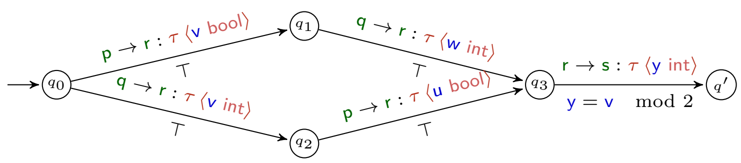
q'_0 and q'_2 are new (in particular q'_0 is the new initial state), introduced to correctly model iteration. The assertions on the transitions from states q'_0 and q'_2 model recursive calls where the try parameter is respectively set to 0 and incremented (cf. Ex. 4.5). ◇

Transitions $t = (q, (\lambda, \mathsf{A}), q')$, written as $q \xrightarrow[\mathsf{A}]{\lambda} q'$, are interpreted according to their label:

- If $\lambda = \mathsf{p} \to \mathsf{q} : \mathsf{m}$ then t (dubbed *interaction transition*) establishes a rely-guarantee relation: when t is fired, p *guarantees* A while q *assumes* that A holds.
- If $\lambda = \mathbf{r} \cdot \iota$ then t (dubbed *iteration transition*) records the invariant A (fixed by the recursion context ρ) that should be maintained through each loop corresponding to $\mathbf{r}$.

Variable scoping requires attention, as best illustrated by the following example.

▶ **Example 4.4** (Confusion). In the following FSA



it is not clear if the assertion on the transition from q_3 predicates on the variable v bound in the interaction between p and r or in the one between q and r, hence its sort is not clear. ◇

The binding and scoping of variables yield a first difference w.r.t. [3], where syntactic structures of global assertions facilitate the definition of these notions. The lack of syntactic structures of c-automata requires instead to introduce constructions to handle variables.

Let us now consider recursion. An FSA A *respects* a recursion context ρ when there are no loops without iteration transitions and for each iteration transition $t = q \xrightarrow[\mathsf{A}]{\mathbf{r}\cdot\iota} \hat{q}$ in A with $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, \hat{q})$

(a) t is the only outgoing transition of q and $q \neq \hat{q}$ and

(b) either q is the initial state of A or there is a unique transition entering q and it is an interaction transition.

Condition (a) forbids self-loops while (b) forces iterations to be guarded by interactions.

▶ **Example 4.5** (OLW is respectful). The requirements imposed by respectfulness are met by the FSA in Ex. 4.3. ◇

For an FSA $A = (Q, q_0, \widehat{\mathcal{L}_{\text{int}}} \times \mathcal{A}, \mathcal{T})$ on $\widehat{\mathcal{L}_{\text{int}}} \times \mathcal{A}$, we let $\mathsf{SPath}_A(q)$ denote the set of simple paths[1] reaching the state $q \in Q$ from q_0; also, $\mathsf{var}(q \xrightarrow[\mathsf{A}]{\alpha} q') \triangleq \mathsf{var}(\alpha)$ and $\mathsf{var}(q \xrightarrow[\mathsf{A}]{\mathbf{r}\cdot\iota} q') \triangleq \mathsf{var}(\mathbf{r}) \triangleq \mathtt{V}$ if $\rho(\mathbf{r}) = (\mathtt{V}, \mathsf{A}, \hat{q})$. Finally, we say that a transition $t \in \mathcal{T}$ from a state $q \in Q$ *fixes a variable* v *(in A)* if $\mathsf{v} \in \mathsf{var}(t)$ and, for each path $\pi \in \mathsf{SPath}_A(q)$ there is no transition $t' \in \pi$ that fixes v.

The next definition addresses the issues of confusion and respectfulness described above.

▶ **Definition 4.6** (Asserted c-automata). *An FSA, say* CA*, on the alphabet $\widehat{\mathcal{L}_{int}} \times \mathcal{A}$ such that*

1. *for each co-final span (π, π') in* CA*, if there are $t \in \pi$ and $t' \in \pi'$ such that both t and t' fix v then t and t' assign the same sort to v*
2. CA *respects the (fixed) recursion context ρ*
3. *the underlying c-automaton obtained by removing the assertions from* CA *is deterministic*

[1] A path is *simple* if no state occurs twice on it.

is an asserted c-automaton *(ac-automaton for short).*

Intuitively, one can think of a variable v fixed at a transition t as "local" to the receiver of the interaction labelling t; also, the sender of the interaction is aware of the value to be assigned to v. Condition (1) in Def. 4.6 simply avoids confusion on the sort of a variable when it could be assigned along different paths.

Without loss of generality, we can assume that $\mathsf{var}(t) \cap \mathsf{bvar}(\mathsf{A}) = \emptyset$ for all transitions and predicates A of an ac-automaton; in fact, such condition can be enforced by simply renaming bound variables in predicates. Hereafter, we write $q \xrightarrow{\lambda} q'$ instead of $q \xrightarrow[\top]{\lambda} q'$.

4.2 Consistent choreography automata

Our interpretation of transitions as rely-guarantee relations requires some care. Indeed, for a transition t to be viable, participants involved in t must "know" the variables used in t. In particular, if t is an interaction variable then the sender and receiver in t must "know" the assertion in t and participants involved in an iteration should "know" the invariant of the loop. Before formalising this in the next definition, we introduce the auxiliary concept of *assertion of a path of an ac-automaton*, which yields the conjunction of all assertions in π while substituting recursive variables with actual values of recursive calls. Formally, if $t = q_1 \xrightarrow[\mathsf{A}]{\mathbf{r}\cdot\iota} q_2$ then $\nabla(t) = \iota$, otherwise $\nabla(t)$ is the empty substitution.

Then the *assertion of a path* π is defined as $\mathbb{A}(\pi) = \mathbb{A}_{\mathsf{id}}(\pi)$ where

$$\mathbb{A}_\iota(\varepsilon) \triangleq \top \qquad \text{and} \qquad \mathbb{A}_\iota(q \xrightarrow[\mathsf{A}]{\lambda} q' \; \pi) \triangleq \mathsf{A}\iota' \wedge \mathbb{A}_{\iota'}(\pi) \quad \text{with} \quad \iota' = \iota[\nabla(q \xrightarrow[\mathsf{A}]{\lambda} q')]$$

Namely, the assertion of a path is the conjunct of all the assertions of its transitions once the recursion parameters are updated with their actual values. We can now define the notion of *knowledge* of a variable.

▶ **Definition 4.7** (Knowledge). *Let* CA *be an ac-automaton. A participant* $\mathsf{p} \in \mathcal{P}$ knows v at a transition $t = q \xrightarrow[\mathsf{A}]{\lambda} q'$ in CA *if*

- *either t fixes v and*
  (a) *if $\lambda \in \mathcal{L}_{int}$ then $\mathsf{p} \in \mathit{ptp}(\lambda)$ and*
  (b) *if $\lambda = \mathbf{r} \cdot \iota$ with $\rho(\mathbf{r}) = (\mathbb{V}, \mathsf{A}, q')$ and p is on a cycle from q' to q' then $\mathsf{v} \in \mathbb{V}$*
- *or $\mathsf{v} \in \mathit{var}(\mathsf{A})$ and there are a variable u and a transition t' on each path $\pi \in \mathit{SPath}_{\mathsf{CA}}(q)$ such that p knows u at t' and $\mathbb{A}(\pi) \supset \mathsf{v} = \mathsf{u}$ holds.*

Let $\mathsf{knw}_{\mathsf{CA}}(\mathsf{p}, t)$ *be the set of variables that* p *knows at* t *in* CA.

▶ **Example 4.8** (OLW knowledge). In the FSA of Ex. 4.2 both vendor and customer know bill at the outgoing transition of state q_5. Also, customer and wallet know the recursion variable try of the ac-automaton in Ex. 4.3. ◇

The notion of knowledge in Def. 4.7 is more complex than the one in [3]; this is an effect of the higher complexity in the notions of binding and scoping of variables. Def. 4.7 is instrumental to transfer the concept of *history-sensitivity* introduced in [3] to ac-automata.

▶ **Definition 4.9** (History sensitiveness). *An ac-automaton* CA *is* history-sensitive *if the following holds for each transition $t = q \xrightarrow[\mathsf{A}]{\lambda} q'$ in* CA

1. *$\lambda = \mathsf{p} \to \mathsf{q} : \mathsf{m}$ implies $\mathit{var}(\mathsf{A}) \subseteq \mathsf{knw}_{\mathsf{CA}}(\mathsf{p}, t)$, namely p knows each variable free in A at t.*
2. *$\lambda = \mathbf{r} \cdot \iota$ implies $\mathit{var}(\mathbf{r}) \subseteq \mathsf{knw}_{\mathsf{CA}}(\mathsf{p}, t)$ for each $\mathsf{p} \in \mathcal{P}$ occurring on a cycle from q' to q'.*

Condition (1) guarantees that the assertion of a transition cannot predicate on variables not "accessible" to the participants of the interaction. Condition (2) ensures that participants involved in a loop are aware of the loop invariant. The notion of history sensitivity in [3] relies on the fact that participant p knows a variable v on each interaction involving v. Here instead a weaker notion is adopted since, due to selective participation, the c-automaton may have a transition fixing v but not involving p.

