Choreographic Development of Message-Passing Applications
A Tutorial

Alex Coto[1(✉)], Roberto Guanciale[2], and Emilio Tuosto[1]

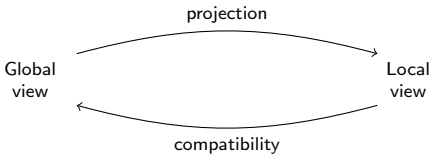
[1] Gran Sasso Science Institute, L'Aquila, Italy
{alex.coto,emilio.tuosto}@gssi.it
[2] KTH, Stockholm, Sweden
robertog@kth.se

Abstract. Choreographic development envisages distributed coordination as determined by interactions that allow peer components to harmoniously realise a given task. Unlike in orchestration-based coordination, there is no special component directing the execution. Recently, choreographic approaches have become popular in industrial contexts where reliability and scalability are crucial factors. This tutorial reviews some recent ideas to harness choreographic development of message-passing software. The key features of the approach are showcased within ChorGram, a toolchain which allows software architects to identify defects of message-passing applications at early stages of development.

1 Introduction

Choreographic approaches advocate model-driven engineering (MDE) based on two different *views* of distributed applications. A **global view** specifies the interactions among the various distributed components (aka *participants*) while a **local view** models each component of the system.

This tutorial illustrates how global and local views enable both top-down and bottom-up engineering of message-passing applications. This interplay can be described by the following diagram:



In the top-down engineering approach, designers provide the global view which can then be "projected" to obtain local views. Developers can independently

Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 and by the MIUR project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems).

Published by Springer Nature Switzerland AG 2020
S. Bliudze and L. Bocchi (Eds.): COORDINATION 2020, LNCS 12134, pp. 20–36, 2020.
https://doi.org/10.1007/978-3-030-50029-0_2

realise components and test their "compliance" with the corresponding local view. In the bottom-up approach, one can extract the local view from existing components, check for their compatibility, and generate a global view. Besides documenting the composition and enhancing program comprehension, global views yield an abstract model that can be used to evolve the system.

Choreographic development of message-passing applications offers, among others, the following advantages:

- Global views are amenable to being expressed using visual languages understandable to lay stakeholders (akin to BPMN [11] or UML diagrams [21], message-sequence charts [17], etc.).
- Local views can be algorithmically projected from global specifications. Also, projected local views:
  - come with correctness guarantees when global views satisfy sufficient conditions (*well-formedness*) for communication soundness,
  - are typically executable models that can be developed independently.
- The execution model of local views is close to several programming languages and environments like Golang, Erlang, Elixir, Akka, or JMS.

Model-driven approaches provide some support, but require care: models may be incomplete or have subtle issues that can lead to misbehaviour such as deadlocks or message loss.

Structure of the Paper. Section 2 surveys the models used in the tutorial. Section 3 shows how the top-down approach works in a simple scenario. Section 4 highlights some of the problems that a "bad design" may cause. Section 5 discusses how to analyse and fix design errors. Section 6 demonstrates the support that choreographic development may provide when amendments are necessary. Section 7 shows choreographic bottom-up engineering. Section 8 draws some conclusions.

2 Our Models

A global view can be represented in terms of a distributed workflow (e.g., BPMN [11] diagrams) or as specifications (similar to UML sequence diagrams or message-sequence charts). We survey the formal models that we use to represent global and local views of choreographies. This is an informal presentation; the referred literature yields the technical details.

Hereafter, we assume two disjoint sets: *participants* and *messages*, respectively ranged over by A, B, etc. and by m.

2.1 Partially Ordered Multisets

Pomsets [24] model concurrency in terms of partial orders of multisets of events. Pomsets provide a general model of concurrency; for instance message-sequence

charts are a particular class of pomsets. We use pomsets of communication events as defined in [25] (which also provide a formal overview of pomsets).

