An Abstract Framework for Choreographic Testing*

Alex Coto
GSSI, Italy
alex.coto@gssi.it

Roberto Guanciale
KTH, Sweden
robertog@kth.se

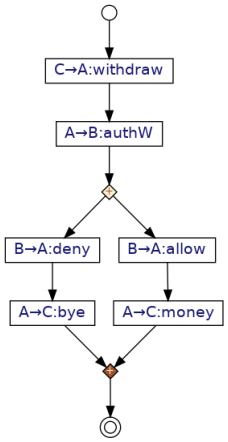
Emilio Tuosto
GSSI, Italy and Univ. of Leicester, UK
emilio.tuosto@gssi.it

We initiate the development of a model-driven testing framework for message-passing systems. The notion of *test* for communicating systems cannot simply be borrowed from existing proposals. Therefore, we formalize a notion of suitable distributed tests for a given choreography and devise an algorithm that generates tests as projections of global views. Our algorithm abstracts away from the actual projection operation, for which we only set basic requirements. The algorithm can be instantiated by reusing existing projection operations (designed to generate local implementations of global models) as they satisfy our requirements. Finally, we show the correctness of the approach and validate our methodology via an illustrative example.

1 Introduction

We propose model-driven testing to complement the *correctness-by-construction* principle of choreographies. We introduce a testing approach based on choreographies which we deem suited to develop model-driven testing that may help to tame the problems of correctness of distributed applications.

Context In the quest for correct-by-construction solutions, formal choreographic models have proven themselves to be valuable approaches. These models are gaining momentum, for instance, in the context of business processes and message-passing applications. The fundamental idea of choreographic models (originally proposed by WS-CDL [18]) is that specifications of systems consist of *global* and *local views*. The global view describes the behaviour of a system in terms of the interactions among (the *role*) of components. The diagram below is an example of a global view of a protocol; we will use this as a running example throughout the paper.



This protocol is a simplified view of the main interactions that a client C willing to withdraw some cash has to perform, together with an ATM A, and a bank B. The protocol starts with the *interaction* $\mathsf{C} \xrightarrow{\texttt{withdraw}} \mathsf{A}$ with which participant C instructs the ATM A about the intention to withdraw some cash. In the next interaction $\mathsf{A} \xrightarrow{\texttt{authW}} \mathsf{B}$, A asks the bank B to authorise the withdrawal. Observe that payloads are abstracted away; for instance, the message `withdraw` is intended to be a data type carrying e.g., the amount of requested cash. A distributed choice starts at the *branching* point ◈, where the bank B decides whether to deny or grant the withdrawal. Note that the choice is non-deterministic since, besides from data, this model abstracts away from local computations. Depending on the local decision of B, the next interaction is either $\mathsf{B} \xrightarrow{\texttt{deny}} \mathsf{A}$ or $\mathsf{B} \xrightarrow{\texttt{allow}} \mathsf{A}$. In each case the client is notified of the decision with interactions $\mathsf{A} \xrightarrow{\texttt{bye}} \mathsf{C}$ (in the first case) or $\mathsf{A} \xrightarrow{\texttt{money}} \mathsf{C}$ if the operation is granted by the bank.

* Research partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233 , MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems) and the TrustFull project, funded by the Swedish Foundation for Strategic Research

J. Lange, A. Mavridou, L. Safina, A. Scalas (Eds.):
13th Interaction and Concurrency Experience (ICE 2020).
EPTCS 324, 2020, pp. 43–60, doi:10.4204/EPTCS.324.5

A main source of problems in distributed protocols is reaching consensus among participants in distributed choices. Indeed, participants have partial knowledge about the global state of the protocol. And, for the protocol to run "smoothly", the partial knowledge of each participant should be consistent with respect to the global state of the protocol. For distributed choices this boils down to require *awareness* of each participant about the branch to follow. For instance, in the example above, the bank is aware of the choice since it decides what to do next and the other participants become aware of the choice from the messages they exchange.

The correctness-by-construction principle of choreographic models is usually realised through the identification of *well-formedness* conditions on global views. These are sufficient conditions guaranteeing that the protocol can be executed distributively, without breaking the consistency between the global state and the local knowledge of participants. In particular, formal choreographic approaches (such as [17, 10, 14, 15, 11, 7] to mention a few) study notions of well-formedness to guarantee the safety of communications (usually, deadlock-freedom, no message losses, etc.). The local view of a protocol indeed provides a specular specification of the behaviour of (the role of) each component "in isolation". In this way, the local view yields a set of computational units enacting the communications specified in the global view. For instance, the local view of the bank B above consists of an artefact waiting for a message `authW` from A to which it replies by sending either of the messages `deny` or `allow`. Note that the client and the bank are "oblivious" of each other, in the sense that they interact only with the ATM.

The typical scheme to realise the correctness-by-construction principle consists of the steps below:

1. provide an artefact defining the global view of the system;
2. revise the global view until well-formedness is achieved;
3. project global views into local views;
4. verify that code implementing the local view of a component complies with its projection.

(It is also possible to avoid step (4) and project global views directly on code.) Steps (1) and (2) are mainly human activities, although some algorithmic support[1] is offered by the verification of well-formedness conditions. The remaining steps can instead be supported by algorithms. In fact, (an approximation of) compliance is usually decidable and projections can often be straightforwardly computed by "splitting" interactions into complementary send and receive actions.

Problem Although paramount for the development of message-passing applications, the correctness-by-construction principle advocated by formal choreographies is not enough. At first sight this utterance may look controversial. In fact, we do not contend that correctness-by-construction is not worth pursuing (or not achievable: many models including those mentioned above do realise the correctness-by-construction principle). But, even in a correctly implemented choreographic solution problems may arise. We list three major causes of possible disruption.

Local computation As said, formal choreographies focus on the interactions among components while abstracting away from local computations. Therefore, errors may still be introduced when developing code; for instance, a component expected to receive an integer and return a string, after inputting the integer may diverge on a local computation before delivering the expected string and cause a malfunction in the communication protocol.

[1] Some authors have considered the problem of supporting designers in the identification of problems in non well-formed choreographies [3, 4, 19].

Evolution Software is often subject to continuous changes for instance to increase efficiency or to accommodate evolving requirements. For example, to reduce the communication overhead, a component may be modified so that two outputs are merged into one so to spare an interaction. Besides introducing bugs in the new code for local computations, these changes may alter the original design breaking the compliance required in step (4) of the scheme above.

Openness Increasingly, applications are built by composing computational elements developed independently and available off-the-shelf, over which the developer might have no control. This is for instance the main approach to develop service-oriented architectures. New releases or modifications of third-party components (libraries, run-time support, etc.) may introduce malfunctions in applications using it. For example, a new release of a service invoked by an application may enrich the spectrum of possible messages delivered to some components not designed to handle such new messages.

Contribution We take a first step to equip known choreography-based approaches with testing. More precisely, we start addressing step (3) above. Our main technical contribution is an algorithm to automatically derive (abstract) test cases out of a well-formed choreography (cf. Section 3.2). We develop our results in the setting of *global choreographies* [14, 26] and *communicating finite state machines* [6]. The former is the model we adopt to represent global views and the latter is a well-known model for specifying communication protocols that will serve to represent local views.

Our key contributions are:

- An abstract framework of well-formedness that captures the essential elements of formal choreographic models. This abstract framework makes our algorithm parametric with respect to the notion of well-formedness.
- We lay down the definitions that transfer various notions of (standard) software testing to communication protocols. Formally this is done by adapting a few concepts from traditional software testing such as the notions of test (Definition 4), oracle (Definition 7), and test compliance (Definition 5). Again, the abstract framework paves the way for several alternative developments. We decided to explore one of them first; we discuss alternatives in Section 5.
- As we will see, not all test cases are "meaningful", therefore we identify when tests are *suitable* for a choreography (Definition 6).
- We apply our framework to a non-trivial example (cf. Section 4).

2 Background

We survey the main definitions and constructs needed in the rest of the paper. We focus on *global choreographies* (g-choreographies for short) for the global view [15], and borrow from [6] *communicating finite-state machines* (CFSMs) for the local views. G-choreographies were chosen because they offer an intuitive visual description together with a precise semantics [26, 16]. We adopt CFSMs because they have many similarities with programming languages based on message-passing, such as Erlang.

2.1 Global Choreographies

The global view of a choreography can be suitably specified as a *global choreography* [26, 14, 13]. This model is appealing as it has a syntactic and diagrammatic presentation, and has been given a formal

semantics in terms of pomsets, which enable for automatic processing.