▶ **Example 4.10** (OLW is history-sensitive)**.** The ac-automaton in Ex. 4.3 is history-sensitive. In particular, note that the variable try in the assertion on the transition from q_2 to q_3 is known to $\mathsf{customer}$ and wallet since it is in the invariant of the authentication loop. ◇

For a transition t of an ac-automaton CA to be enabled, it is not enough that the source state of t is reachable from the initial state of CA. In fact, the transition t can be fired if the information accumulated by the participants ensures the satisfiability of the assertion of t. To formalise this notion we introduce the following definitions. Given a state q of an ac-automaton CA, we let

$$\mathbb{P}_{\mathsf{CA}}(q) \triangleq \{\mathbb{A}(\pi) \mid \pi \text{ run to } q \text{ in } \mathsf{CA} \text{ and } \mathbb{A}(\pi) \text{ is satisfiable}\} \tag{3}$$

be the set of *preconditions* of q (in CA) and

$$\mathbb{E}_{\mathsf{CA}}(q) \triangleq \bigcup_{\mathsf{B} \in \mathbb{P}_{\mathsf{CA}}(q)} \left\{ \mathsf{B} \supset \bigvee_{q \xrightarrow[\mathsf{A}]{\lambda} q' \in \mathsf{CA}} \exists \mathsf{var}(\lambda) : \mathsf{A} \right\} \tag{4}$$

be the set of *enabling conditions* of q (in CA)

Similarly to [3] for global types, progress of ac-automata cannot be guaranteed if there is a possible computation leading to a state with no enabled transitions. Hence, we adapt from [3] the notion of *temporal satisfiability.*

▶ **Definition 4.11** (Temporal satisfiability)**.** *An ac-automaton* CA *is* temporally satisfiable *if for each* $q \in \mathsf{CA}$ *reachable from the initial state of* CA *each formula in* $\mathbb{E}_{\mathsf{CA}}(q)$ *is satisfiable.*

▶ **Example 4.12** (OLW is temporally satisfiable)**.** The ac-automaton in Ex. 4.3 is temporally satisfiable because the enabling conditions of all the nodes are satisfiable. However, if the assertion on the transition from q_2 to q_3 were replaced by e.g., $\mathsf{try} > 3 \wedge \mathsf{msg} =$ "fail" then temporal satisfiability would be violated because the precondition of the simple path from q_0 to q_2 would not entail $0 \leq \mathsf{try} < 3 \vee \mathsf{try} > 3$. ◇

As c-automata, ac-automata are well-formed if they are well-sequenced and well-branched; these two notions are as for c-automata modulo the presence of assertions, which are disregarded; we refer to Defs. 4.13–4.15 for the formal definitions. We state explicitly the definitions for well-formedness for ac-automata.

▶ **Definition 4.13** (Well-sequencedness for ac-automata)**.** *An ac-automaton* CA *is* well-sequenced *if for each two consecutive transitions* $q \xrightarrow[\mathsf{A}]{\alpha} q' \xrightarrow[\mathsf{B}]{\beta} q''$ *either*

(a) $\alpha \not\parallel \beta$ *or*

(b) *there is* q''' *such that* $q \xrightarrow[\mathsf{B}]{\beta} q''' \xrightarrow[\mathsf{A}]{\alpha} q''$*; furthermore for each transition* $q''' \xrightarrow[\mathsf{A'}]{\gamma} q''''$*,* $\gamma \parallel \alpha$ *and* $\gamma \parallel \beta$*.*

For well-branchedness we need to slightly adjust the notion of trace; traces of runs of an ac-automaton simply ignore assertions:

$$\mathsf{trace}(\emptyset) \triangleq \varepsilon \qquad \text{and} \qquad \mathsf{trace}(q \xrightarrow[\mathsf{A}]{\lambda} q' \; \pi) \triangleq \lambda \, \pi$$

Likewise, we let $\mathsf{ptp}((\alpha, \mathsf{A})) \triangleq \mathsf{ptp}(\alpha)$. We can now tune up full-awareness for ac-automata.

▶ **Definition 4.14** (Full awareness for ac-automata). *Let (π_1, π_2) be a pair of q-runs of an ac-automaton* CA*. Participant $\mathsf{p} \in \mathit{ptp}(\pi_1) \cap \mathit{ptp}(\pi_2)$ is* fully aware *of (π_1, π_2) if there are two labels $(\alpha_1, \mathsf{A}_1), (\alpha_2, \mathsf{A}_2) \in \mathcal{L}_{int} \times \mathcal{A}$ such that $\alpha_1 \neq \alpha_2$, $\mathsf{p} \in \mathit{ptp}(\alpha_1) \cap \mathit{ptp}(\alpha_2)$, and*

1. *either α_h is the first interaction in $L(\pi_h)$ for $h = 1, 2$*
2. *or there are proper prefixes $\hat{\pi}_1$ of π_1 and $\hat{\pi}_2$ of π_2 such that $\mathit{trace}(\hat{\pi}_1 \downarrow_{\mathsf{p}}) = \mathit{trace}(\hat{\pi}_2 \downarrow_{\mathsf{p}})$, the partner of p in α_h is fully aware of $(\hat{\pi}_1, \hat{\pi}_2)$, and $\mathit{trace}(\hat{\pi}_h)\alpha_h \in \mathit{pref}(\mathit{trace}(\pi_h))$ for $h \in \{1, 2\}$.*

Notice that, analougously to full-awareness for c-automata, Def. 4.14 considers only interaction labels. Finally we define well-branchedness.

▶ **Definition 4.15** (Well-branchedness for ac-automata). *An ac-automaton* CA *is* well-branched *if for all of its states q there is a partition $T_1, \ldots, T_k$ of $\mathit{out}(\mathsf{CA}, q)$ such that*

- *for all $1 \leq i \neq j \leq k$, $\mathit{ptp}(T_i) \cap \mathit{ptp}(T_j) = \emptyset$ and for each $q \xrightarrow[\mathsf{A}_i]{\alpha_i} q_i \in T_i$, $q \xrightarrow[\mathsf{A}_j]{\alpha_j} q_j \in T_j$ there exists q' such that $q_i \xrightarrow[\mathsf{A}_j]{\alpha_j} q'$ and $q_j \xrightarrow[\mathsf{A}_i]{\alpha_i} q'$*
- *for all $1 \leq i \leq k$, $\bigcap_{t \in T_i} \mathit{ptp}(t) \neq \emptyset$ and for all $\mathsf{p} \in \mathit{ptp}(\mathsf{CA}) \setminus \bigcap_{t \in T_i} \mathit{ptp}(t)$ and q-span (π_1, π_2) starting from transitions in T_i, if $\pi_1 \downarrow_{\mathsf{p}} \neq \pi_2 \downarrow_{\mathsf{p}}$ then either p is fully aware of (π_1, π_2) or there is $i \in \{1, 2\}$ such that $\mathsf{p} \notin \mathit{ptp}(\pi_i)$ and*
  - *the first transition in π_{3-i} involving p is with a participant which is fully aware of (π_1, π_2) and*
  - *for all runs π' such that $\pi_i \pi'$ is a candidate q-branch of* CA *the first transition in π' involving p is with a participant which is fully aware of (π_1, π_2).*

Finally, we can define *consistent* ac-automata.

▶ **Definition 4.16** (Consistency). *An ac-automaton is* consistent *if it is history-sensitive, temporally satisfiable, and well-formed.*

4.3 Asserted communicating systems

Projecting ac-automata requires to handle asserted transitions. We therefore extend communicating systems to *asserted communicating systems* (a-CSs for short), which basically are communicating systems where CFSMs are *asserted* (a-CFSMs for short), namely they have transitions decorated with formulae in $\mathcal{A}$. The synchronous semantics of a-CSs can be defined as an LTS similarly to the semantics of communicating systems. In fact, configurations can be defined as in Def. 2.9 taking into account assertions when synchronising transitions. This basically means that assertions are used to verify that a sent message guarantees the expectation of its receiver, that is the assertion the receiver relies upon.

Recall that a *prenex normal form* is a formula $\mathcal{Q}\mathsf{A}$ where $\mathcal{Q}$ is a sequence of quantifiers and variables (called *prefix*) and A is a quantifiers-free logical formula (called *matrix*) [33]. If $\mathsf{A}, \mathsf{B} \in \mathcal{A}$ then $\mathsf{A} \circ \mathsf{B}$ is a logical formula obtained by quantifying with the prefix of a prenex normal form A' logically equivalent to A the conjunction of B with the matrix of A'. Similarly to assertions for paths on ac-automata, we define assertions of a run of an a-CFSM

$$\mathbb{A}(\varepsilon) \triangleq \top \qquad \text{and} \qquad \mathbb{A}(q \xrightarrow[\mathsf{A}]{\ell} q' \; \pi) \triangleq \mathsf{A} \circ \mathbb{A}(\pi)$$

The preconditions of a state of an a-CFSM are defined as for ac-automata but for the use of the assertion function $\mathbb{A}$ for CFSMs instead of the corresponding one for ac-automata.

▶ **Definition 4.17** (Semantics of a-CS). *The* semantics *of an a-CS* $\mathsf{S} = (M_\mathsf{p})_{\mathsf{p}\in\mathcal{P}}$ *is the transition system* $[\![\mathsf{S}]\!]$ *defined by taking the set of configurations as in Def. 2.9 and as set of transitions the smallest set including*

- $s_1 \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} s_2$ *if* $\mathsf{p},\mathsf{q} \in \mathcal{P}$ *and*
  - $s_1(\mathsf{p}) \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} s_2(\mathsf{p})$ *in* M_p, $s_1(\mathsf{q}) \xrightarrow[\mathsf{B}]{\mathsf{pq?m}} s_2(\mathsf{q})$ *in* M_q *and, there are* $\mathsf{A}' \in \mathbb{P}_{M_\mathsf{p}}(s_1(\mathsf{p}))$ *and* $\mathsf{B}' \in \mathbb{P}_{M_\mathsf{q}}(s_1(\mathsf{q}))$ *such that it holds* $(\mathsf{A}' \supset \mathsf{A}) \wedge (\mathsf{B}' \supset \mathsf{B}) \wedge (\mathsf{A}' \circ \mathsf{B}' \circ \mathsf{A}) \supset \exists \mathit{var}(\mathsf{m}) : \mathsf{B}$
  - *and* $s_1(\mathsf{x}) = s_2(\mathsf{x})$ *for all* $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p},\mathsf{q}\}$
- $s_1 \xrightarrow[\mathsf{A}]{\varepsilon} s_2$ *if* $\mathsf{p} \in \mathcal{P}$ *and*
  - $s_1(\mathsf{p}) \xrightarrow[\mathsf{A}]{\varepsilon} s_2(\mathsf{p})$ *in* M_p *and there is* $\mathsf{A}' \in \mathbb{P}_{M_\mathsf{p}}(s_1(\mathsf{p}))$ *such that* $\mathsf{A}' \supset \mathsf{A}$
  - *and* $s_1(\mathsf{x}) = s_2(\mathsf{x})$ *for all* $\mathsf{x} \in \mathcal{P} \setminus \{\mathsf{p}\}$.