Roughly speaking, we consider pomsets as directed-acyclic graphs where nodes are labelled by communication actions and edges capture causality among communications. A simple example illustrates this. The following diagram

$$r_{(1)} = \left[\begin{array}{ccc} \mathsf{A}\,\mathsf{B}!\mathtt{int} & \longrightarrow & \mathsf{A}\,\mathsf{B}?\mathtt{int} \\ \downarrow & & \downarrow \\ \mathsf{A}\,\mathsf{B}!\mathtt{bool} & \longrightarrow & \mathsf{A}\,\mathsf{B}?\mathtt{bool} \end{array}\right] \tag{1}$$

represents a pomset $r_{(1)}$ capturing the causality relations among the communication events between participants A and B; in (1), horizontal arrows establish the causality relations induced by the interactions of participants, while vertical arrows order the communications events of each participant (marked with background colours for the sake of illustration). More precisely, A sends to B a message of type `int` and one of type `bool`; the two leftmost nodes are indeed the output events labelled with $\mathsf{A}\,\mathsf{B}!\mathtt{int}$ and $\mathsf{A}\,\mathsf{B}!\mathtt{bool}$ while the rightmost nodes are the corresponding input events $\mathsf{A}\,\mathsf{B}?\mathtt{int}$ and $\mathsf{A}\,\mathsf{B}?\mathtt{bool}$. Moreover, the edges establish that the each output event *precedes* the corresponding input event (the horizontal edges in (1)) and that the events involving integers precede those involving booleans. Intuitively, $r_{(1)}$ models process A that first sends a `int` and then a `bool` message to B, which on turn receives the messages in the sending order.

To represent alternative executions we simply take collections of pomsets. For instance, the set made of the following two pomsets

$$\left[\begin{array}{ccc} \mathsf{A}\,\mathsf{B}!\mathtt{int} & \longrightarrow & \mathsf{A}\,\mathsf{B}?\mathtt{int} \\ \downarrow & & \downarrow \\ \mathsf{A}\,\mathsf{B}!\mathtt{int} & \longrightarrow & \mathsf{A}\,\mathsf{B}?\mathtt{int} \end{array}\right] \qquad \left[\begin{array}{ccc} \mathsf{A}\,\mathsf{B}!\mathtt{bool} & \longrightarrow & \mathsf{A}\,\mathsf{B}?\mathtt{bool} \\ \downarrow & & \downarrow \\ \mathsf{A}\,\mathsf{B}!\mathtt{bool} & \longrightarrow & \mathsf{A}\,\mathsf{B}?\mathtt{bool} \end{array}\right] \tag{2}$$

represents a system where A sends B either two `int` or two `bool` messages.

We remark that events involving the same participants are not ordered unless explicitly stated by the pomset. For instance, if we remove the rightmost vertical edge imposing an order of the input events from the pomset (1) then the messages sent by A can be received in any order by B.

2.2 A Workflow Model

The global view of a choreography can be suitably specified as workflows. We use *global choreographies* [12,25] (g-choreographies for short), a structured version of *global graphs* [8,20]. This model is appealing as it has a syntactic and diagrammatic presentation amenable of a formal semantics in terms of pomsets.

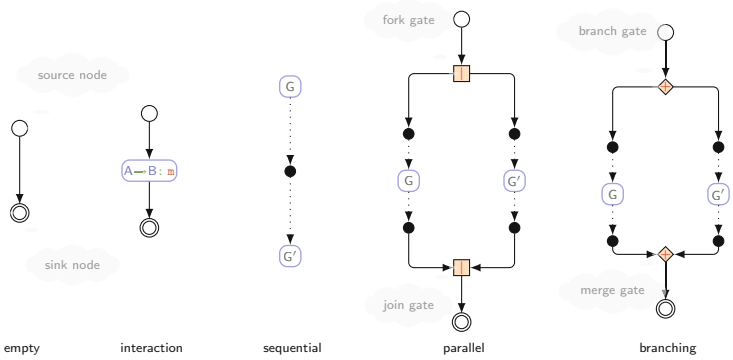


Fig. 1. A visual presentation of g-choreographies

The syntax of g-choreographies is given by the following grammar:

$$
\begin{array}{lll}
\mathsf{G} & ::= & (\mathsf{o}) \qquad\qquad\qquad\qquad\quad \text{empty} \\
& \mid & \mathsf{A}{\to}\mathsf{B}\colon \mathtt{m} \qquad\qquad\quad \text{interaction} \\
& \mid & \mathsf{G}\,;\,\mathsf{G} \qquad\qquad\qquad\quad \text{sequential} \\
& \mid & \mathsf{G} \mid \mathsf{G} \qquad\qquad\qquad\quad \text{fork} \\
& \mid & \mathsf{sel}\ \{\mathsf{G} \ + \ \cdots \ + \ \mathsf{G}\} \qquad \text{choice}
\end{array}
$$

Productions are given according to the decreasing order of precedence of connectives. Curly brackets can modify precedence. Iterative g-choreographies are omitted since they are not used in this tutorial.
The grammar of data types is left implicit; in examples we will assume that `m` ranges over basic types such as `int`, `bool`, `string`, etc.