Fix a set $\mathscr{P}$ of *participants* and a set $\mathscr{M}$ of *message (types)* such that $\mathscr{P} \cap \mathscr{M} = \emptyset$; let $\mathsf{A}, \mathsf{B}, \ldots$ range over $\mathscr{P}$ and $\mathtt{m}, \mathtt{n}, \ldots$ range over $\mathscr{M}$. A global choreography (or g-choreography) is a term derivable from the following grammar:

$$\begin{array}{rcll} \mathsf{G} & ::= & (\mathsf{o}) & \text{empty} \\ & \mid & \mathsf{A} \xrightarrow{\mathtt{m}} \mathsf{B} & \text{interaction} \\ & \mid & \mathsf{G} \mid \mathsf{G} & \text{fork} \\ & \mid & \mathsf{G} + \mathsf{G} & \text{choice} \\ & \mid & \mathsf{G}; \mathsf{G} & \text{sequential} \\ & \mid & \mathsf{repeat}\ \mathsf{G} & \text{iteration} \end{array}$$

The empty choreography (o) yields no interactions; trailing occurrences of (o) may be omitted. An interaction $\mathsf{A} \xrightarrow{\mathtt{m}} \mathsf{B}$ represents the exchange of a message of type $\mathtt{m}$ between A and B, provided that $\mathsf{A} \neq \mathsf{B}$. We remark that data are abstracted away: in $\mathsf{A} \xrightarrow{\mathtt{m}} \mathsf{B}$, the message $\mathtt{m}$ is not a value and should rather be thought of as (the name of) a data type[2]. G-choreographies can be composed sequentially or in parallel ($\mathsf{G}; \mathsf{G}'$ and $\mathsf{G} \mid \mathsf{G}'$). A (non-deterministic) choice $\mathsf{G}_1 + \mathsf{G}_2$ specifies the possibility to continue according to either G_1 or G_2. The body G in an iteration $\mathsf{repeat}\ \mathsf{G}$ is repeated until a participant in G (non-deterministically) chooses to exit the loop. Although for simplicity we do not consider iterative g-choreographies in our examples, the techniques we introduce further on can work on arbitrary finite unfoldings of the loops, as is commonplace in software testing or in verification techniques such as bounded model-checking.

Example 1. *The g-choreography for the example introduced in Section 1 is*

$$\mathsf{G}_{ATM} = \mathsf{C} \xrightarrow{\mathtt{withdraw}} \mathsf{A}; \mathsf{A} \xrightarrow{\mathtt{authW}} \mathsf{B}; \{\mathsf{B} \xrightarrow{\mathtt{deny}} \mathsf{A}; \mathsf{A} \xrightarrow{\mathtt{bye}} \mathsf{C} + \mathsf{B} \xrightarrow{\mathtt{allow}} \mathsf{A}; \mathsf{A} \xrightarrow{\mathtt{money}} \mathsf{C}\}$$

where we assume that sequential composition takes precedence over choice.

The semantics of a g-choreography as defined in [14, 26] is a family of pomsets (partially ordered multisets); each pomset in the family is the partial order of events occurring on a particular "branch" of the g-choreography. Events are therefore labelled by *(communication) actions* l occurring in the g-choreography. The output of a message $\mathtt{m} \in \mathscr{M}$ from participant $\mathsf{A} \in \mathscr{P}$ to participant $\mathsf{B} \in \mathscr{P}$ is denoted by $\mathsf{A}\,\mathsf{B}!\mathtt{m}$, while the corresponding input is denoted by $\mathsf{A}\,\mathsf{B}?\mathtt{m}$. More formally,

$$\mathscr{L}_{\text{act}} = \{\mathsf{A}\,\mathsf{B}!\mathtt{m}, \mathsf{A}\,\mathsf{B}?\mathtt{m} \mid \mathsf{A}, \mathsf{B} \in \mathscr{P} \text{ and } \mathtt{m} \in \mathscr{M}\}$$

is the set of (communication) *actions* and l ranges over $\mathscr{L}_{\text{act}}$. The subject of an action is defined as $\mathsf{sbj}(\mathsf{A}\,\mathsf{B}!\mathtt{m}) = \mathsf{A}$ and $\mathsf{sbj}(\mathsf{A}\,\mathsf{B}?\mathtt{m}) = \mathsf{B}$.

It is not necessary to restate here the whole constructions for the semantics which is given by induction on the structure of the g-choreography; we simply give an informal account. The semantics $[\![(\mathsf{o})]\!]$ is the set $\{\varepsilon\}$ containing the empty pomset ε, while for interactions we have

$$[\![\mathsf{A} \xrightarrow{\mathtt{m}} \mathsf{B}]\!] = \left[\, \mathsf{A}\,\mathsf{B}!\mathtt{m} \longrightarrow \mathsf{A}\,\mathsf{B}?\mathtt{m} \,\right]$$

namely, the semantics of an interaction is a pomset where the output event precedes the input event. The semantics of the other operations is basically obtained by composing the semantics of sub g-choreographies. More precisely,

[2] We leave implicit the grammar of data types; in the examples we will assume that $\mathtt{m}$ ranges over basic types such as `int`, `bool`, `string`, etc.

- for a choice we essentially have $[\![\mathsf{G} + \mathsf{G}']\!] = [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!]$;
- the semantics of the parallel composition $\mathsf{G} \mid \mathsf{G}'$ is essentially built by taking the disjoint union of each pomset in $[\![\mathsf{G}]\!]$ with each one in $[\![\mathsf{G}']\!]$;
- the semantics of the sequential composition $[\![\mathsf{G};\mathsf{G}']\!]$ is the disjoint union of each pomset in $[\![\mathsf{G}]\!]$ with each one in $[\![\mathsf{G}']\!]$ and, for every participant A, making every output of A in $[\![\mathsf{G}]\!]$ precede all events of A in $[\![\mathsf{G}']\!]$.

Example 2. *Consider G_{ATM} of Example 1. We have*

$$[\![\mathsf{G}_{ATM}]\!] = \left\{ \begin{array}{l} \left[\mathtt{CA!withdraw} \longrightarrow \mathtt{CA?withdraw} \longrightarrow \mathtt{AB!authW} \longrightarrow \mathtt{AB?authW} \longrightarrow \mathtt{BA!deny} \longrightarrow \mathtt{BA?deny} \longrightarrow \mathtt{AC!bye} \longrightarrow \mathtt{AC?bye} \right], \\ \left[\mathtt{CA!withdraw} \longrightarrow \mathtt{CA?withdraw} \longrightarrow \mathtt{AB!authW} \longrightarrow \mathtt{AB?authW} \longrightarrow \mathtt{BA!allow} \longrightarrow \mathtt{BA?allow} \longrightarrow \mathtt{AC!money} \longrightarrow \mathtt{AC?money} \right] \end{array} \right\}$$

For the sake of illustration, the singleton

$$\left\{ \left[\begin{array}{l} \mathtt{CA!withdraw} \longrightarrow \mathtt{CA?withdraw} \longrightarrow \mathtt{AB!authW} \longrightarrow \mathtt{AB?authW} \longrightarrow \mathtt{BA!deny} \longrightarrow \mathtt{BA?deny} \longrightarrow \mathtt{AC!bye} \longrightarrow \mathtt{AC?bye} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \searrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{BA!allow} \longrightarrow \mathtt{BA?allow} \longrightarrow \mathtt{AC!money} \longrightarrow \mathtt{AC?money} \end{array} \right] \right\}$$

is the semantics of the g-choreography obtained by replacing choice with parallel composition in G_{ATM}.

The language of a g-choreography G, written $\mathscr{L}[\mathsf{G}]$, is the closure under prefix of the set of all *linearizations* of $[\![\mathsf{G}]\!]$ where a linearisation of a pomset is a permutation of its events that preserves the order of the pomset.

Example 3. *The language of the last pomset in Example 2 is the set of prefixes of words obtained by concatenating* $\mathsf{CA}!\mathtt{withdraw}\ \mathsf{CA}?\mathtt{withdraw}\ \mathsf{AB}!\mathtt{authW}\ \mathsf{AB}?\mathtt{authW}$ *with both* $\mathsf{BA}!\mathtt{deny}\ \mathsf{BA}?\mathtt{deny}\ \mathsf{AC}!\mathtt{bye}\ \mathsf{AC}?\mathtt{bye}$ *and* $\mathsf{BA}!\mathtt{allow}\ \mathsf{BA}?\mathtt{allow}\ \mathsf{AC}!\mathtt{money}\ \mathsf{AC}?\mathtt{money}$.