Like the projection of communicating systems (cf. Def. 2.7), the projection of a-CSs relies on the determinisation and minimisation of a-CFSMs. The presence of assertions imposes to adapt the classical constructions on FSA to a-CFSMs. More precisely, we have to generalise equality on labels of the form (λ, A). Essentially, this is done by (injectively) renaming the variables occurring in actions and assertions decorating transitions. For σ an endofunction on variables and $\mathsf{m} = \tau \langle \mathsf{v}_1\ \mathsf{s}_1, \ldots, \mathsf{v}_h\ \mathsf{s}_h \rangle$ let $\mathsf{m}\sigma \triangleq \tau \langle \sigma(\mathsf{v}_1)\ \mathsf{s}_1, \ldots, \sigma(\mathsf{v}_h)\ \mathsf{s}_h \rangle$; we define

$$\varepsilon\sigma \triangleq \varepsilon \qquad \text{and} \qquad (\mathsf{p} \to \mathsf{q} : \mathsf{m})\sigma \triangleq \mathsf{p} \to \mathsf{q} : \mathsf{m}' \qquad \text{where} \qquad \mathsf{m}' = \mathsf{m}\sigma$$

Two labels (λ, A) and (λ', A') are *equivalent*, in symbols $(\lambda, \mathsf{A}) \sim (\lambda', \mathsf{A}')$, if there is an injective substitution of variables such that $\lambda = \lambda'\sigma$ and A is logically equivalent to $\mathsf{A}'\sigma$. We will similarly consider equivalence on $\mathcal{L}_{\text{act}} \times \mathcal{A}$.

The ε-closure of an a-CFSM $M = (Q, q_0, \mathcal{L}_{\text{act}}, \mathcal{T})$ is the map $\varepsilon\text{-clos}_M : Q \to 2^{Q\times\mathcal{A}}$ defined assigning to each state q of M the set of states reachable with ε-transitions together with their assertions; more precisely, for each $q \in Q$, $\varepsilon\text{-clos}_M(q)$ is the smallest set satisfying

$$\varepsilon\text{-clos}_M(q) \triangleq \{(q, \top)\} \cup \bigcup_{(q',\mathsf{A})\in\varepsilon\text{-clos}_M(q)} \left\{ (q'', \mathsf{A} \circ \mathsf{A}') \mid q' \xrightarrow[\mathsf{A}']{\varepsilon} q'' \in \mathcal{T} \right\} \tag{5}$$

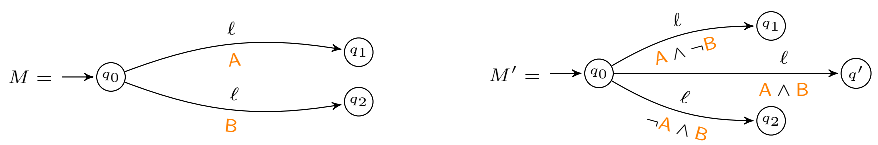
Removal of ε-transitions from an a-CFSM M is computed, using (5), similarly to the classical algorithm on FSAs $\langle \mathbb{Q}, \varepsilon\text{-clos}(q_0), \mathcal{L}_{\text{act}}, \mathbb{T} \rangle$ where

$$\mathbb{Q} = \{\varepsilon\text{-clos}_M(q) \mid q \in Q\} \qquad \text{and}$$

$$\mathbb{T} = \{Q \xrightarrow[\mathsf{A}_1\circ\mathsf{A}\circ\mathsf{A}_2]{\ell} Q' \mid q_1 \xrightarrow[\mathsf{A}]{\ell} q_2 \in \mathcal{T} \text{ for some } (q_1, \mathsf{A}_1) \in Q \text{ and } (q_2, \mathsf{A}_2) \in Q'\}$$

Handling assertions in the determinisation algorithm requires some care. We illustrate the problem in the following example.

▶ **Example 4.18** (Non-determinism & assertions). Consider the two a-CFSMs below



If both A and B are satisfiable then M has a non-deterministic behaviour. We therefore aim to define a determinisation algorithm which on M yields something like M'. Also, the new state q' should provide transitions corresponding to both transitions from q_1 and q_2. ◇

Let $M = (Q, q_0, \mathcal{L}_{\mathrm{act}}, \mathcal{T})$ be a CFSM. A state $q \in Q$ *is non-deterministic on* $\ell \in \mathcal{L}_{act}$ if its *derivative in* M *with respect to* ℓ, defined as $\partial_M(q, \ell) \triangleq \{(\mathsf{A}, q') \mid q \xrightarrow[\mathsf{A}]{\ell} q' \text{ in } M\}$, has more than one element. Also, if $X, Y \subseteq \partial_M(q, \ell)$ then $\Delta(X, Y) \triangleq \bigwedge_{(\mathsf{A},q)\in X} \mathsf{A} \wedge \bigwedge_{(\mathsf{B},q)\in Y} \neg\mathsf{B}$. The determinisation of M is obtained by applying the classical FSA determinisation algorithm to the ε-closure of the a-CFSM $M' = (Q', q_0, \mathcal{L}_{\mathrm{act}}, \mathcal{T}' \cup \mathcal{T}'' \cup \mathcal{T}''')$ where

$$Q' \triangleq Q \cup \bigcup_{q\in Q, \ell\in\mathcal{L}_{\mathrm{act}}} \{ \langle X \rangle \mid q \text{ is non-deterministic on } \ell \text{ and } \emptyset \neq X \subseteq \partial_M(q, \ell) \}$$

$$\mathcal{T}' \triangleq \left\{ q \xrightarrow[\mathsf{A}]{\ell} q' \in \mathcal{T} \mid \partial_M(q, \ell) \text{ is a singleton} \right\}$$

$$\mathcal{T}'' \triangleq \bigcup_{\emptyset\neq X\subseteq\partial_M(q,\ell)} \left\{ q \xrightarrow[\Delta(X,Y)]{\ell} \langle X \rangle \mid \partial_M(q, \ell) \text{ not a singleton} \quad \text{and} \quad Y = \partial_M(q, \ell) \setminus X \right\}$$

$$\mathcal{T}''' \triangleq \bigcup_{\emptyset\neq X\subseteq\partial_M(q,\ell)} \left\{ \langle X \rangle \xrightarrow[\mathsf{A}]{\ell} q' \mid \text{there is } q \xrightarrow[\mathsf{A}]{\ell} q' \in \mathcal{T} \quad \text{with} \quad \{q\} \times \mathcal{A} \cap X \neq \emptyset \right\}$$

Basically, we (i) introduce a new state $\langle X \rangle$ for any combination of assertions of ℓ-transitions, (ii) replace non-deterministic behaviours on ℓ with a set of ℓ-transitions with "disjoint" assertions, and (iii) let state $\langle X \rangle$ have the transitions that any of the states $q \in X$ has in M.

We remark that the adaptation of the determinisation algorithm is imposed by the use of a-CFSMs to model local behaviour. This is a main technical difference with respect to [3] where local types with assertions, which need no determinisation, play the role of a-CFSMs.

The projection of an ac-automaton acts as the projection of c-automata on interactions and accommodates the variables not known to the participant by existentially quantifying them. This requires to consider the points in the ac-automaton where variables are fixed.

▶ **Definition 4.19** (Projection of ac-automata). *The* projection *on* $\mathsf{p} \in \mathcal{P}$ *of an asserted transition* t *in an ac-automaton* CA *on* $\mathcal{P}$*, written* $t{\downarrow}_{\mathsf{CA},\mathsf{p}}$*, is defined by:*

$$t{\downarrow}_{\mathsf{CA},\mathsf{p}} = \begin{cases} q \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} q' & \text{if } t = q \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q' \\ q \xrightarrow[\mathsf{A}]{\mathsf{qp?m}} q' & \text{if } t = q \xrightarrow[\mathsf{A}]{\mathsf{q}\to\mathsf{p}:\mathsf{m}} q' \\ q \xrightarrow[\exists X\,:\ \mathsf{A}(\nabla(t))]{\varepsilon} q' & \text{if } t = q \xrightarrow[\mathsf{A}]{\lambda} q',\ \mathsf{p} \notin \mathit{ptp}(\lambda),\ \text{and } X = \{\mathsf{v} \in \mathit{var}(\mathsf{A}) \mid t \text{ fixes } \mathsf{v} \text{ in } \mathsf{CA}\} \end{cases}$$

The projection *of* CA *on* $\mathsf{p} \in \mathcal{P}$*, denoted* $\mathsf{CA}{\downarrow}_{\mathsf{p}}$*, is obtained by determinising and minimising up-to-language equivalence the* intermediate *a-CFSM*

$$A_{\mathsf{p}} = \left\langle \mathbb{S}, q_0, \mathcal{L}_{act}, \left\{ (q \xrightarrow[\mathsf{A}]{\lambda} q'){\downarrow}_{\mathsf{CA},\mathsf{p}} \mid q \xrightarrow[\mathsf{A}]{\lambda} q' \text{ in } \mathsf{CA} \right\} \right\rangle$$

where (i) syntactic equality of labels is replaced by $\sim$ *and (ii)* ε*-transitions are those with label of the form* $(\varepsilon, \mathsf{A})$*. The* projection of CA*, written* $\mathsf{CA}{\downarrow}$*, is the a-CS* $(\mathsf{CA}{\downarrow}_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$.

Well-formed consistent ac-automata are deadlock-free; the proof mimics the one for c-automata. Let $q \xrightarrow[\mathsf{A}_1,\ldots,\mathsf{A}_n]{\varepsilon} q'$ abbreviate $q \xrightarrow[\mathsf{A}_1]{\varepsilon} \ldots \xrightarrow[\mathsf{A}_n]{\varepsilon} q'$.