The empty g-choreography (o) yields no interactions; trailing occurrences of (o) may be omitted. An interaction $\mathsf{A}{\to}\mathsf{B}\colon \mathtt{m}$ represents the exchange of a message of type `m` between A and B, provided that $\mathsf{A} \neq \mathsf{B}$. We remark that data values are abstracted away: the payload `m` in $\mathsf{A}{\to}\mathsf{B}\colon \mathtt{m}$ is not a value and should rather be thought of as (the name of) a data type. G-choreographies can be composed sequentially or in parallel ($\mathsf{G}\,;\,\mathsf{G}'$ and $\mathsf{G} \mid \mathsf{G}'$). A (non-deterministic) choice $\mathsf{sel}\ \{\mathsf{G}_1 \ + \ \cdots \ + \ \mathsf{G}_n\}$ specifies the possibility to continue according to either of the g-choreographies $\mathsf{G}_1, \ldots, \mathsf{G}_n$.

The syntax of g-choreographies can be visually depicted as in Fig. 1. A global graph G is represented as a rooted graph with a single "enter" and "exit" nodes, respectively called *source* (graphically ○) and *sink* (graphically ◎). Special nodes, dubbed *gates*, are used for branch and fork points (respectively depicted as ◆ and ▯). Each fork or branch gate in our visual notation will have a corresponding join and merge "closing" gate.

The semantics of a global graph is a family of pomsets; each pomset in the family partially orders the communication events on a particular "trace" of g-choreography. For instance, the semantics $[\![(\mathsf{o})]\!]$ is simply the set $\{\epsilon\}$ containing the empty pomset ϵ while for interactions we have

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

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,

- for a choice we have $[\![G + G']\!] = [\![G]\!] \cup [\![G']\!]$;
- the semantics of the parallel composition $G \mid G'$ is essentially built by taking the set of the disjoint union of each pomset in $[\![G]\!]$ with each one in $[\![G']\!]$;
- the semantics of the sequential composition $[\![G; G']\!]$ is the set of the disjoint union of each pomset in $[\![G]\!]$ with each one in $[\![G']\!]$ and adding causal relations from events in $[\![G]\!]$ to those in $[\![G']\!]$ if they are executed by the same participant (i.e., making the former precede the latter).

3 When All Goes Fine

We use a simple yet representative application to highlight a top-down choreographic approach to the design and prototyping of message-passing applications.

A server S allows client C to convert strings into a date format and vice versa (we assume a basic data type `dateFmt` to represent formats). Both C and S use a logger service L to record their requests and responses respectively; for this, data types `reqLog` and `resLog` are used.

We first consider a couple of solutions that straightforwardly model the scenario above. Take the global specification

```
sel {
  {C → L: logReq | C → S: dateFmt}; .. date2string & logging
     S → L: logRes;
     S → C: string
  +
  {C → L: logReq | C → S: string};  .. string2date & logging
     S → L: logRes;
     S → C: dateFmt
}
```

Despite its simplicity, it is not immediate that the g-choreography above is sound. To illustrate this, consider that

- the first interaction between the client C and the server allows S to determine which service is requested (convert a date to a string or vice versa),
- the logger behaves uniformly through the choice (since it first receives the log message of a request and then the one of a response).

Notice that, although L is oblivious of the choice, its behaviour cannot be syntactically factored out without violating some dependencies among communications.

A variant of the above g-choreography is

```
sel {
  {C → L: logReq | C → S: dateFmt};
     sel {
```

```
      S → L: logRes
      + .. S may send less informative logs
      S → L: basicLog
    };
    S → C: string
  +
  {C → L: logReq | C → S: string};
    S → L: logRes;
    S → C: dateFmt
}
```

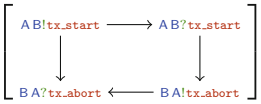
where, once a request to transform a date into a string is made, S may decide to log the response either fully or send L less information with `basicLog`. Note that C is now unaware of this choice between S and L while L can discern the initial choice made by C.