2.2 Communicating Systems

As in [20, 13], we adopt *communicating finite state machines* (CFSMs) as local artefacts. We borrow the definition of CFSMs in [6] adapting it to our context. A CFSM $M = (Q, q_0, \rightarrow)$ is a finite transition system where

- Q is a finite set of *states* with *initial* state $q_0 \in Q$, and
- $\rightarrow \subseteq Q \times \mathscr{L}_{\text{act}} \times Q$; we write $q \xrightarrow{\ell} q'$ for $(q, \ell, q') \in \rightarrow$.

Machine M is *local* to a participant $\mathsf{A} \in \mathscr{P}$ (or A-*local*) if $\mathsf{sbj}(\ell) = \mathsf{A}$ for each transition $q \xrightarrow{\ell} q'$ of M. A *(communicating) system* is a map $S = (M_\mathsf{A})_{\mathsf{A}\in\mathscr{P}}$ where $M_\mathsf{A} = (Q_\mathsf{A}, q_{0\mathsf{A}}, \rightarrow_\mathsf{A})$ is a A-local CFSM for each $\mathsf{A} \in \mathscr{P}$. The set of *channels* (fixed for all communicating systems) is $\mathscr{C} = \{\mathsf{AB} \mid \mathsf{A} \neq \mathsf{B} \in \mathscr{P}\}$; for all $\mathsf{AB} \in \mathscr{C}$, it is assumed that there is an unbound finite multiset $b_{\mathsf{AB}} = \{\!|\, \mathtt{m}_1, \ldots, \mathtt{m}_n \,|\!\}$ containing the messages that M_A sends to M_B and from which M_B consumes the messages sent by M_A. We use $\_ \uplus \_$ for multiset union and $\_ - \_$ for multiset difference.

The semantics of communicating systems is defined in terms of *transition systems*, which keep track of the state of each machine and the content of each buffer. Let $S = (M_\mathsf{A})_{\mathsf{A}\in\mathscr{P}}$ be a communicating system. A *configuration* of S is a pair $s = \langle \vec{q}\ ;\ \vec{b} \rangle$ where $\vec{q} = (q_\mathsf{A})_{\mathsf{A}\in\mathscr{P}}$ with $q_\mathsf{A} \in Q_\mathsf{A}$ and $\vec{b} = (b_{\mathsf{AB}})_{\mathsf{AB}\in\mathscr{C}}$ mapping each channel to a multiset of messages; q_A keeps track of the *local state* of machine M_A in s and

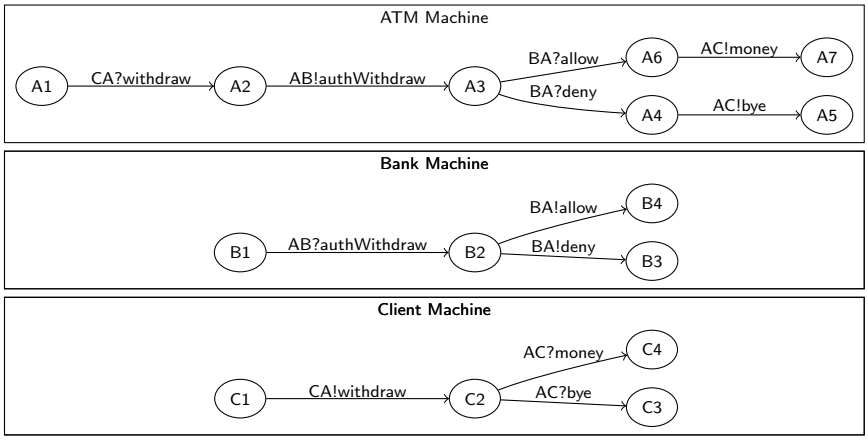


Figure 1: CFSMs for the protocol in Section 1

buffer b_{AB} keeps track of the messages sent from A to B. The *initial* configuration s_0 is the one where, for all $\mathsf{A} \in \mathscr{P}$, q_{A} is the initial state of the corresponding CFSM and all buffers are empty.

A configuration $s' = \langle \vec{q}' \,;\, \vec{b}' \rangle$ is *reachable* from another configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ by *firing an l-transition*, written $s \overset{l}{\Rightarrow} s'$, if there is a message $\mathtt{m} \in \mathscr{M}$ such that either (1) or (2) below holds:

1. $l = \mathsf{AB}!\mathtt{m}$, $\vec{q}(\mathsf{A}) \overset{l}{\rightarrow}_{\mathsf{A}} \vec{q}'(\mathsf{A})$, and
   a. $\vec{q}' = \vec{q}[\mathsf{A} \mapsto \vec{q}'(\mathsf{A})]$ and
   b. $\vec{b}' = \vec{b}[\mathsf{AB} \mapsto \vec{b}(\mathsf{AB}) \uplus \{\!|\, \mathtt{m} \,|\!\}]$
2. $l = \mathsf{AB}?\mathtt{m}$, $\vec{q}(\mathsf{B}) \overset{l}{\rightarrow}_{\mathsf{B}} \vec{q}'(\mathsf{B})$, $\vec{b}(\mathsf{AB})(\mathtt{m}) > 0$, and
   a. $\vec{q}' = \vec{q}[\mathsf{A} \mapsto \vec{q}'(\mathsf{A})]$ and
   b. $\vec{b}' = \vec{b}[\mathsf{AB} \mapsto \vec{b}(\mathsf{AB}) - \{\!|\, \mathtt{m} \,|\!\}]$

(where $f[x \mapsto y]$ is the usual update operation that redefines function f on an element x of its domain with y). Condition (1) puts $\mathtt{m}$ on channel AB, while (2) gets $\mathtt{m}$ from channel AB. In both cases, any machine or buffer not involved in the transition is left unchanged in the new configuration s'.

Note that this construction differs from the original definition in [6], (where unbounded FIFO queues were used) in order to make the communication model similar to the one of Erlang.

Example 4. *A local view of the protocol in Section 1 is given by the CFSMs in Fig. 1. Notice how the events reflected in the global view have been split into their* send *and* receive *counterparts.*

The starting state of each CFSM is the leftmost one. The CFSM of the client initiates the protocol by sending a withdraw message to the ATM, which reacts by sending a message to the bank to check whether the client can actually perform this withdrawal. CFSMs A *and* B *will gradually proceed as they take messages out from the queues existing between all pairs of participants.*

State B2 *of* B *is the* internal choice *state that corresponds to the branching point of the g-choreography. Namely, in* B2*, the bank locally chooses how to proceed. As soon as* B *sends either an* `allow` *or a* `deny` *message, the ATM either delivers the money or finishes the conversation with a* `bye` *message.*

A configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ is *stable* if all buffers are empty (note that stability does not impose any requirement on a machine's enabled transitions): s is *stable* for $\mathscr{C}' \subseteq \mathscr{C}$ if all buffers in $\mathscr{C}'$ are empty in s, and it is a *deadlock* if $s \not\Rightarrow$ and either there is a participant $\mathsf{A} \in \mathscr{P}$ such that $\vec{q}(\mathsf{A}) \xrightarrow{\mathsf{AB}?\mathtt{m}}_{\mathsf{A}}$ or s is not stable. This definition is adapted from [9] and is meant to capture communication misbehaviour. Observe that, according to this definition, a configuration s where all machines are in a state with no outgoing transitions and all buffers are empty is not a deadlock configuration even though $s \not\Rightarrow$.

Let $\Pi(S,s)$ be the set of *runs* of a communicating system S starting from a configuration s of S, that is the set of sequences $\pi = \{(\hat{s}_i, l_i, \hat{s}_{i+1})\}_{0\leq i\leq n}$ with $n \in \mathbb{N}\cup\{\infty\}$ such that $\hat{s}_0 = s$, and $\hat{s}_i \overset{l_i}{\Rightarrow} \hat{s}_{i+1}$ for every $0 \leq i \leq n$; we say that run π is *maximal* if $n = \infty$ or $\hat{s}_n \not\Rightarrow$ and denote with $\Pi(S)$ the runs of S starting from its initial state. The *language of a communicating system S* is the set

$$\mathscr{L}[S] = \bigcup_{\pi\in\Pi(S)} \{\text{trace of } \pi\}$$

where the *trace* of a run $\{(\hat{s}_i, l_i, \hat{s}_{i+1})\}_{0\leq i\leq n} \in \Pi(S)$ is the sequence $l_0 \dots l_{n-1}$. Notice that $\mathscr{L}[S] \subseteq \mathscr{L}^{\omega}_{\text{act}} \cup \mathscr{L}^{\star}_{\text{act}}$ (where $\mathscr{L}^{\omega}_{\text{act}}$ is the set of infinite words over $\mathscr{L}_{\text{act}}$) and it is prefix-closed.