▶ **Lemma 4.20.** *If* A_{p} *and* A_{q} *are the intermediate a-CFSM for two participants* p *and* q *of a well-formed ac-automaton* CA *and*

$$q \xrightarrow[\mathsf{A}_1,\ldots,\mathsf{A}_n]{\varepsilon} q_{\mathsf{p}} \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} q'_{\mathsf{p}} \text{ in } A_{\mathsf{p}} \qquad \text{and} \qquad q \xrightarrow[\mathsf{A}'_1,\ldots,\mathsf{A}'_n]{\varepsilon} q_{\mathsf{q}} \xrightarrow[\mathsf{A}]{\mathsf{pq?m}} q'_{\mathsf{q}} \text{ in } A_{\mathsf{q}}$$

then there is a state q' such that $q \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q' \in out(\mathsf{CA}, q)$.

Proof. The proof of Lemma 3.12 can be repeated by observing that assertions do not play any role for concurrent transitions. Details follow.

Let the two runs for p and q be the projections of runs π_{p} and π_{q} in CA. We have two cases, depending on whether $\pi_{\mathsf{p}} = \pi_{\mathsf{q}}$.

If $\pi_{\mathsf{p}} = \pi_{\mathsf{q}}$ then the last transition of π_{p} is concurrent to all the previous ones. Indeed, the previous transitions are projected to ε both on p and on q, hence neither p nor q can occur in the label. Thus, by well-sequencedness, the state q has a transition with label $(\mathsf{p} \to \mathsf{q} : \mathsf{m}, \mathsf{A})$.

Assume $\pi_{\mathsf{p}} \neq \pi_{\mathsf{q}}$. By construction,

$$\pi_{\mathsf{p}} = \pi\ t_{\mathsf{p}}\ \pi'_{\mathsf{p}}\ q_{\mathsf{p}} \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'_{\mathsf{p}} \qquad \text{and} \qquad \pi_{\mathsf{q}} = \pi\ t_{\mathsf{q}}\ \pi'_{\mathsf{q}}\ q_{\mathsf{q}} \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'_{\mathsf{q}}$$

for a q-run π, two transitions t_{p} and t_{q}, two runs π'_{p} and π'_{q}; observe that the ending state of π, say $\hat{q}$, is also the source state of t_{p} and t_{q} while π'_{p} (resp. π'_{q}) ends in q_{p} (resp. q_{q}).

Note that $\mathsf{p} \notin \mathsf{ptp}(\pi\ t_{\mathsf{p}}\ \pi'_{\mathsf{p}})$ and $\mathsf{q} \notin \mathsf{ptp}(\pi\ t_{\mathsf{q}}\ \pi'_{\mathsf{q}})$. If $\mathsf{q} \notin \mathsf{ptp}(\pi\ t_{\mathsf{p}}\ \pi'_{\mathsf{p}})$ or $\mathsf{p} \notin \mathsf{ptp}(\pi\ t_{\mathsf{q}}\ \pi'_{\mathsf{q}})$ then the thesis immediately follows by well-sequencedness as before. Therefore, without loss of generality, assume $\mathsf{p} \in \mathsf{ptp}(\pi\ t_{\mathsf{q}}\ \pi'_{\mathsf{q}})$ (the case where q occurs in $\pi\ t_{\mathsf{p}}\ \pi'_{\mathsf{p}}$ is similar and hence omitted).

Consider now the two $\hat{q}$-runs $\pi_1 = t_{\mathsf{p}}\ \pi'_{\mathsf{p}}$ and $\pi_2 = t_{\mathsf{q}}\ \pi'_{\mathsf{q}}$. By well-branchedness, there is a partition of $out(\mathsf{CA}, \hat{q})$ satisfying the conditions of Def. 4.15 (using $\sim$ for label equality). Then t_{p} and t_{q} cannot belong to the same equivalence class of such partition since neither p nor q are fully aware of (π_1, π_2) and the first interaction of p on π_1 is with q. Hence, $t_{\mathsf{p}} = \hat{q} \xrightarrow{\alpha_1} q_1$ and $t_{\mathsf{q}} = \hat{q} \xrightarrow{\alpha_2} q_2$ necessarily belong to different equivalence classes. Therefore, again by well-branchedness, there is a state $\hat{q}'$ such that $q_1 \xrightarrow{\alpha_2} \hat{q}'$ and $q_2 \xrightarrow{\alpha_1} \hat{q}'$. Hence, the transitions of π'_{p} and those of π'_{q} form commuting diamonds and therefore there is a q-run in CA where q does not occur and all the transitions involving p in π'_{q} follow a transition with label $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ (easily by induction on the length of π'_{p} and π'_{q}). The thesis then follows since, as before, the transition labelled by $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ commutes with any preceding transition by well-sequencedness. ◀

▶ **Lemma 4.21.** *Let CA be a well-formed ac-automaton and (π_1, π_2) a q-span in CA with first transitions t_1 and t_2, respectively. Let $t_1 = q \xrightarrow{\alpha_1} q'_1$. If t_1 and t_2 are concurrent then either α_1 occurs on $\mathsf{trace}(\pi_2)$ or for each run $\pi_2\ \pi$ in CA, α_1 occurs in $\mathsf{trace}(\pi)$.*

Proof. We can reshape the proof of Lemma 3.13 observing that assertions are immaterial to the reasoning. ◀

▶ **Lemma 4.22.** *Let CA be a well-formed ac-automaton, $\mathsf{p}, \mathsf{q} \in \mathcal{P}_{\mathsf{CA}}$, and $\alpha \in \mathcal{L}_{int}$. If $w_{\mathsf{p}}\alpha$, $w_{\mathsf{q}}\alpha$, $w \in L(\mathsf{CA})$ are three words such that $w_{\mathsf{p}} \downarrow_{\mathsf{p}} = w \downarrow_{\mathsf{p}}$ and $w_{\mathsf{q}} \downarrow_{\mathsf{q}} = w \downarrow_{\mathsf{q}}$ then $w\alpha \in L(\mathsf{CA})$.*

Proof. We can reshape the proof of Prop. 3.14 observing that assertions are immaterial to the reasoning. ◀

We show that projections of consistent ac-automata yield deadlock-free asserted communicating systems. The next result corresponds to Thm. 3.15 for ac-automata. The main differences are (i) that consistency of ac-automata is required (as opposed to well-formedness for c-automata) and (ii) that an ac-automaton is weakly bisimilar to the corresponding projected system due to the fact that iterative transitions of the ac-automaton are projected on ε-transitions.

▶ **Proposition 4.23.** *Any consistent ac-automaton* CA *is weakly bisimilar to* $[\![\mathsf{CA}\downarrow]\!]$.

Proof. Let $\mathsf{CA} = \langle Q, q_0, \mathcal{L}_{\text{int}}, \mathcal{T} \rangle$ and let $\mathbb{S}$ be the set of configurations of $[\![\mathsf{CA}\downarrow]\!]$. Recall that, due to determinisation and minimisation, for each $s \in \mathbb{S}$ and each participant p, $s(\mathsf{p})$ is a subset of $Q \cup (Q \times \mathcal{A})$. Also, $q \in s(\mathsf{p})$ holds if q belongs to $s(\mathsf{p})$ or if there is an assertion A such that $(q, \mathsf{A}) \in s(\mathsf{p})$. We show by coinduction that the relation

$$\mathcal{R} = \{(q, s) \in Q \times \mathbb{S} \mid q \in s(\mathsf{p}) \text{ for each } \mathsf{p} \in \mathcal{P}_{\mathsf{CA}}\}$$

is a weak bisimulation, where in CA iterative transitions are treated as ε-transitions. Since weak bisimulation implies trace equivalence, we also have that corresponding elements are reachable via the same trace.

Let $(q, s) \in \mathcal{R}$; fixed $\mathsf{B} \in \mathbb{P}_{\mathsf{CA}}(q)$, by definition of temporal satisfiability we have that

$$\mathsf{B} \supset \bigvee_{q \xrightarrow[\mathsf{A}]{\lambda} q' \in \mathsf{CA}} \exists \mathsf{var}(\lambda) : \mathsf{A} \qquad \text{is satisfiable}$$

and we take an enabled transition t from q in CA (i.e., the assertion of t is entailed by B). We have two cases depending on whether the challenge t is an interaction or an iterative transition.

- If $t = q \xrightarrow[\mathsf{A}]{\mathsf{p} \to \mathsf{q}:\mathsf{m}} q'$ then, by definition of projection (cf. Def. 4.19) and ε-closure, $s(\mathsf{p}) \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} s'(\mathsf{p})$ is in $\mathsf{CA}\downarrow_{\mathsf{p}}$ and $s(\mathsf{q}) \xrightarrow[\mathsf{A}]{\mathsf{pq?m}} s'(\mathsf{q})$ is in $\mathsf{CA}\downarrow_{\mathsf{q}}$ (since $q \in s(\mathsf{p}) \cap s(\mathsf{q})$) while $s(\mathsf{x}) = s'(\mathsf{x})$ for each $\mathsf{x} \notin \{\mathsf{p}, \mathsf{q}\}$. Since $\mathsf{B} \in \mathbb{P}_{\mathsf{CA}}(q)$, by definition, there is a run π to q in CA such that $\mathbb{A}(\pi)$ is satisfiable; hence $\pi\downarrow_{\mathsf{p}}$ is a run to q in (the intermediate of) $\mathsf{CA}\downarrow_{\mathsf{p}}$ with $\mathbb{A}(\pi\downarrow_{\mathsf{p}})$ satisfiable and entailing A, and likewise for q. The thesis then follows from the definition of semantics of a-CS since $s \xrightarrow[\mathsf{A}]{\mathsf{p} \to \mathsf{q}:\mathsf{m}} s'$ because A entails $\exists \mathsf{var}(\mathsf{m}) : \mathsf{A}$.
- If $t = q \xrightarrow[\mathsf{A}]{\mathbf{r} \cdot \iota} q'$, then, by definition of projection (cf. Def. 4.19), for all $\mathsf{p} \in \mathcal{P}_{\mathsf{CA}}$ the projected a-CFSM $\mathsf{CA}\downarrow_{\mathsf{p}}$ contains the transition $q \xrightarrow[\exists \mathsf{X} :\ \mathsf{A}(\nabla(t))]{\varepsilon} q'$ with $\mathsf{X} = \{\mathsf{v} \in \mathsf{var}(\mathsf{A}) \mid t \text{ fixes } \mathsf{v} \text{ in } \mathsf{CA}\}$. Let s be the configuration of $[\![\mathsf{CA}\downarrow]\!]$ such that $q \in s(\mathsf{p})$ for all $\mathsf{p} \in \mathcal{P}_{\mathsf{CA}}$. By Def. 4.17, $[\![\mathsf{CA}\downarrow]\!]$ has a run

  $$\pi = s \xrightarrow[\exists \mathsf{X} :\ \mathsf{A}(\nabla(t))]{\varepsilon} \cdots \xrightarrow[\exists \mathsf{X} :\ \mathsf{A}(\nabla(t))]{\varepsilon} s'$$

  such that $q' \in s'(\mathsf{p})$ for all $\mathsf{p} \in \mathcal{P}_{\mathsf{CA}}$ (run π is obtained by firing transition t in a-CFSM $\mathsf{CA}\downarrow_{\mathsf{p}}$). Hence, $(q', s') \in \mathcal{R}$ as required.