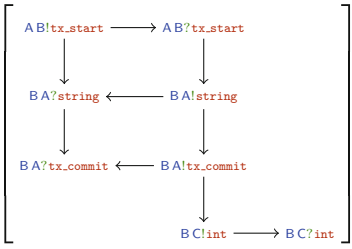
Once the soundness of a g-choreography is attained, local behaviours can be automatically projected either to local specifications or to executable code. For instance, ChorGram can generate *communicating finite-state machines* [6] that specify the behaviour of each component of the system as well as executable Erlang code implementing the communication pattern of the g-choreography. Notably, this approach is an instance of a *correctness-by-design* principle: the projected behaviour is *communication sound* in the sense that it does not exhibit misbehaviour such as deadlocks or loss of messages (provided that the communication infrastructure does not fail).

4 Designing Problems

Models featuring distributed workflows (such as BPMN or the g-choreographies adopted here) may introduce inconsistency related to distributed choices. In scenario-based models, the designer may overlook cases that may lead to runtime errors. This is illustrated with the following examples.

Consider the protocol specified by the following two pomsets





(3)

Participant A starts a transaction with B by sending message `tx_start` and then engages in a distributed choice where either B aborts the transaction immediately, or it send a string with A and commits the transaction before sending an integer with C. This specification leaves C uncertain about whether the integer from B is going to be sent or not. Hence C could locally decide to

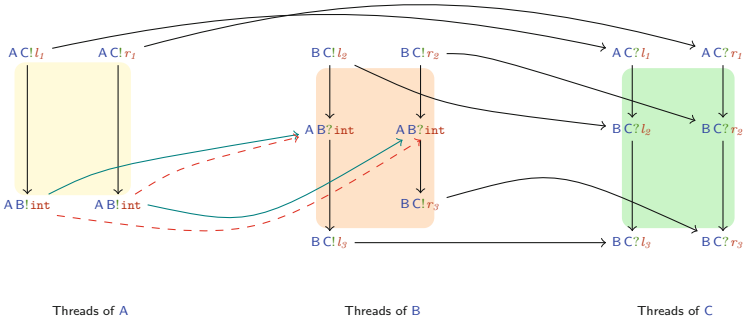


Fig. 2. Inter-participant closure

(a) terminate immediately and not receive the integer from B or
(b) to wait for the integer, even if B opted to abort the transaction.

Relying on the pomset semantics of g-choreographies, in [25] we defined *termination awareness* in order to avoid reaching run-time configurations where non-terminating participants unnecessarily lock resources once the coordination is completed. This condition requires that in no accepting configuration the participants of interest have input transitions. More precisely, a set of pomsets R violates the terminating condition for participant A if there are two pomsets r and r' in R such that an execution trace of A in r is a prefix of an execution trace of A in r' and the difference between the two traces starts with an input. Basically, the designer can specify which participants should be aware of the termination of the choreography. Note that, depending on the application requirements, termination awareness may be important for B, but not for C; in the example above for instance, the termination of C is not crucial if it is not locking resources or if it is immaterial that such resources are left locked.

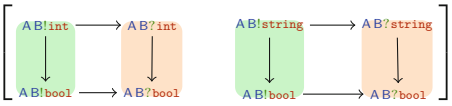
In [25], besides the terminating condition above, we identified two other pomset-based conditions dubbed **CC2-POM** and **CC3-POM** to check realisability of global specifications. We briefly describe those *closure* conditions.

Intuitively, **CC2-POM** takes into consideration the executions emerging from "confusion" caused by different threads in a pomset (such as the one in (4)). For a given set of pomsets R, the satisfaction of **CC2-POM** ensures that if an execution trace t cannot be distinguished by any of the participants from a valid trace of R then t is a trace of R. To check **CC2-POM**, one needs to compute a closure set of pomsets out of R; the closure set yields the pomsets characterising the execution traces due to inter-participants' concurrency. This closure generates all "acceptable" matches between output and input events entailed by a pomset capturing the behaviour of a participant. We borrow and adapt Fig. 2 from [25] to give an intuition of this construction. The participants A, B, and C there have each two threads, the "left" and the "right" thread; those are identified by sorts l_i and r_i which are meta-variables on sorts immaterial here (with the only assumption that $l_i \neq r_i$ for each $i \in \{1, 2, 3\}$). The bottom-most solid edges

from the threads of A to those of B represent the causality relations specified in the original design. The closure of the original design however yields also the execution with the causality relations given by the bottom-most dashed edges.