3 Generating Tests

The goal of model-driven testing is to find mismatches between a specification and an implementation. We focus on *component-testing*, which in our setting corresponds to test a single participant of a g-choreography. We dub *component under test* (CUT) an implementation which should be tested.

3.1 Baseline concepts

Top-down approaches of choreographies define projection functions that generate local models from global models. In order to parameterise our framework with respect to these notions, we introduce *abstract projections* on g-choreographies.

Definition 1 (Abstract projection)**.** *A map* $\_\!\upharpoonright_{\_}$ *is an* abstract projection *if it takes a g-choreography* G *and a participant* $\mathsf{A} \in \mathscr{P}$ *and returns an* A*-local CFSM. Given a g-choreography* G*, the* system induced by $\_\!\upharpoonright_{\_}$ *is defined as* $\mathsf{G}\!\upharpoonright = (\mathsf{G}\!\upharpoonright_{\mathsf{A}})_{\mathsf{A}\in\mathscr{P}}$.

There are several ways to define projection operations that are instances of Definition 1. For example, in [14, 26] a g-choreography is projected on a participant A in two steps, which we briefly summarise since we will illustrate our framework by adopting this operation in our examples. By induction on the structure of the g-choreography, the first step transforms each interaction in the transition of an automaton according to the role of A in that interaction. More precisely, the interaction becomes an output or an input transition depending on whether A is the sender or the receiver; otherwise the iteration corresponds to a silent transition. In the second step, the CFSM obtained as above is determinised.

Example 5. *The CFSMs shown in Fig. 1 are obtained by means of the projection operation in [14, 26] applied to the g-choreography* G_{ATM} *in Example 1 where some equivalent states (e.g.,* $\mathsf{A5}$ *and* $\mathsf{A7}$*) are replicated for readability.*

Not every g-choreography can be *faithfully* projected. In fact, the asynchronous semantics of communicating systems may introduce behaviour that does not correspond to the intended behaviour of the g-choreography. In concrete instances, sufficient conditions on g-choreographies are given so that the semantics of projected communicating systems reflect the semantics of the g-choreography. These conditions are abstractly captured in the next definition.

Definition 2 (Abstract well-formedness)**.** *A predicate on g-choreographies is an* abstract well-formedness condition *if* $\mathit{WF}(\mathsf{G})$ *implies that there is a communicating system S with initial configuration* s_0 *such that*

- $\mathscr{L}[S] \subseteq \mathscr{L}[\mathsf{G}]$ *and*
- *no run in* $\Pi(s_0)$ *contains a deadlock configuration;*

in this case we say that S realises G.

Note that Definition 2 admits trivial instances such as the predicate which does not hold on any g-choreography. The choreography in Section 1 is considered well-formed in the majority of existing work. In this example there is only one participant that makes a choice (i.e., the bank) and the rest of the participants are informed of which decision was taken. Intuitively, this avoids coordination problems and therefore the choreography can be correctly realized by CFSMs, such as the ones in Fig. 1. In this case, the language of the choreography is the same as the language of the projected communicating system, which is deadlock-free.

Hereafter, we assume projections that *respect* abstract well-formedness.

Definition 3 (Compatible projections)**.** *An abstract projection $\_\downarrow\_$ is* compatible with *WF when, for all g-choreographies G, if WF(G) then the system induced by $\_\downarrow\_$ realises G.*

An abstract projection mapping all participants to a machine without any transitions is trivially compatible with any abstract well-formedness condition. Of course, we are interested in abstract projections for which $\mathscr{L}[\mathsf{G}\downarrow] = \emptyset$ only if $\mathsf{G} = (\mathsf{o})$.

We can now formalise the main notions of our choreographic testing framework. A *test case* for a CUT A is a set of CFSMs with a distinguished set of success states; the outcome of a test case is determined by its interaction with A.

Definition 4 (Test case)**.** *A* test case for a CUT $\mathsf{A} \in \mathscr{P}$ *is a set $T = \{\langle M_1, \underline{Q}_1\rangle, \ldots, \langle M_n, \underline{Q}_n\rangle\}$ such that for every $1 \leq i, j \leq n$, $M_i = (Q_i, q_{0i}, \rightarrow_i)$ is a CFSM with $\underline{Q}_i \subseteq Q_i$ and*

- *if $q \xrightarrow{l}_i q'$ then $\mathsf{sbj}(l) \neq \mathsf{A}$* *(1)*
- *if $q \xrightarrow{\mathsf{B\,C!m}}_i q'$ and $q \xrightarrow{l}_i q''$ then $l = \mathsf{B\,C!m}$* *(2)*
- *if $q \xrightarrow{l}_i q'$ and $q \xrightarrow{l}_i q''$ then $q' = q''$* *(3)*
- *if $q_1 \xrightarrow{l}_i q_2$ and $q_1' \xrightarrow{l'}_j q_2'$ and $\mathsf{sbj}(l) = \mathsf{sbj}(l')$ then $i = j$* *(4)*

We dub $\underline{Q}_i$ the success states *of M_i.*

We briefly justify the conditions in Definition 4. Condition (1) forces the CUT not to be the subject of any transition, since tests cannot force it directly to take specific actions. Conditions (2) and (3) together enforce that there is always a single possible output for the system to proceed, that the machines are deterministic and, in particular, that they cannot have internal choice or mixed[3] states. The rationale behing conditions (2) and (3) is to "confine" non-determinism in the CUT and its concurrent execution with the test so that it is easier to analyse the outcome of tests. The last condition enforces transitions across machines to have different subjects: if this was not the case, generating code for each participant could be significantly more complex. Note that this does not force the CFSMs in a test case to be necessarily local; in fact, Definition 4 admits different subjects in the labels of different transitions.

The following example shows the requirements of Definition 4 and a violation of those requirements.

Example 6. *Consider M_{A}, M_{B} and M_{C} in Fig. 1 that respectively are the CFSMs of the ATM, the bank, and the client. Then $T_1 = \{\langle M_{\mathsf{A}}, \{\mathsf{A5}, \mathsf{A7}\}\rangle, \langle M_{\mathsf{C}}, \{\mathsf{C3}, \mathsf{C4}\}\rangle\}$ is a test case for B (i.e., bank). In fact, M_{A} and M_{C} are deterministic, internal choice-free and do not include any transitions where the subject is B. Instead, $T_2 = \{\langle M_{\mathsf{B}}, \{\mathsf{B3}, \mathsf{B4}\}\rangle, \langle M_{\mathsf{C}}, \{\mathsf{C3}, \mathsf{C4}\}\rangle\}$ is not a test case for A (i.e., the ATM) because M_{B} has an internal choice in state $\mathsf{B2}$.*

[3] A mixed choice state is one with both input and output outgoing transitions.

Definition 5 (Test compliance). *Let $\mathscr{C}' \subseteq \mathscr{C}$ be a set of channels, $\hat{M}$ a CFSM, and T a test case. Denote with $\hat{M} \otimes T$ the communicating system consisting of $\hat{M}$ and the CFSMs in T. We say that $\hat{M}$ is T-*compliant w.r.t *$\mathscr{C}'$ ($\hat{M} \triangleright_{\mathscr{C}'} T$) if every finite maximal run of $\hat{M} \otimes T$ contains a stable configuration s for $\mathscr{C}'$ such that for every $\langle M, \underline{Q} \rangle \in T$ the local state of M in s is in $\underline{Q}$.*

In the following, we dub the configuration s in Definition 5 a *successful configuration for* T and we use $\hat{M} \triangleright T$ for $\hat{M} \triangleright_{\mathscr{C}} T$. Notice that the parametrization on $\mathscr{C}'$ allows a CFSM to be considered compliant even if some runs leave channels in $\mathscr{C} \setminus \mathscr{C}'$ not empty. The next series of examples illustrate the notion of test compliance with four tests for CUTs in Fig. 1.