Now, let $(q, s) \in \mathcal{R}$ and consider a challenge from $[\![\mathsf{CA}\downarrow]\!]$, namely $s \xrightarrow[\mathsf{A}]{\mathsf{p} \to \mathsf{q}:\mathsf{m}} s'$ (note that $\mathsf{CA}\downarrow$ does not have ε-transitions since the a-CFSMs projected from CA are determinised). By Def. 4.17, $s(\mathsf{x}) = s'(\mathsf{x})$ for each $\mathsf{x} \notin \{\mathsf{p}, \mathsf{q}\}$, $s(\mathsf{p}) \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} s'(\mathsf{p})$, and $s(\mathsf{q}) \xrightarrow[\mathsf{A}]{\mathsf{pq?m}} s'(\mathsf{q})$ are respectively in $M_{\mathsf{p}} = \mathsf{CA}\downarrow_{\mathsf{p}}$ and $M_{\mathsf{q}} = \mathsf{CA}\downarrow_{\mathsf{q}}$. Hence, by definition of determinisation, there are runs such that

$$\tilde{q}_{\mathsf{p}} \xrightarrow[\mathsf{A}_1, \dots, \mathsf{A}_n]{\varepsilon} q_{\mathsf{p}} \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} q'_{\mathsf{p}} \text{ in } A_{\mathsf{p}} \qquad \text{and} \qquad \tilde{q}_{\mathsf{q}} \xrightarrow[\mathsf{A}'_1, \dots, \mathsf{A}'_n]{\varepsilon} q_{\mathsf{q}} \xrightarrow[\mathsf{A}]{\mathsf{pq?m}} q'_{\mathsf{q}} \text{ in } A_{\mathsf{q}}$$

where A_{p} and A_{q} are the intermediate automata of M_{p} and M_{q} respectively.

We have two cases:

- If $q = \tilde{q}_{\mathsf{p}} = \tilde{q}_{\mathsf{q}}$ then $q \xrightarrow[\mathsf{A}]{\mathsf{p} \to \mathsf{q}:\mathsf{m}} q'$ by Lemma 4.20.

- Otherwise, since q and s are in the bisimulation, they are also reached by the same trace. Now consider a word w that reaches q and matching words w_{p} and w_{q} obtained by lifting to CA runs reaching $\tilde{q}_{\mathsf{p}}$ and $\tilde{q}_{\mathsf{q}}$ in the respective auxiliary automata. By construction, they are in the hypothesis of Lemma 4.22, hence $q \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'$.

Therefore, in both cases we have $t = q \xrightarrow[\mathsf{A}]{\mathsf{p}\to\mathsf{q}:\mathsf{m}} q'$ is in CA.

We have then to show that $q' \in s'(\mathsf{x})$ for each x. By construction (Def. 4.19), for each x, the intermediate a-CFSM A_{x} contains the transition $t_{\mathsf{x}} = q_{\mathsf{x}} \xrightarrow[\mathsf{A}_{\mathsf{x}}]{(\mathsf{p}\to\mathsf{q}:\mathsf{m})\downarrow_{\mathsf{x}}} q'_{\mathsf{x}}$ for some $q_{\mathsf{x}} \in s'(\mathsf{x})$ and some q'_{x} where

- if $\mathsf{x} \notin \{\mathsf{p},\mathsf{q}\}$ then $(\mathsf{p} \to \mathsf{q} : \mathsf{m})\downarrow_{\mathsf{x}} = \varepsilon$ and A_{x} is $\exists \mathsf{X} : \mathsf{A}$ with $\mathsf{X} = \{\mathsf{v} \in \mathsf{var}(\mathsf{A}) \mid t_{\mathsf{x}} \text{ fixes } \mathsf{v} \text{ in CA}\}$. Hence $q_{\mathsf{x}} \in s(\mathsf{x})$ implies $q'_{\mathsf{x}} \in s'(\mathsf{x})$ as required since $s(\mathsf{x})$ contains the ε-closure of its elements by construction.
- If $\mathsf{x} = \mathsf{p}$ then $(\mathsf{p} \to \mathsf{q} : \mathsf{m})\downarrow_{\mathsf{x}} = \mathsf{pq!m}$ and $\mathsf{A}_{\mathsf{x}} = \mathsf{A}$. Thus $q_{\mathsf{x}} \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} q'_{\mathsf{x}}$ and, as shown above, $s(\mathsf{p}) \xrightarrow[\mathsf{A}]{\mathsf{pq!m}} s'(\mathsf{p})$. Since $q_{\mathsf{x}} \in s(\mathsf{p})$ and CA is deterministic then $q'_{\mathsf{x}} \in s'(\mathsf{x})$ since $s'(\mathsf{x})$ is the ε-closure of q'_{x} by construction.
- If $\mathsf{x} = \mathsf{q}$ then the reasoning is analogous to the previous case.

Finally, by Def. 4.17, there are $\mathsf{A}' \in \mathbb{P}_{M_{\mathsf{p}}}(s(\mathsf{p}))$ and $\mathsf{A}'' \in \mathbb{P}_{M_{\mathsf{q}}}(s(\mathsf{q}))$ such that both A' and A'' entail A. Hence the thesis follows since A entails $\exists \mathsf{var}(\mathsf{m}) : \mathsf{A}$. ◀

As for c-automata, Prop. 4.23 ensures that the language of a consistent ac-automaton coincides with the language of its projection.

▶ **Corollary 4.24.** $L(\mathsf{CA}) = L([\![\mathsf{CA}\downarrow]\!])$ *for any consistent ac-automaton* CA.

Proof. From Prop. 4.23 given that bisimulation implies trace equivalence. ◀

Final states and deadlock freedom of an ac-automaton are defined as for c-automata (cf. Def. 3.17 and Def. 3.18 respectively) modulo the different labels of transitions.

▶ **Theorem 4.25** (Projections of consistent ac-automata are deadlock-free). *If* CA *is a consistent ac-automaton then* CA↓ *is deadlock-free.*

Proof. By contradiction, assume that there is a reachable configuration s in $[\![\mathsf{CA}\downarrow]\!]$ with no outgoing transition and a participant p for which $s(\mathsf{p})$ is not final. Then, by definition of final state, for each $q \in s(\mathsf{p})$ and each candidate q-branch, $\mathsf{p} \in \mathsf{ptp}(\pi)$. From the proof of Prop. 4.23, there is a configuration s bisimilar to q; hence, s should answer the challenge from the first action of π, hence it has an outgoing transition against the hypothesis. ◀

Observe that Thm. 4.25 requires ac-automata to be consistent; in particular, it requires history sensitiveness (cf. Def. 4.9) and temporal satisfiability (cf. Def. 4.11). The two requirements ensure that assertions on the transitions do not spoil deadlock freedom.

5 TypeScript Programming via Flexible C-Automata

We showcase the main theoretical results and constructions in this paper with a tool, CAScr, the first implementation of Scribble [20, 37, 47] that relies on c-automata, for deadlock-free distributed programming. CAScr takes the popular *top-down approach* to system development based on choreographic models, following the original methodology of Scribble and multiparty session types [21]. The top-down approach enables *correctness-by-construction*: a developer provides a global description for the whole communication protocol; by projecting the global

$$[\textsc{Choice}]\ \boldsymbol{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i \xrightarrow{\mathsf{p}\to\mathsf{q_j}:\mathsf{m_j}} G_j \quad (j \in I)$$

$$[\textsc{Rec}]\ \frac{G[\mu\mathbf{r}.G/\mathbf{r}] \xrightarrow{\alpha} G'}{\mu\mathbf{r}.G \xrightarrow{\alpha} G'} \qquad [\textsc{Pass}]\ \frac{G_j \xrightarrow{\alpha} G'_j \quad \mathsf{p}, \mathsf{q}_j \notin \mathsf{ptp}(\alpha) \quad \forall j \in I}{\boldsymbol{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i \xrightarrow{\alpha} \boldsymbol{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G'_i}$$

Figure 3 LTS semantics over global types

protocol, APIs are generated from local CFSMs, which ensure the safe implementation of each participant. The core theory of c-automata from § 3 guarantees deadlock freedom for the distributed implementation of flexible global protocols. As a first application we target web development, supporting in particular the TypeScript programming language.

In this section we present our development in three steps:

1. *translation of global protocols into choreography automata*: for the specification of global protocols, CAScr relies on the Scribble language, and global Scribble protocols are formally global multiparty session types protocols [37]; we define a function that maps these into choreography automata, and discuss the relation between the two formalisms;
2. *protocol specification and projections*: from the specification of the global protocol, CAScr generates, through its translation into c-automata and the subsequent projection, a collection of CFSMs, which are the abstract representation of the communication behaviour of each participant (cf. part **(a)**, Fig. 4a on page 28);
3. *API generation for deadlock-free distributed web development*: we discuss our choice of targeting TypeScript and web development, and illustrate how CAScr provides support for this (cf. part **(b)**, Fig. 4a on page 28); finally we comment on possible extensions.