The condition **CC3-POM** accounts for implied executions that may break distributed choices. It is similar to **CC2-POM** barred that the closure set is built by checking all the prefixes of the traces of the pomsets.

We remark that our framework focuses on the identification of communication problems that are data-oblivious. For this reason the implementations of some specifications may exhibit unintended interactions even if they satisfy the verification conditions. Consider the following example:



(4)

which specifies two concurrent threads[1] whereby A (threads with green background) and B (threads with orange background) exchange two boolean values after exchanging an integer and a string. In an asynchronous setting, the boolean values may "swap", namely the one sent by the thread which sent the integer is received by the thread which received the string, and vice-versa. Notice that this does not yield communication problems, but may violate the dependencies among data induced by the causal dependencies specified in the pomset.

To sum up, we address termination awareness, thread confusion (**CC2**), and undetermined choices (**CC3**). The violation of termination awareness could lead to participants oblivious of the termination of the protocol, the violation of **CC2** could make messages to be consumed by a unintended threads of a participant, **CC3** could lead to participants to follow one branch of a choice while other participants are executing another branch.

5 When Something Goes Wrong

We now consider some examples where development is not as straightforward as in the examples of Sect. 3. More precisely, we consider scenarios where the closure properties **CC2-POM** or **CC3-POM** may be violated, or termination awareness does not hold for some participants. For the analysis we rely on PomCho, part of the ChorGram tool chain that supports choreographic development through the models discussed in Sect. 2.

Let us start with the following (erroneous) g-choreography giving another variant of the protocol in Sect. 3 where a less informative log message is sent when C requests to transform a date in a string and a more informative one otherwise:

[1] Note the bracketing here: enclosing the two groups of events in different brackets would correspond to specifying a choice between the pomsets.

```
sel {
  {C → L: logReq | C → S: dateFmt};
     L → S: basicLog;                    .. oops
     S → C: string
  +
  {C → L: logReq | C → S: string};
     S → L: logRes;
     S → C: dateFmt
}
```

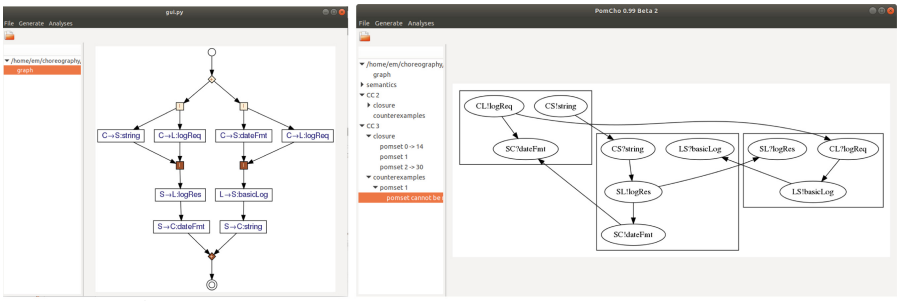
The attentive reader may have noticed a problem: the exchange of basicLog should go the other way around. This intentional mistake is instrumental to illustrate the analysis of the g-choreography above, summarised by the screenshots in Fig. 3. Before doing so, we briefly digress about the GUI. After loading, ChorGram computes the g-choreography to analyse as shown in the left-most screenshot in Fig. 3. The other screenshot presents a counterexample of the violation of termination awareness once such analysis is executed. Note that the hierarchical menu in the left pane is dynamically expanded to include clickable references to the results of the operations. The right-hand side pane of the second screenshot, represents the pomset of the first branch of the previous g-choreography. This pomset is a counterexample showing the violation of **CC3-POM** (as shown on the hierarchical menu). The events performed by each participant are grouped with a box to make the pomset clearer to the user.