Example 7. *Let* B *be the CUT and T_1 be the test case in Example 6. Then M_B is T_1-compliant. In fact, the system consisting of M_B and (the CFSMs in) T_1 is exactly the system implementing the choreography of the running example. However, M_B is not compliant with the test case $\{\langle M_\mathsf{A}, \{\mathsf{A3}\}\rangle, \langle M_\mathsf{C}, \{\mathsf{C3}, \mathsf{C4}\}\rangle\}$. In fact,* C *can reach* C3 *or* C4 *only after that* A *has left state* A3*. Similarly, M_B is not compliant with the test case $\{\langle M_\mathsf{A}, \{\mathsf{A7}\}\rangle, \langle M_\mathsf{C}, \{\mathsf{C3}\}\rangle\}$, since the success states of* A *and* C *represent conflicting branches.*

Example 8. *Suppose that the CUT is the CFSM M'_B obtained by removing the transition* B A!`allow` *from M_B. Then $M'_\mathsf{B} \triangleright T_1$ however, M'_B is not compliant with $\{\langle M_\mathsf{A}, \{\mathsf{A7}\}\rangle, \langle M_\mathsf{C}, \{\mathsf{C4}\}\rangle\}$. This is due to the fact that the test and the CUT select different branches. Similarly, M'_B is not compliant with $\{\langle M'_\mathsf{A}, \{\mathsf{A7}\}\rangle, \langle M_\mathsf{C}, \{\mathsf{C3}, \mathsf{C4}\}\rangle\}$, where M'_A is obtained by removing the transition* B A?`deny` *from M_A.*

Example 9. *Finally, let* A *be the CUT and M'_B be the CFSM obtained by removing the transition* B A!`allow` *from M_B. Then M_A is compliant with $\{\langle M'_\mathsf{B}, \{\mathsf{B3}\}\rangle, \langle M_\mathsf{C}, \{\mathsf{C3}, \mathsf{C4}\}\rangle\}$.*

We finally define when a test case is meaningful for a choreography, by requiring that the correct implementation (i.e., the projection) of the choreography is compliant with the test.

Definition 6 (Test suitability). *Test T is $(\mathsf{G}, \mathsf{A})-$suitable if $\mathsf{G}\!\downarrow_\mathsf{A} \triangleright T$.*

3.2 Test generation algorithm

To generate tests we follow a straightforward strategy: we start from the projections of the participants that are not the CUT and we remove their internal choices. The intuition is that for well formed g-choreographies, the projections are "compatible" with any implementation that restricts internal choices with respect to the projection of the CUT. We use the following auxiliary function to identify non-deterministic states. These are the states that the algorithm uses to split the transitions to obtain deterministic tests. Given a CFSM $M = (Q, q_0, \rightarrow)$, let

$$nds(M) = \left\{ q \;\middle|\; \exists q \xrightarrow{l_1} q_1 \neq q \xrightarrow{l_2} q_2 \,:\, l_1 = l_2 \vee \{l_1, l_2\} \cap \in \mathscr{L}^!_{\text{act}} \neq \emptyset) \right\}$$

be the set of non-deterministic states of M, that is the states with at least two different transitions that either have the same label or one of which is an output transition. For convenience, we let $M(q)$ denote the set of outgoing transitions of q in M and $M - t$ (resp. $M + t$) be the operation that removes from (resp. adds to) M transition t (these operations extend element-wise to sets of transitions). The following

function produces sets of machines that are internal choice free:

$$\mathsf{split}(M) = \begin{cases} \{M\} & \text{if } \mathit{nds}(M) = \emptyset \\ \bigcup_{q \in \mathit{nds}(M)} \mathsf{split}(M, q) & \text{otherwise} \end{cases}$$

$$\mathsf{split}(M, q) = \begin{cases} \bigcup\limits_{q \xrightarrow{\mathsf{AB!m}} q'} \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{AB!m}} q') & \text{if } M(q) \text{ has output transitions} \\ \bigcup\limits_{\substack{q \xrightarrow{\mathsf{AB?m}} q' \\ \neq \\ q \xrightarrow{\mathsf{AB?m}} q''}} \mathsf{split}(M - q \xrightarrow{\mathsf{AB?m}} q') & \text{otherwise} \end{cases}$$

Once these simpler CFSMs are obtained, success states have to be set for each of them. This is analogous to problem commonly known in software testing as the *oracle problem*: deciding when a test is successful. This decision is application-dependent and its solutions usually requires human intervention [2]. In our setting, this corresponds to single out configurations of communicating systems according to a sub-tree of a choreography as defined below. Intuitively, we would like success states from the CFSMs to correspond to the execution of specific syntactic subtrees of the choreography.

We now introduce an additional definition that helps us determine the success states for our tests. In the following, given a g-choreography G, let $\mathbb{T}(\mathsf{G})$ be the set of sub-trees of the abstract syntax tree producing G once we fix a suitable precedence among the operators. Our algorithm relies on abstract syntax trees of g-choreographies, but it does not depend on the precedence relation chosen.

Definition 7 (Oracle scheme)**.** *Let* G *be a g-choreography,* $\_\!\upharpoonright_{\_}$ *an abstract projection compatible with a given well-formedness condition WF. An* oracle scheme of G *for* $\_\!\upharpoonright_{\_}$ *is a function* $\Omega_{\mathsf{G},\upharpoonright}$ *mapping a pair* $(\mathsf{A}, \tau) \in \mathscr{P} \times \mathbb{T}(\mathsf{G})$ *on a set of states of the CFSM* $\mathsf{G}\upharpoonright_{\mathsf{A}}$ *such that if* $\mathit{WF}(\mathsf{G})$ *and* S *is the communicating system induced by* $\_\!\upharpoonright_{\_}$*, then for every* $\tau \in \mathbb{T}(\mathsf{G})$ *and maximal run* $\pi \in \Pi(S)$ *there exists a stable configuration* s *in* π *such that, for each* $\mathsf{A} \in \mathscr{P}$*, for the local state* q_{A} *of* A *in* s *we have that* $q_{\mathsf{A}} \in \Omega_{\mathsf{G},\upharpoonright}(\mathsf{A}, \tau)$*.*

The main purpose of the oracle scheme Ω_{G} is to map a participant and a subtree $(\mathsf{A}, \tau) \in \mathscr{P} \times \mathbb{T}(\mathsf{G})$ to a set of states of $\mathsf{G}\upharpoonright_{\mathsf{A}}$ that correspond to the states the system can be in after the execution of the sub-tree τ of G.

Example 10. *Below is a fragment of a possible oracle scheme for the g-choreography from Example 1 and the CFSMs shown in Fig. 1.*

$$\Omega_{\mathsf{G},\upharpoonright}(\mathsf{A}, \mathsf{G}) = \{\mathsf{A5}, \mathsf{A7}\} \quad \textit{and} \quad \Omega_{\mathsf{G},\upharpoonright}(\mathsf{B}, \mathsf{G}) = \{\mathsf{B3}, \mathsf{B4}\} \quad \textit{and} \quad \Omega_{\mathsf{G},\upharpoonright}(\mathsf{C}, \mathsf{G}) = \{\mathsf{C3}, \mathsf{C4}\}$$

$$\Omega_{\mathsf{G},\upharpoonright}(\mathsf{A}, \mathsf{B} \xrightarrow{\mathtt{allow}} \mathsf{A}) = \{\mathsf{A6}, \mathsf{A4}\} \textit{ and } \Omega_{\mathsf{G},\upharpoonright}(\mathsf{B}, \mathsf{B} \xrightarrow{\mathtt{allow}} \mathsf{A}) = \{\mathsf{B3}, \mathsf{B4}\} \textit{ and } \Omega_{\mathsf{G},\upharpoonright}(\mathsf{C}, \mathsf{B} \xrightarrow{\mathtt{allow}} \mathsf{A}) = \{\mathsf{C3}, \mathsf{C4}\}$$

Notice that for the whole g-choreography G*, the oracle scheme* $\Omega_{\mathsf{G},\upharpoonright}$ *yields the last states of the CFSMs, and for the sub-tree* $\mathsf{B} \xrightarrow{\mathtt{allow}} \mathsf{A}$ *it returns the first state that allows the participant to acknowledge either the execution of the interaction or the selection of an alternative branch.*

Test cases are then built by combining machines obtained by the split function and by identifying the success states via the oracle function, i.e. states that correspond to the execution of the interactions of the subtrees of the g-choreography:

$$\mathit{tests}(\mathsf{G}, \mathsf{A}) = \left\{ (\langle M_{\mathsf{B}}, \Omega_{\mathsf{G},\upharpoonright}(\mathsf{B}, \tau) \rangle)_{\mathsf{B} \neq \mathsf{A} \in \mathscr{P}} \;\middle|\; \forall \mathsf{B} \neq \mathsf{A} \in \mathscr{P} \,:\, M_{\mathsf{B}} \in \mathsf{split}(\mathsf{G}\upharpoonright_{\mathsf{B}}) \wedge \tau \in \mathbb{T}(G) \right\} \quad (1)$$

More intuitively, for every participant we select a single machine from the ones generated by split and combine them (exhaustively) into test cases. Each test case corresponds to a unique path of execution (i.e. selection of internal choices) of the original g-choreography.