5.1 From Multyparty Session Protocols to C-Automata

C-automata and asserted c-automata can be directly produced by the system designer and fed to our approach to ensure their correct behaviour. However, to improve the usability of the approach, our implementation, detailed in the next section, integrates c-automata with the Scribble framework. This framework is based on the theory of global types, hence we study below the relations between global types and c-automata. The syntax of global types is given by the following grammar:

$$G \quad ::= \quad \mathsf{end} \quad | \quad \mu\mathbf{r}.G \quad | \quad \mathbf{r} \quad | \quad \boldsymbol{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i$$

We simply write $\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i$ instead of $\boldsymbol{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i$ when $I = \{i\}$. In a recursive type $\mu\mathbf{r}.G$ all occurrences of the recursion variable $\mathbf{r}$ in G are bound (this is the only binder for global types); we moreover assume that the occurrences of $\mathbf{r}$ in G are guarded. Hereafter we assume the so-called Barendregt convention, that is names of bound variables are all distinct and different from names of free variables.

The operational semantics of global types is the LTS induced by the rules in Fig. 3 where labels are drawn from the alphabet $\mathcal{L}_{\text{int}}$. Since the semantics of global types is an LTS, it can be represented as a c-automaton only if it is finite state. Unfortunately, the interplay between rule [Pass] and recursion allows one to generate infinite state LTSs, as shown below.

▶ **Example 5.1** (Infinite-state LTS). Let $G_{\text{inf}} = \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r} + \beta.\mathsf{end})$ where $\delta \parallel \alpha$, $\delta \parallel \gamma$, $\alpha \not\parallel \gamma$, $\beta \not\parallel \delta$, and $\beta \not\parallel \alpha$. Note that the traces $(\alpha\ \gamma)^n$ are included in the semantics for all $n > 1$. Executing $(\alpha\ \gamma)^n$ results in the following computation:

$$\mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r} + \beta.\mathsf{end}) \xrightarrow{\alpha} \alpha; \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r}) + \gamma; \delta; \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r}) + \beta.\mathsf{end}$$
$$\xrightarrow{\gamma} \delta; \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r} + \beta.\mathsf{end}) \quad \ldots \quad \xrightarrow{\gamma} \delta^n; \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r} + \beta.\mathsf{end})$$

States $\delta^n; \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r} + \beta.\mathtt{end})$ and $\delta^m; \mu\mathbf{r}.\alpha; (\alpha; \mathbf{r} + \gamma; \delta; \mathbf{r} + \beta.\mathtt{end})$ are bisimilar only if $n = m$. Indeed, one needs to execute n times δ (and an α) before being able to execute β. ◇

It is worth remarking that the semantics in Fig. 3 yields finite-state LTSs on global types without consecutive independent transitions, a restriction actually considered in many global type formalisms, since rule [Pass] never applies. Likewise, the semantics consisting of rules [Choice] and [Rec] only generates finite-state LTSs.

Function $\mathsf{ca}(G)$ below defines a c-automaton with subterms of G as states, G as initial state, labels in $\mathcal{L}_{\text{int}}$, and transitions inductively defined by the function $\mathsf{catr}(G)$ below:

$$\mathsf{catr}(\mathtt{end}) = \mathsf{catr}(\mathbf{r}) = \emptyset \qquad \mathsf{catr}(\mu\mathbf{r}.G) = \mathsf{catr}(G) \cup \{(\mathbf{r}, \epsilon, \mu\mathbf{r}.G), (\mu\mathbf{r}.G, \epsilon, G)\}$$

$$\mathsf{catr}(\mathbf{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i) = \bigcup_{j\in I}(\{(\mathbf{\Sigma}_{i\in I}\mathsf{p} \to \mathsf{q_i} : \mathsf{m_i}; G_i, \mathsf{p} \to \mathsf{q_j} : \mathsf{m_j}, G_j)\} \cup \mathsf{catr}(G_j))$$

▶ **Proposition 5.2.** *Let G a global type. The language of $\mathsf{ca}(G)$ coincides with the language generated by rules [Choice] and [Rec] of the semantics of G.*

Proof. First note that the languages of $\mathsf{ca}(\mu\mathbf{r}.G)$ and of $\mathsf{ca}(G[\mu\mathbf{r}.G/\mathbf{r}])$ do coincide. The only non trivial point is when the recursion variable is reached. State $\mathsf{ca}(\mu\mathbf{r}.G)$ has a transition returning to state G whilse state $\mathsf{ca}(G[\mu\mathbf{r}.G/\mathbf{r}])$ has a transition to the first state of an unfolding of G, hence the languages do coincide. We have to prove two inclusions. For the inclusion of the language of the semantics in the language of $\mathsf{ca}(G)$ the proof is by rule induction. The case for [Choice] is by construction. For rule [Rec] the thesis follows from the observation above.

For the other inclusion the proof is by structural induction on G. The only difficult case is the one of recursion, which follows from the observation above. ◀

Function $\mathsf{caPass}(G)$ below extends $\mathsf{ca}(G)$ to deal with the semantics of global types with rule [Pass]. However, the computed LTS may be infinite state, hence not a c-automaton, and in this case the function cannot be used in practice. This is, e.g., the case with the global type in Ex. 5.1. The LTS has G as initial state, labels in $\mathcal{L}_{\text{int}}$, transitions inductively defined by the function $\mathsf{catrPass}(G)$ below, and as states the ones occurring in the transitions:

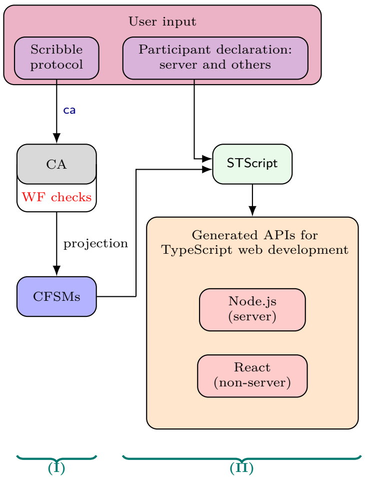
$$\begin{aligned}
\mathsf{catrPass}(\mathtt{end}) = \mathsf{catrPass}(\mathbf{r}) &= \emptyset \\
\mathsf{catrPass}(\mu\mathbf{r}.G) &= \mathsf{catrPass}(G) \cup \{(\mathbf{r}, \epsilon, \mu\mathbf{r}.G), (\mu\mathbf{r}.G, \epsilon, G)\} \\
\mathsf{catrPass}(\mathbf{\Sigma}_{i\in I}\alpha_i; G_i) &= \bigcup_{j\in I}(\{(\mathbf{\Sigma}_{i\in I}\alpha_i; G_i, \alpha_j, G_j)\} \cup \mathsf{catrPass}(G_j)) \cup \\
&\bigcup_{\alpha \text{ s.t. } G_i \xrightarrow{\alpha} G_i' \wedge \alpha_i \| \alpha \forall i\in I} \{(\mathbf{\Sigma}_{i\in I}\alpha_i; G_i, \alpha, \mathbf{\Sigma}_{i\in I}\alpha_i; G_i')\} \cup \mathsf{catrPass}(\mathbf{\Sigma}_{i\in I}\alpha_i; G_i')
\end{aligned}$$

▶ **Proposition 5.3.** *Let G a global type. The language of $\mathsf{caPass}(G)$ coincides with the language generated by the semantics of G.*

Proof. As in Prop. 5.2 we can notice that the languages of $\mathsf{ca}(\mu\mathbf{r}.G)$ and of $\mathsf{ca}(G[\mu\mathbf{r}.G/\mathbf{r}])$ do coincide.

We have to prove two inclusions. For the inclusion of the language of the semantics in the language of $\mathsf{ca}(G)$ the proof is by rule coinduction. The case for [Choice] and for [Pass] is by construction. For rule [Rec] the thesis follows from the observation above.

For the other inclusion the proof is by structural coinduction on G. The only difficult case is the one of recursion, which follows from the observation above. ◀



(a) Toolchain of CAScr

```
1  global protocol OnlineWallet
2  (role Wallet, role Customer, role Vendor) {
3    rec AuthLoop {
4      login(account: int) from Customer to Wallet;
5      pin(pin: int) from Customer to Wallet;
6      choice at Wallet {
7        login_ok() from Wallet to Customer;
8        login_ok() from Wallet to Vendor;
9        request(bill: int) from Vendor to Customer;
10       choice at Customer {
11         authorise() from Customer to Wallet;
12         pay(payment: int) from Customer to Vendor;
13       } or {
14         reject() from Customer to Wallet;
15         reject() from Customer to Vendor;
16       }
17     } or {
18       login_retry(msg: string) from Wallet to
       Customer;
19       continue AuthLoop;
20     } or {
21       login_denied(msg: string) from Wallet to
       Customer;
22     }}}
```

(b) Scribble Protocol for the OLW

Figure 4 CAScr: Toolchain and OLW Protocol

We remark that global types with infinite semantics cannot be implemented faithfully using communicating systems with the semantics in Def. 2.9. Indeed, a communicating system has a finite number of configurations, which is $O(S^n)$ where S is the size of the largest CFSM and n the number of participants.

5.2 Validating Global Protocols with Choreography Automata

The first component of our toolchain is part **(I)** in Fig. 4a; it allows the user to perform protocol specification, well-formedness checks, and the generation of CFSMs for each participant.

Let us consider the OLW example: the first step for the user is to specify the global protocol, `OnlineWallet.scr` (Fig. 4b), in the *Scribble protocol description language*, often referred to as "the practical incarnation of multiparty session types" [20, 37]. The syntax of Scribble (`http://www.scribble.org`, `https://nuscr.dev/`) has a straightforward correspondance to the syntax of global types, so Scribble implementations of communicating processes will be supported by multiparty session type theory, and inherit its semantic guarantees. Our development for part **(I)** of the toolchain is based on the νScr implementation [47], but fundamentally differs from this (and other Scribble versions) in two aspects:

- the underlying choreographic objects—normating the communication among multiple participants—are not global types, but c-automata, and
- we allow for participants to join the communication at later stage, only in branches where they are needed (selective participation).