Let us return to our analysis. The screenshots in Fig. 3 show that while the closure properties are satisfied, termination awareness is violated for L. By inspecting the pomset in the top-right screenshot we can notice that the logging information in the two branches wrongly goes from L to S (instead of going the other way around). This is immediately evident from the projection on L reported in the bottom-most screenshot of Fig. 3; notice that state 1 is a mixed-choice state, namely that it has both input and output outgoing transitions.

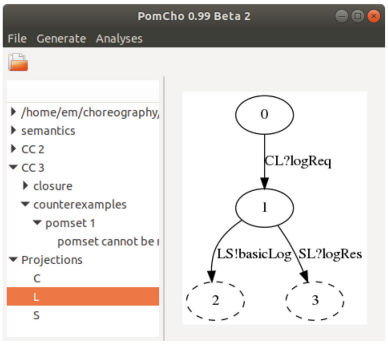
Swapping the sender and the receiver in the introduced interaction solves all the issues (which results in ChorGram displaying empty lists of counterexamples). One could argue that this is such a blunt glitch that one could spot it immediately and without the use of tools. While this might be true for simple examples like this one, these mishaps might not be as obvious in larger designs.

The next variant of our protocol exhibits a subtle problem. Consider

```
sel {
  {C → L: logReq | C → S: dateFmt};
     S → L: basicLog;
     S → L: logReqExt;
     S → C: string
  +
  {C → L: logReq | C → S: string};
     S → L: logReqExt;
     S → L: basicLog;
     S → C: dateFmt
}
```

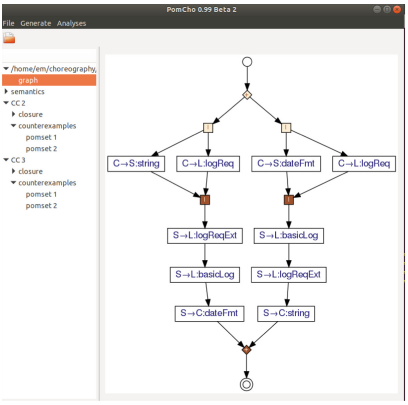


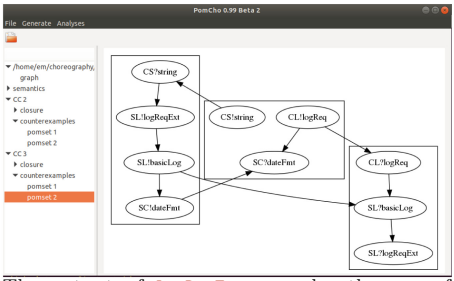
The left-hand pane allows the user to select the model onto which to apply the next operation; results are displayed in the pane on the right-hand side.



A possible way of representing local views is by *communicating finite-state machines* [6] which basically are finite-state automata where transitions are labelled by communication actions. A projection function of ChorGram can generate CFSMs of participants from a g-choreography as illustrated for L here.

Fig. 3. Violation of termination awareness and projection on L





The output of logReqExt precedes the one of basicLog (implying that S opted for the left-most branch of the g-choreography). However, the corresponding input events of L are in the opposite order (making L follow the right-most branch).

Fig. 4. Problems with non-FIFO asynchrony

The analysis must take into account the semantics of communication. In particular, asynchronous communications require care. In fact, if the messages are buffered and accessed according to a FIFO discipline, in the above g-choreography the message basicLog is before (resp. after) the message

logResExt in the buffer from S to L when the first (resp. second) branch is chosen. However, when the order of sent messages is not guaranteed problems may arise, as highlighted by the screenshots of PomCho in Fig. 4. Although the screenshot on the left suggests that the interactions happen in the order specified in the g-choreography, L may "misunderstand" the choice taken by C. In fact, suppose that C selects the left branch and that the message basicLog may reach L before the message logResExt and, consequently, L may behave according to the branch on the right. Both closure properties are violated because the order of messages no longer allows L to distinguish which branch S opted for. The pomset depicted in the screenshot on the right shows possible executions where the order of outputs is not preserved.

The above problem can be fixed by letting L acknowledge the first message from the server S, namely

```
sel {
  {C → L: logReq | C → S: dateFmt};
     S → L: basicLog;
     L → S: ack;                          .. L acknowledges S
     S → L: logReqExt;
     S → C: string
  +
  {C → L: logReq | C → S: string};
     S → L: logReqExt;
     L → S: ack;                          .. L acknowledges S
     S → L: basicLog;
     S → C: dateFmt
}
```

This variant enjoys all our closure conditions.