Theorem 1. *If* $WF(\mathsf{G})$ *then every test case in* $tests(\mathsf{G},\mathsf{A})$ *is* (G,A) *– suitable.*

4 Choregraphy-based Testing

We now delve into a larger example in order to demonstrate the test generation procedure in a more complex scenario. Fig. 2 shows a choreography involving the participants A B, and C, i.e., respectively the ATM, the bank, and a client as in the running example used so far. Observe that the running example

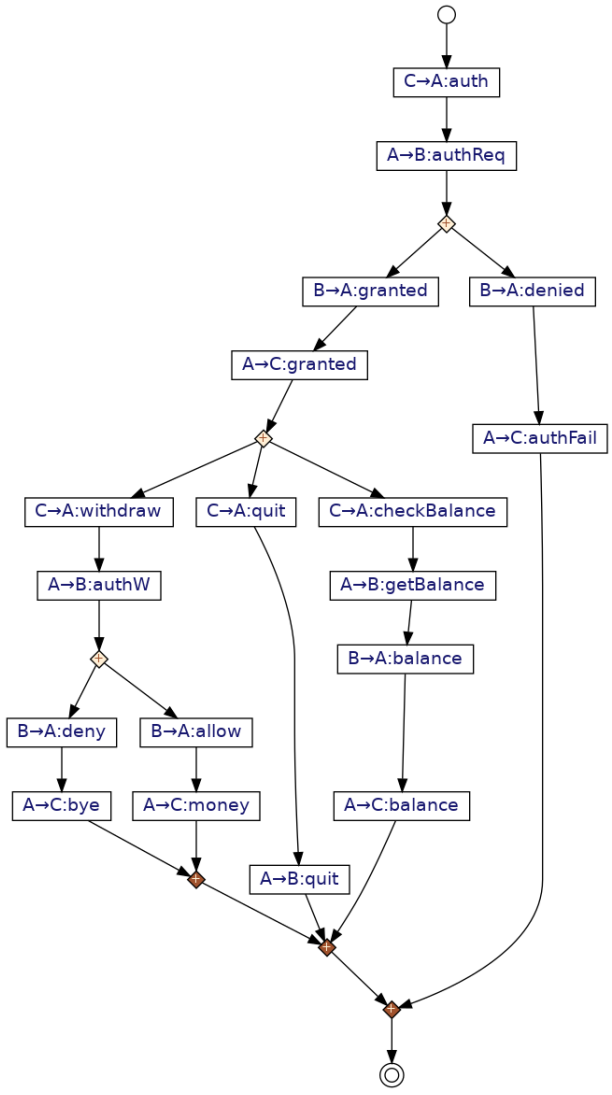


Figure 2: The complete choreography for the ATM scenario

is a sub-choreography of the g-choreography in Fig. 2. The bigger scenario can be straightforwardly represented as a g-choreography as done in Example 1 for the choreography in Section 1.

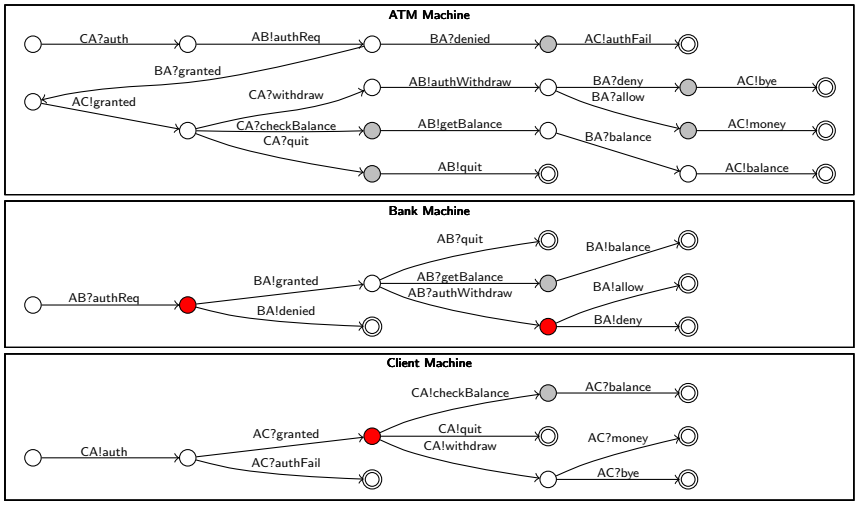


Figure 3: Projections of the choreography of Fig. 2

The client starts a session of the protocol by authenticating with the ATM machine (`auth`). The ATM then delegates the authentication to the bank, which can either reject or accept the request by replying with either a `denied` or a `granted` message. In both cases the ATM forwards the authentication result to the client. The choreography terminates if the authentication fails. If authentication is successful then the ATM offers three options to the client: (M) withdraw money (`money`), (Q) terminate the session (`quit`), or (B) check the account balance `checkBalance`.

In case (B), the ATM requests to the bank the balance and forwards the result to the client via a `balance` message. In case (Q), the ATM simply notifies the bank of the termination of the session. Case (M) is the choreography of Section 1 whereby the withdrawal request is forwarded to the bank which decides if to allow or deny the request.

We demonstrate the test case generation for the ATM (i.e., participant A in Fig. 2 is our CUT). We first project the g-choreography corresponding to the choreography in Fig. 2, using again the projection operation in [26]. We obtain the three CFSMs of Fig. 3. The oracle scheme is visually represented by decorating states only for two sub-trees of the choreography. More precisely:

- double-circles denote the states marked by the oracle scheme for the whole choreography, and
- gray-circles correspond to the states marked by the oracle scheme for the interaction $\mathsf{B} \xrightarrow{\texttt{allow}} \mathsf{A}$.

It is straightforward to check that the system consisting of these three CFSMs effectively generates the same language as the one generated by the g-choreography.

The two machines for B and C of Fig. 3 cannot be directly used as a test for A since they have states with internal choices. These states, obtained by applying *nds* to the CFSMs of B and C, are the sets of red states shown in the figure. At this point the algorithm applies $\mathsf{split}(\mathsf{B})$ to compute a set of four machines, say $\mathbb{M}_\mathsf{B}$. This is done by selecting in all possible ways one of the output transitions from states of B (according to the second clause in the definition of split). Likewise for C, the algorithm produces a set of three machines, say $\mathbb{M}_\mathsf{C}$. The resulting sets of CFSMs are shown in Figs. 4 and 5 where, for the sake of conciseness, we remove unreachable states, also omitting isomorphic CFSMs. For a sub-tree of the choreography, we obtain a test case by combining a machine from $\mathbb{M}_\mathsf{B}$ and one from $\mathbb{M}_\mathsf{C}$ and defining their success states using the oracle scheme. Function $\mathit{tests}(\mathsf{G},\mathsf{A})$ generates all the test cases by freely choosing the machines as above and exhaustively iterating over the sub-trees of the choreography.

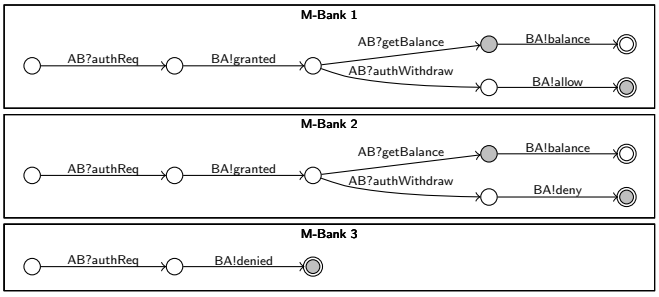


Figure 4: CFSMs resulting from splitting the projected CFSMs for the bank

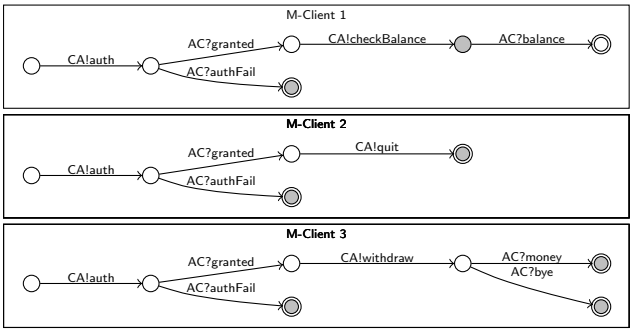


Figure 5: CFSMs resulting from splitting the projected CFSMs for the client

This process results in nine tests for each sub-tree of the g-choreography. For the tree corresponding to the whole g-choreography, the success states are those depicted as double-circles. For the tree corresponding to the interaction $\mathsf{B} \xrightarrow{\texttt{allow}} \mathsf{A}$, the success states are those in gray. Notice that some states are success states for both trees. Moreover, all the resulting tests satisfy the requirements of Definition 4.