Fig. 4b shows the protocol `OnlineWallet.scr` for the OLW. Noteworthy, unlike νScr, we can specify the selective participation of the vendor. In particular, the vendor participant is involved only in the first branch of the choice (lines 7-12), namely on successful login.

After its specification, the Scribble protocol is translated into a c-automaton, with the implementation of the function ca from § 5.1 (this is exactly the c-automaton from Ex. 2.5, § 2). On this automaton, well-sequencedness and well-branchedness checks are performed. If the c-automaton passes the above well-formedness checks, it is then projected onto each

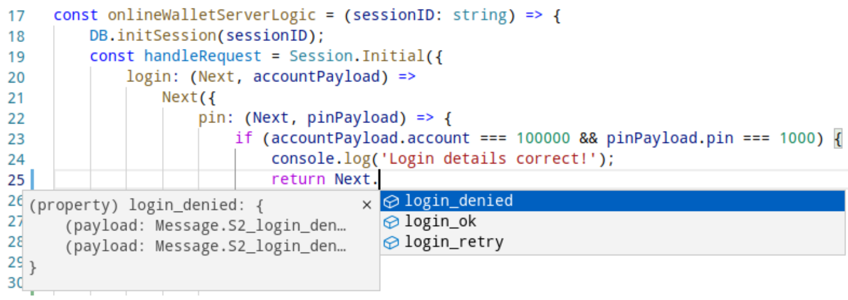


Figure 5 Implementation with the API for wallet in Visual Studio Code

participant (Def. 2.7), thus obtaining a collection of CFSMs, whose semantics is equivalent to the one of the original c-automaton. Both global c-automata and local CFSMs are represented using the DOT graph description language. Ex. 2.8 from § 2 shows the CFSM obtained by projection on vendor of the c-automaton for the OLW; for the other participants analogous CFSMs are obtained. The local CFSM representations provide the communication behaviour of each participant and, as such, they retain all the information for obtaining deadlock-free endpoint implementations. Each CFSM is the projection of the global c-automaton onto one of the communicating participants; from this local automaton, the API for the implementation of the participant is generated.

5.3 API Generation for Distributed Web Development

Our chosen domain of application is *distributed web development*. By nature, web services are distributedly developed and feature communication among multiple participants. In services where some courses of actions are optional, it is likely that the participation of some role is also optional (selective participation). Our OLW example is a minimal, yet representative example that selective participation is commonplace in transactions, auctions, or contracts. For instance, Kickstarter [29] is a worldwide popular crowdfunding platform where the money of *supporters* is given to a project *initiator* only if the initially set goal is met; otherwise the money is returned to supporters. In other words, when the deadline is passed, if the goal is met, only the initiator is involved in the communication, if not, only the supporters are.

More technically, our development builds on and extends STScript [35]. We target server-centric protocols (based on the WebSocket standard [13]), where one role is chosen as privileged, the *server*. The generated APIs are compatible with the Node.js runtime for server-side endpoints and the React.js framework for browser-side endpoints. The STScript toolchain in [35] is based on the multiparty session type theory, where there is no privileged role; hence which role is the server has to be declared by the user. The same holds for our development based on c-automata. We have discussed in the previous section how the Scribble protocol in input is translated into a c-automaton and, once well-formedness checks are performed, projected onto a CFSM for each participant. This CFSM is passed to the code-generation component of our toolchain (part **(II)** of Fig. 4a), together with the role in input and the information about whether it is the server role or not.

Fig. 5 shows an example of the usage of the generated API, when implementing the participant wallet in Visual Studio Code (`https://code.visualstudio.com/`). The autocomplete function of the editor offers the developer appropriate options, so that the implementation of the login choice abides by the global discipline of the OnlineWallet protocol.

From an engineering point of view, for developing the part **(I)** of the toolchain (Fig. 4a) we have adapted to our theory of c-automata, the codebase of νScr: a recent implementation of Scribble that offers a toolchain for "language-independent code generation" [47]. However, νScr itself does not provide direct support for TypeScript. Hence, the development of part **(II)** in Fig. 4a integrates the νScr codebase with STScript. This is is a Scribble extension—also based on multiparty session types, but relying on the ScribbleJava implementation `http://www.scribble.org`. Building the API-generation of CAScr on top of the one of STScript has been a conventient choice: STScript targets distributed web development directly and offers a full implementation for generating TypeScript APIs from νScr-projected CFSMs.

The result of our development is CAScr, of which we list the distinctive features.

- *Scope.* CAScr specifically targets TypeScript and enables safe distributed web development.
- *Input.* The user specifies the global protocol in the Scribble language and picks one of the communicating participants as the server.
- *Correctness.* CAScr relies on the flexible theory of c-automata: the protocol in input is translated into a c-automaton, which, if well-formed, is then projected onto CFSMs.
- *APIs Generation.* From each CFSM, CAScr generates the TypeScript API for the respective role.
- *Safe Endpoint Implementation.* The distributed implementation of the participants, using the generated APIs, is guaranteed to be deadlock and lock free by the underlying theory.

In our first implementation of CAScr (`https://github.com/Tooni/CAScript-Artifact`), we provide three simple examples: an "adder" (the client sends to the server, in a loop, two numbers to be added), a simple contract protocol, and the OLW, which we have used as a running example, since it carries and shows all the core features of our novel theory, and, in particular, selective participation (see also the discussion at the beginning of this section). Furthermore, we have provided a small tutorial in the README file of CAScr, to guide the user through the implementation of their own protocols.

It is worth mentioning that a first extension of CAScr is under development (see also Appendix A): current implementation, based on previous work [48], allows the generation of APIs for Scribble protocols with assertions. However, the necessary extension of the function ca in § 5.1 to assertions, as well as subsequent consistency checks, have not been implemented yet. While conceptually straightforward, in practice one needs to integrate the CAScr toolchain with tools manipulating logical formulae such as SAT solvers in order to implement the check for the consistency property (cf. Def. 4.16).

To conclude, we have developed the first version of Scribble based on *choreography automata*. It improves on the flexibility of traditional implementations of multiparty session types, by accomodating for *selective participation*, and it integrates previous developments with our new theory: the νScr toolchain with the TypeScript support provided by STScript. On the one hand, our toolchain enables verified communication for web development with selective participation, on the other hand it paves the road to interesting extensions, e.g., fully capturing the asynchronous semantics of websockets (see § 7), or supporting assertions and the design-by-contract approach, as discussed above.

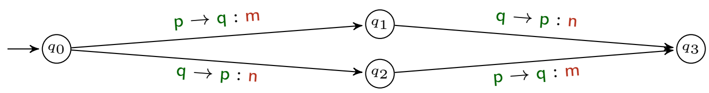
6 Related Work

Conditions similar to our well-branchedness and well-sequencedness arise naturally in investigations on choreographies and their realisability. Uniqueness of choice selector is commonly imposed syntactically (as in § 5.1) in several multiparty session types (MPSTs) formalisms (e.g., [21, 3, 9, 44, 48]) and also adopted in global graphs [11, 45], and in choreography languages in general (cf. the notion of *dominant role* in [40]). Also, notions close to well-sequencedness occur quite naturally in "well-behaved" choreographies (e.g., the notion of *well-informedness* of [6] in collaboration diagrams). A distinguishing element of our notion of well-branchedness is that we admit protocols where disjoint groups of participants may concurrently engage in a choice. This generalises (and corrects) the notion of well-branchedness in [2] and, to the best of our knowledge, is not supported in any other choreographic framework.

Global graphs [11, 16, 45, 32] are another model of global specifications. We refer the reader to [2] for a comparison between c-automata and global graphs.

The first work advocating a design-by-contract framework for MPSTs is [3]. Asserted c-automata have been strongly inspired by it. In particular, our notion of consistency (cf. Def. 4.16) can be seen as a generalisation of *well-assertedness* in [3]. More recently, ideas similar to the one in [3] have been developed in [48], where refined MPSTs have been proposed. The results of these papers are in the vein of guaranteeing properties of programs by a behavioural type system ensuring communication soundness in presence of data dependencies.

Besides the added flexibility of c-automata with respect to structured formalisms discussed in the Introduction, ac-automata do not suffer from the constraints imposed on global types in [3, 48]. More precisely, interactions guarding choices in [3, 48] syntactically restrict to a unique partner of the *selector* (i.e., the participant choosing the branch to follow). On the contrary, (asserted) c-automata do not have such restriction. For instance,



is a well-branched c-automaton which would be ruled out by all the choreography models based on global types we are aware of. Both [3, 48] rely on a merge operator to guarantee well-formedness (and projectability) of global types. This is an obstacle for selective participation which our notion of well-branchedness (cf. Def. 3.7) overcomes. We also note that our notion of knowledge is more general than the one in [3]. In fact, as observed in [48], the notion of history sensitivity in [3] does not allow a participant to know variables fixed in interactions it is not involved in. Like for refined MPSTs, asserted c-automata do not have this limitation and can in fact deal with protocols like the one in Example 4.1 in [48].

Our theoretical work sees its first application in the development of CAScr, a toolchain for communication-safe web development. CAScr takes the popular *top-down approach*, following the original methodology of MPSTs [21]. The top-down approach enables *correctness-by-construction*: a developer provides a global description for the whole protocol; by projecting the global protocol, APIs are generated from local CFSMs, which ensure the safe implementation of each participant. MPSTs toolchains that take the top-down approach have seen multiple implementations and targeted a variety of mainstream programming languages, such as (in no particular order) Java [23, 24, 30], OCaml [26], Go [8], Scala [42, 46], F# [36], F* [48] and Rust [10, 31]. Like CAScr, most of the above implementations rely on the Scribble protocol description language [20, 37, 47] (`http://www.scribble.org`, `https://nuscr.dev/`). More relevant to this work is [35], in which the authors develop STScript, a full toolchain that applies such top-down methodology and targets TypeScript for web development.