6 Suggesting Amendments

This section illustrates an experimental feature recently added to ChorGram. As seen earlier, the top-down approach of choreographic design requires g-choreographies to enjoy well-formedness properties. For instance, the closure properties surveyed in Sect. 4 ensure the realisability of g-choreographies by components coordinating through asynchronous message-passing.

Attaining well-formedness requires some ingenuity. In fact, designers can easily overlook problems and introduce defects leading to communication problems such as those in the scenarios of Sect. 5. When this happens, as seen in the previous examples, ChorGram identifies counterexamples that highlight defects. It may therefore be helpful to have advice on how to possibly fix problems.

Possible amendments are suggested by ChorGram as a g-choreography, determined out of the initial one and the counterexample identified in the analysis. We demonstrate this by considering a further variation of the application used in Sect. 5. The following g-choreography models a protocol where a less informative log message is sent when C requests to transform a date into a string:

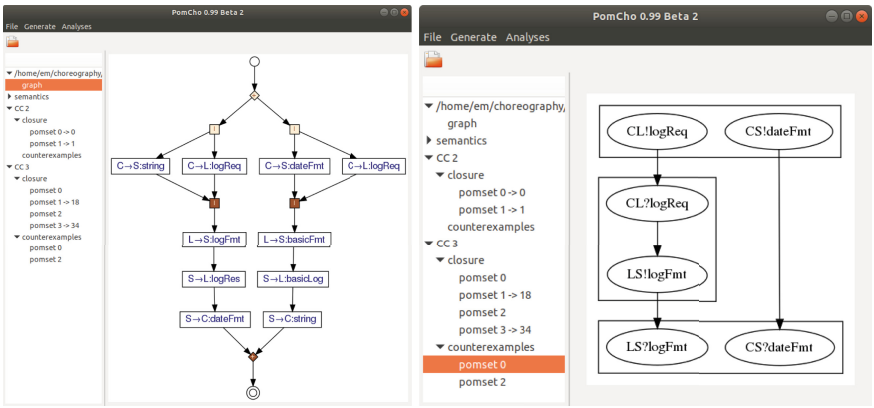


Fig. 5. An unexpected execution

```
sel {
  {C → L: logReq | C → S: dateFmt};
      L → S: basicFmt;
      S → L: basicLog;
      S → C: string
  +
  {C → L: logReq | C → S: string};
      L → S: logFmt;
      S → L: logRes;
      S → C: dateFmt
}
```

Notice that L informs S about the format of the log. The analysis in Fig. 5 shows that, while **CC2-POM** is satisfied (since no counterexample exists), **CC3-POM** is however violated due to an unsound choice. In fact, the screenshot on the right of Fig. 5 represents the pomset missing from the semantics of the global model. In this counterexample, S gets stuck after receiving the two parallel inputs from C and L. Here, the problem arises because L cannot identify which message should be sent to S, since L receives the same message in both branches. Accordingly, L is not informed about the branch selected by C. This problem is evident in the screenshots of Fig. 6, which report the projection of L, the g-choreography corresponding to the counterexample, and a model consisting of a suggested amendment. This model maps the counterexample back to the initial g-choreography. This is attained by computing the "minimal" transformation required on the original design to match the counterexample. More precisely, ChorGram applies an edit-distance algorithm to the (part of the) original design to be changed so that it corresponds to the counterexample. The algorithm can be tuned up by setting a *cost* to edit operations.

The edit operations are

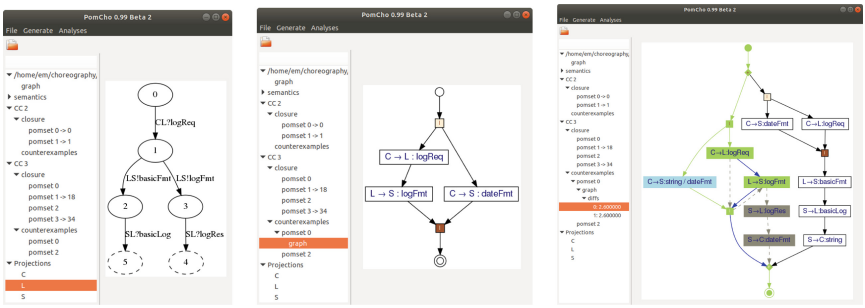


Fig. 6. A possible amendment (Color figure online)

– node insertion/deletion
– edge insertion/deletion
– modifications to sender/receivers/payload of interactions.