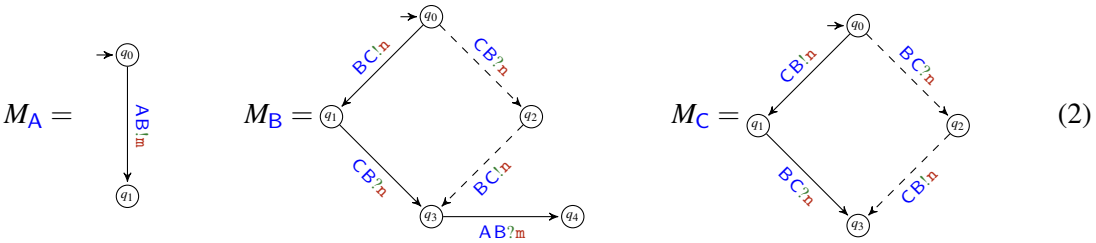
5 Discussion & Open Problems

We started the exploration of mechanisms to support model-driven testing of message-passing systems based on choreographies. To this purpose, we decided to rely on the so called top-down approach featured by an existing choreographic model. The choreographic model adopted here is rather abstract, but it is close to real programming paradigms such as those of Erlang.

We exploited the notion of projection of global views of choreographies in order to devise an automatic test generation mechanism. The design of our algorithm required us to fix the basic notion of test, test feasibility, and test success within the framework of g-choreographies and communicating systems. Although we tried to give a general framework that abstracts away from actual projection operations, we took some design decisions for the identification of our framework.

The notion of test case considered here (Definition 4) requires tests not to contain mixed-choice states (that is, states with both output and input outgoing transitions). In fact, without assumptions on the projection operation mixed-choice states cannot be split easily as they are. Consider the system

consisting of following CFSMs:



(2)

where M_A is the CUT. The split of the mixed choices of M_B and M_C is unsafe, because the test including the dashed transitions has a run to a deadlock configuration despite the fact that M_A behaves as expected. Note that with insights on the actual notion of well-formedness and of the projection operation one can deal with mixed choices. For instance, the well-formedness condition and the projection operation in [26] yields mixed choice states only when projecting parallel g-choreographies. Therefore, it is safe in a mixed-choice state, say q, to select a test starting with one of the output transitions of q and drop all the others. Note that this yields "simpler" tests, in line with the principles of software testing.

Another limitation of the algorithm is its efficiency. As noted in Section 3, our algorithm is exponential in the size of the g-choreography. This is due to the fact that the oracle specification used in the algorithm exhaustively considers all the syntactic sub-trees of the g-choreographies. This could be unfeasible for large g-choreographies. Note however that the oracle specification is a parameter of our algorithm and, in practice, one can tune it up in order to consider only "interesting" parts of the g-choreography to target. Moreover, some optimisations are possible. A first optimisation can be the reduction of internal choices generated by the parallel composition as those for G_{par} above. In fact, those tests are redundant and one would be enough in the semantics of communicating systems adopted here (where channels are multisets of messages similar to Erlang's mailboxes). Note that the tests would not be redundant in the case of communicating systems interacting through FIFO queues. Another optimisation relies on the analysis of the syntactic structure to exclude immaterial sub-trees. For instance, for the g-choreography $\mathsf{G};\mathsf{A}\xrightarrow{\mathtt{m}}\mathsf{B};\mathsf{X}\xrightarrow{\mathtt{n}}\mathsf{Y};\mathsf{G}'$ it is not necessary to check $\mathsf{A}\xrightarrow{\mathtt{m}}\mathsf{B};\mathsf{X}\xrightarrow{\mathtt{n}}\mathsf{Y};\mathsf{G}'$ because the sub-tree $\mathsf{X}\xrightarrow{\mathtt{n}}\mathsf{Y};\mathsf{G}'$ subsumes the runs that "go through" the former tree. A pre-processing of the oracle specification may therefore improve efficiency. Note that adopting this approach probably requires a careful transformation of the oracle specification. This may not be easy to attain. Another optimisation comes from the study of some notion of "dominance" of tests. The discussion above about mixed-choices is an example: in a mixed-choice state, the tests with a bias on first-outputs dominate those starting with inputs. For instance, the test with solid transitions in (2) above dominate the one with dashed transitions.

This leads us to consider some other related open questions. In software testing it is widely accepted that it is unfeasible to perform a high number of tests. Hence, test suites are formed by carefully selected tests that satisfy some *coverage* criteria. This yields a number of questions that we did not address yet: What is a good notion of coverage for communicating systems? Can choreographic models help in identifying good coverage measures? What heuristics lead to good coverage? Remarkably, this problem pairs off with the problem of *concretisation* in model-driven testing [23]. Given an abstract test (as the ones we generate), how should it be concretised to test actual implementations? In fact, the abstract notion of coverage only considers distributed choices, but actual implementations may have local branching computations that should also be covered to some extent. This probably requires our approach to be combined with existing approaches to testing.

As said, we took some design decisions to devise our framework. Alternative approaches are possible. Firstly, test generation may be done differently when adopting different types of tests. In fact, a

natural alternative is to take the projection of one component as the CUT, say M, and consider as test cases the CFSM obtained by dualising M. Note that this yields a non-local CFSM as a test case; we preferred to explore first an approach which yields "standard" communicating systems.

Definition 6 formalizes when a test case is meaningful for a choreography. It would be also desirable to relate traces of machines that are test-compliant with the language of the choreography. Ideally, for a choreography G an *adherent* test T should guarantee that for every T*-compliant* machine M the traces of runs of $M \otimes T$ that end in a successful configuration are in $\mathscr{L}[\mathsf{G}]$. This property cannot be guaranteed by our framework for arbitrary choreographies. Firstly, the CUT may force causal relations. For example, consider $\mathsf{A} \xrightarrow{\mathtt{x}} \mathsf{B}; \mathsf{B} \xrightarrow{\mathtt{y}} \mathsf{A}; \mathsf{A} \xrightarrow{\mathtt{z}} \mathsf{C}$ where A is the CUT. The event $\mathsf{B}\,\mathsf{A}!\mathtt{y}$ should always precede $\mathsf{A}\,\mathsf{C}!\mathtt{z}$. However, this dependency is enforced by A and cannot be checked by B and C without communication between them. Secondly, in an asynchronous setting it may be impossible to distinguish some behaviors of the CUT. For example, in $\mathsf{A} \xrightarrow{\mathtt{x}} \mathsf{B}; \mathsf{A} \xrightarrow{\mathtt{y}} \mathsf{B}$ the event $\mathsf{A}\,\mathsf{B}!\mathtt{x}$ should always precede $\mathsf{A}\,\mathsf{B}!\mathtt{y}$, but this order is not observable by B in case of asynchronous communication.

In summary, the notion of adherence is not enforceable for all g-choreographies or all possible implementations of the CUT. This hints to the following open problems: the identification of a proper notion of adherence in an asynchronous setting, the identification of "interesting" subclasses of g-choreographies for which the strict notion of adherence is meaningful, and the extension of the testing framework to enforce such notion, either by adding communications between components or by using non-local machines.

In this work, we consider component testing. The level of granularity we adopt implies that participants are components, and our framework is designed to test a single component at a time. An intriguing open problem is to apply our framework to support integration testing [24]. In fact, one could think of defining *group* projections, namely projection operations that generate communicating systems representing the composition of several participants. We believe that this approach could pay off when the group onto which the g-choreography is projected can be partitioned in a set of "shy" participants that interact only with participants within the group and others that also interact outside the group. The former set of participants basically corresponds to units that are stable parts of the system that and do not need to be (re-)tested as long as the components in the other group pass some tests.