All the implementations above are based on MPSTs; they exploit the equivalence between local types and CFSMs [11, 12] to generate APIs for all the participants. In [24], *explicit connections*, similar to our selective participation, have been introduced in Scribble, and more recently [18] uses an analogous approach to implement adaptations for an actor domain-specific language. Both [24] and [18] need to add explicit disconnections and connections to the syntax of Scribble. In CAScr (§ 5), we have integrated the theory of c-automata into the

νScr toolchain [47], to allow for more flexible protocols, where participants may appear only in selected branches after a choice, with no need to change the Scribble syntax.

7 Conclusion and Future Work

We have presented a flexible framework to describe protocols in a setting of c-automata combining selective participation to branches of choices and assertions supporting design-by-contract. This allows us to model non trivial examples such as the OnLineWallet, and ensures faithful realisability. In fact, we exploited the flexibility of c-automata to generalise well-branchedness (so to account for selective participation) and to transfer the DbC approach [3] (so to account for *data-aware* protocols). Remarkably, the fact that c-automata are finite-state models does not allow us to fully capture Scribble. Nonetheless, a semi-decidable approach has been considered (cf. § 5.1) which becomes effective when restricting to protocols without interplay between consecutive independent interactions and recursion. More precisely, it should not be possible to split a recursive protocol into groups of interactions with disjoint participants. This restriction mildly affects applicability: indeed, to faithfully implement such specifications one would need infinite-state systems of CFSMs, while ours are finite-state. Also, a clear advantage of our approach is that we can verify more general conditions for Scribble specifications that can be faithfully mapped on c-automata.

We implemented our theory by allowing Scribble protocols to be translated into c-automata, checked for well-formedness, and finally used to derive APIs for TypeScript programming. The flexibility of c-automata has been instrumental to capture Scribble [20, 37, 47] specifications. Scribble notation (and semantics) may be not easy to grasp for practitioners as it involves a non-trivial amount of technicalities. Hence, defining and understanding well-formedness conditions on Scribble could not be straightforward.

Our framework can be immediately used in practice in interesting examples: the design of a variety of existing web services (e.g., for authentication or transactions) include selective participation; with the OLW implementation, we witness how protocols carrying this feature can be specified in CAScr (which from these generates APIs for implementations). Nonetheless, we envisage some extensions (see § 5.3 and Appendix A for details).

Our focus is on selective participation and design-by-contract. Hence, for simplicity, we consider synchronous semantics. CAScr builds instead on an asynchronous implementation of Scribble [35], which makes our results applicable only to protocols in which asynchronous executions do not break the causal relations imposed by the synchronous semantics so that choices are affected. This is the case for the case studies in the artifact, including our running example OLW. The discrepancy disappears if a synchronous transport layer (e.g., http) replaces web sockets. To increase the applicability of CAScr—and also because of its theoretical interest, we plan to extend the results to cover an asynchronous communication model based on queues. While the general structure of the theory remains the same, well-branchedness needs to be updated since send and receive actions would not be symmetric anymore. E.g., a participant that only occurs in one branch of a choice, thanks to selective participation, needs to interact with a fully-aware participant by performing a receive, while right now it can also interact through a send action. We conjecture that the extension to asynchronous semantics does not affect the treatment of DbC in ac-automata. In fact, assertions are guaranteed by the sender and relied upon by the receiver (hence, the nature of communication is orthogonal to the flow of data).

Our methodology follows the top-down software development approach of choreographies (cf. § 1 and § 6). An interesting direction for future work is to develop an analysis of existing

APIs; for instance, by extracting an abstract representation of the API, its conformance could be checked against a projection of the global specification. Such design would improve on the applicability of our theory, for analysing and reusing existing developments.

References

1 Marco Autili, Paola Inverardi, and Massimo Tivoli. Automated synthesis of service choreographies. *IEEE Softw.*, 32(1):50–57, 2015. doi:10.1109/MS.2014.131.

2 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020.

3 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In Paul Gastin and François Laroussinie, editors, *Concur 2010*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.

4 Jonas Bonér. *Reactive Microsystems - The Evolution Of Microservices At Scale*. O'Reilly, 2018.

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

6 Tevfik Bultan and Xiang Fu. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008. doi:10.1007/s11761-008-0022-7.

7 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

8 David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in Go: Statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290342.

9 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.

10 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:10.1145/3503221.3508404.

11 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. URL: http://dx.doi.org/10.1007/978-3-642-28869-2_10, doi:10.1007/978-3-642-28869-2_10.

12 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, pages 174–186, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

13 Ian Fette and Alexey Melnikov. The websocket protocol, 2011. URL: https://www.rfc-editor.org/info/rfc6455.

14 Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.

15 Leonardo Frittelli, Facundo Maldonado, Hernán C. Melgratti, and Emilio Tuosto. A choreography-driven approach to APIs: The OpenDXL case study. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020,*

Proceedings, volume 12134 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2020. doi:10.1007/978-3-030-50029-0_7.

16 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *J. Log. Algebraic Methods Program.*, 108:69–89, 2019.

17 Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL: https://rfc-editor.org/rfc/rfc6749.txt, doi:10.17487/RFC6749.

18 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14053, doi:10.4230/LIPIcs.ECOOP.2021.10.

19 Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.

20 Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega Ojo, editors, *Distributed Computing and Internet Technology*, pages 55–75, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-19056-8_4.

21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL*, pages 273–284. ACM Press, 2008.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

23 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint api generation. In Perdita Stevens and Andrzej Wąsowski, editors, *Fundamental Approaches to Software Engineering*, pages 401–418, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

24 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*, page 116–133, Berlin, Heidelberg, 2017. Springer-Verlag. doi:10.1007/978-3-662-54494-5_7.

25 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

26 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13166, doi:10.4230/LIPIcs.ECOOP.2020.9.

27 Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217. Working Draft 17 December 2004.

28 Kerberos 5. https://web.mit.edu/kerberos/krb5-1.19/. Accessed: 14/02/2022.

29 Kickstarter. https://www.kickstarter.com/about. Accessed: 14/02/2022.

30 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, page 146–159, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2967973.2968595.

31 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types. In *36th European Conference on Object-Oriented Programming*, LIPIcs, 2022. in this volume.

32 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd*

Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 221–232. ACM, 2015.

33 Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.

34 Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992. `doi:http://dx.doi.org/10.1109/2.161279`.

35 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 94–106, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3446804.3446854`.

36 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: Compile-time API generation of distributed protocols with refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 128–138, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3178372.3179495`.

37 Rumyana Neykova and Nobuko Yoshida. *Featherweight Scribble*, volume 11665 of *LNCS*, pages 236–259. Springer, Cham, 2019. `doi:10.1007/978-3-030-21485-2_14`.

38 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local verification of global protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

39 Object Management Group. Business Process Model and Notation. `http://www.bpmn.org`.

40 Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*, pages 973–982, 2007.

41 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

42 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2017/7263`, `doi:10.4230/LIPIcs.ECOOP.2017.24`.

43 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *PACMPL*, 3(POPL):30:1–30:29, 2019.

44 Paula Severi and Mariangiola Dezani-Ciancaglini. Observational equivalence for multiparty sessions. *Fundamenta Informaticae*, 170:267–305, 2019. URL: `http://www.di.unito.it/~dezani/papers/sd19.pdf`.

45 Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *J. Log. Algebr. Meth. Program.*, 95:17–40, 2018.

46 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. `doi:10.1145/3485501`.

47 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In Evripidis Bampis and Aris Pagourtzis, editors, *Fundamentals of Computation Theory*, pages 18–35, Cham, 2021. Springer International Publishing.

48 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. In *OOPSLA 2020: Conference on Object-Oriented Programming Systems, Languages and Applications*, number OOPSLA (Article 148) in PACMPL, page 30 pages, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3428216`.

```
1  (*# CheckDirectedChoiceDisabled, RefinementTypes, ValidateRefinementProgress,
       ValidateRefinementSatisfiability #*)
2
3  global protocol OnlineWallet(role Wallet, role Customer, role Vendor) {
4    rec AuthLoop [try<Customer, Wallet>: int = 0] {
5      login(account: int{account >= 100000 && account < 1000000}) from Customer to
       Wallet;
6      pin(pin: int{pin >= 1000 && pin < 10000}) from Customer to Wallet;
7      choice at Wallet {
8        login_ok() from Wallet to Customer;
9        login_ok() from Wallet to Vendor;
10       request(bill: int{bill > 0}) from Vendor to Customer;
11       choice at Customer {
12         authorise() from Customer to Wallet;
13         pay(payment: int{payment = bill}) from Customer to Vendor;
14       } or {
15         reject() from Customer to Wallet;
16         reject() from Customer to Vendor;
17       }
18     } or {
19       login_retry(msg: string{try < 3}) from Wallet to Customer;
20       continue AuthLoop [try + 1];
21     } or {
22       login_denied(msg: string{try = 3}) from Wallet to Customer;
23     }
24   }
25 }
```

Figure 6 Scribble Protocol for the OLW with Assertions

A Extending the Toolchain to Design-by-Contract

In § 4 we have shown how our c-automata theory can be endowed with assertions, thus supporting design-by-contract.

We have started engineering an extension of CAScr, which combines selective participations and design-by-contract. In [48], the authors extend the Scribble language with assertions (*refinements*); we integrate a similar approach in our specification language. As an example, we show here a description of the OLW protocol with annotated assertions (Fig. 6). Assertions are used, for example, (line 5) to enforce that the integer account is a six-digit number, or to allow for a finite number of login attempts: the integer try is initiated (line 4) as 0; then incremented at each following attempt (lines 19 and 20); finally, when try $= 3$ (line 22), the login is denied. Such prototype extension has allowed us to combine design-by-contract and selective participation in Scribble protocols for selected examples, and to generate TypeScript APIs for multiple participants, with assertions to guide the developer's implementation process. A future extension of the function ca in § 5.1, to assertions, will allow us to have a more comprehensive version of CAScr.