Instead of concretising abstract tests, one could extract CFSMs from actual implementations and run the tests on them. Machines could potentially be extracted directly from source code. If however source code was not available it could still be possible to test components (e.g., by using some machine learning algorithm to infer the CFSMs from data such as traces). Note that such technique should be more efficient than concretisation (because it does not let abstract tests proliferate into many concrete ones). Moreover, another advantage of this approach could be that it enables us to exploit the bottom-up approach of choreographies, where global views are synthesised from local ones [20]. The synthesised choreography can be compared with a reference one to derive tests that are more specific to the implementation at hand.

6 Conclusions & Related Work

In software engineering, testing is considered *the* tool[4] for validating software and assuring its quality. The *Software Engineering Book of Knowledge* available from `http://www.swebok.org` describes *software testing* as (bold text is ours):

[4] Regrettably, barred for few exceptions, rigorous formal methods that aim to show absence of defects rather than their presence are less spread in current practices. We cannot embark in a discussion on this state of the matter here.

> "the dynamic **verification** of the behaviour of a program on a **finite** set of **test cases**, **suitably selected** from the usually **infinite** executions domain, **against the expected behavior**."

Our framework reflects the description above for model-driven testing of message-passing systems. Traditional testing has been classified [25] according to parameters such as the scale of the system under test, the source from which tests are derived (e.g., requirements, models, or code). There are also classifications according to the specific characteristics being checked [22]; our work can be assigned to the category of behavioural testing.

An immediate goal of ours is to experimentally check the suitability of the test cases obtained with our algorithm. For this, we plan to identify suitable concretisation mechanisms of the abstract tests generated by our algorithm, and verify Erlang or Golang programs.

Since message-passing systems fall under the class of *reactive systems* we got inspiration from the work done on model-driven testing of reactive systems [8]. In particular, we showed that choreographies can, at least to some extent, be used to automatically generate executable tests and as test case specifications [23]. Technically, we exploited the so-called *projection* operation of choreographic models. Here, we gave an abstract notion of projection. A concrete projection was formalised for the first time in [17] (for multiparty session types) and for g-choreographies in [14, 15, 26], elaborating on the projection of global graphs [13]. As discussed in Section 5, in the future we will also explore the use of choreographic model-driven testing to address other problems related to testing message-passing systems.

An interesting theoretical investigation would be to explore the relation between our approach and the theory of testing [12]. At a first glance, our approach corresponds to the must-preorder of the testing theory. In fact, the notion of test compliance (cf. Definition 5) imposes conditions on all the maximal runs of the CUT in parallel with the test. However, there are two key differences between the theory of testing and our approach which make a precise analysis non trivial. The first difference is that we consider asynchronous communications and the second is that our tests are "multiparty", namely tests are obtained by composing many CFSMs. It might be that the results in [5], which extend to asynchronous communications the classical theory of testing, can be combined with the work in [21] to give a suitable theoretical setting to our framework.

According to [27], the generation of test cases is one of the ways model-based testing can support software verification. For example, a component-based testing framework to support online testing of choreographed services is proposed in [1] for BPMN2 models. Among other components, this work sketches a test generation procedure which is however not supported by a formal semantics as we do here. Our model explicitly features a mechanism for test generation paired with the notion of an *oracle scheme* (cf. Definition 7) as a precise mechanism to identify the expected outcome of test cases. In fact, unlike in most cases, choreographic models contain enough information about the expected behaviour of the system under test in order to make accurate predictions. We believe that this is a highlight of our approach.

References

[1] Midhat Ali, Francesco De Angelis, Daniele Fanì, Antonia Bertolino, Guglielmo De Angelis & Andrea Polini (2014): *An Extensible Framework for Online Testing of Choreographed Services*. *IEEE Computer* 47(2), pp. 23–29, doi:10.1109/MC.2013.407.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz & Shin Yoo (2015): *The Oracle Problem in Software Testing: A Survey*. *TOSEM* 41(5), pp. 507–525, doi:10.1109/TSE.2014.2372785.

[3] Laura Bocchi, Julien Lange & Emilio Tuosto (2011): *Amending Contracts for Choreographies*. In: *ICE*, pp. 111–129, doi:10.4204/EPTCS.59.10.

[4] Laura Bocchi, Julien Lange & Emilio Tuosto (2012): *Three Algorithms and a Methodology for Amending Contracts for Choreographies*. *Sci. Ann. Comp. Sci.* 22(1), pp. 61–104, doi:10.7561/SACS.2012.1.61.

[5] Michele Boreale, Rocco De Nicola & Rosario Pugliese (2002): *Trace and testing equivalence on asynchronous processes*. *Information and Computation* 172(2), pp. 139–164, doi:10.1006/inco.2001.3080.

[6] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *JACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[7] Mario Bravetti & Gianluigi Zavattaro (2009): *Contract Compliance and Choreography Conformance in the Presence of Message Queues*. In: *Web Services and Formal Methods*, 5387, Springer, Berlin, Heidelberg, pp. 37–54, doi:10.1007/978-3-642-01364-5˙3.

[8] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker & Alexander Pretschner, editors (2005): *Model-Based Testing of Reactive Systems, Advanced Lectures*. *LNCS* 3472, Springer, doi:10.1007/b137241.

[9] Gérard Cécé & Alain Finkel (2005): *Verification of programs with half-duplex communication*. *I&C* 202(2), pp. 166–190, doi:10.1016/j.ic.2005.05.006.

[10] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida & Luca Padovani (2016): *Global progress for dynamically interleaved multiparty sessions*. *MSCS* 26(2), pp. 238–302, doi:10.1017/S0960129514000188.

[11] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese & Mauro Jacopo (2015): *Dynamic Choreographies - Safe Runtime Updates of Distributed Applications*. In: *COORDINATION*, pp. 67–82, doi:10.1007/978-3-319-19282-6_5.

[12] Rocco De Nicola & Matthew C. B. Hennessy (1984): *Testing equivalences for processes*. *TCS* 34, pp. 83–133, doi:10.1016/0304-3975(84)90113-0.

[13] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In: *ESOP*, LNCS, Springer, pp. 194–213, doi:10.1007/978-3-642-28869-2˙10.

[14] Roberto Guanciale & Emilio Tuosto (2016): *An Abstract Semantics of the Global View of Choreographies*. In: *Interaction and Concurrency Experience*, pp. 67–82, doi:10.4204/EPTCS.223.5.

[15] Roberto Guanciale & Emilio Tuosto (2018): *Semantics of Global Views of Choreographies*. *Journal of Logic and Algebraic Methods in Programming* 95, pp. 17–40, doi:10.1016/j.jlamp.2017.11.002.

[16] Roberto Guanciale & Emilio Tuosto (2019): *Realisability of pomsets*. *JLAMP* 108, pp. 69–89, doi:10.1016/j.jlamp.2019.06.003.

[17] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *JACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695. Extended version of a paper presented at POPL08.

[18] Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher & Yves Lafon: `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217`. Working Draft 17 December 2004.

[19] Ivan Lanese, Fabrizio Montesi & Gianluigi Zavattaro (2013): *Amending Choreographies*. In: *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013.*, pp. 34–48, doi:10.4204/EPTCS.123.5.

[20] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 221–232, doi:10.1145/2676726.2676964.

[21] Rocco De Nicola & Hernán C. Melgratti (2015): *Multiparty Testing Preorders*. In: *Trustworthy Global Computing*, pp. 16–31, doi:10.1007/978-3-319-28766-9_2.

[22] William L. Oberkampf & Christopher J. Roy (2010): *Verification and Validation in Scientific Computing*. Cambridge University Press, doi:10.1017/cbo9780511760396.

[23] Alexander Pretschner & Jan Philipps (2005): *Methodological Issues in Model-Based Testing*. In: *Model-Based Testing of Reactive Systems*, 3472, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 281–291, doi:10.1007/11498490˙13.

[24] Muhammad Jaffar-ur Rehman, Fakhra Jabeen, Antonia Bertolino & Andrea Polini (2007): *Testing software components for integration: a survey of issues and techniques*. *Software Testing, Verification and Reliability* 17(2), pp. 95–133, doi:10.1002/stvr.357.

[25] Jan Tretmans (1999): *Testing Concurrent Systems: A Formal Approach*. In: *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, *Lecture Notes in Computer Science* 1664, Springer, pp. 46–65, doi:10.1007/3-540-48320-9_6.

[26] Emilio Tuosto & Roberto Guanciale (2018): *Semantics of global view of choreographies*. *Journal of Logical and Algebraic Methods in Programming* 95, pp. 17–40, doi:10.1016/j.jlamp.2017.11.002.

[27] Mark Utting & Bruno Legeard (2007): *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann. Available at `http://www.elsevierdirect.com/product.jsp?isbn=9780123725011`.