Pictorially, nodes and edges in green represent unchanged elements, those in blue represent additions to make, and those in dashed gray represent elements to be deleted. For instance, the amendment in Fig. 6 suggests to preserve the right-most branch and change the left-most branch by removing the nodes and edges in gray, adding the blue edges, and modifying the payload of the communication from C to S (from `string` to `dateFmt`).

Note that the suggested amendments are computed without "interpreting" the g-choreography and therefore they may not be meaningful. The designer still needs to vet and approve them. As said, this is an experimental feature added to ChorGram, and we plan to investigate how to improve it. For instance, an interesting development could be to identify how to assign costs depending on the applications at hand. In fact, in some cases it may not be reasonable to apply some of the operations above. This can be improved upon by properly assigning costs (undesired operations should have a higher cost than admissible ones).

7 Going Bottom-Up

We now consider how choreographies can support bottom-up engineering. There are two key motivations for which this support is appealing. Firstly, the validation of composition of distributed components. For instance, service-oriented architectures such as micro-services envisage software development as the composition of publicly available services. One would like to validate that such compositions communicate as expected. Secondly, software evolution possibly compromises the communication soundness of an application.

In the rest of the section, such a scenario is used to show the kind of support offered by choreographies in bottom-up engineering. To this purpose we look at a possible evolution of the last g-choreography in Sect. 5: Suppose that the developers want to deploy a new version of the logger service L that requires

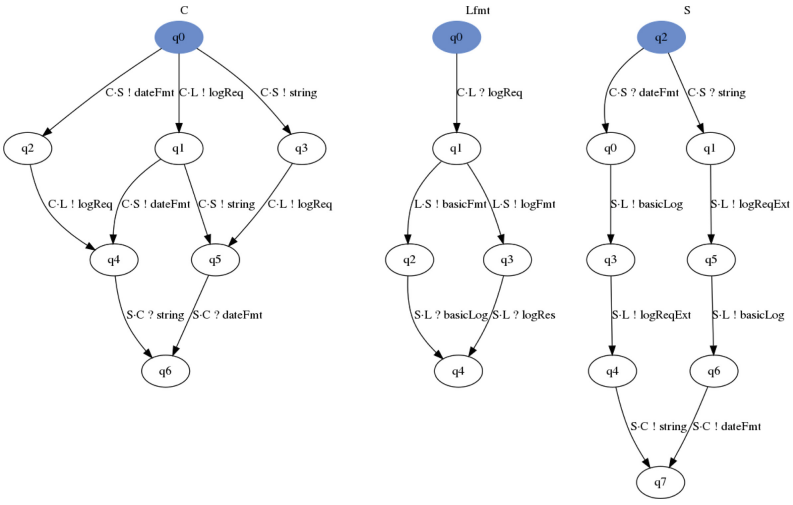


Fig. 7. The system with the evolved L

specific formats for the log information sent by the server S. Therefore, developers implement a new version of L, say Lfmt, that behaves according to the CFSM in Fig. 7. (Note that the CFSMs of C and S do not change.) After the request of the client, the new logger informs S of the format (transitions q1→q2 and q1→q3 of Lfmt). The server reacts accordingly with transitions from its initial state.

It is worth remarking that what usually happens is that the code implementing L evolves without modifying the corresponding models (if any). After a new version of a component is released, one could extract[2] a model like the CFSM on the left to describe its behaviour. This is what we assume in this scenario.

Figure 7 yields a problematic system: The bottom-up analysis of ChorGram (done before deploying Lfmt) flags the problem with the message:

```
Branching representability: [Bp "C" "q4", Bp "C" "q5", Bp "C" "q6",
Bp "L" "q1",Bp "L" "q2",Bp "L" "q3",Bp "L" "q4",
Bp "S" "q0",Bp "S" "q1",Bp "S" "q3",Bp "S" "q4",
Bp "S" "q5",Bp "S" "q6",Bp "S" "q7"]
```

which we now decipher. This message basically reports the CFSMs whose transitions are not reflected in their parallel composition due to some branching.

For instance, in our model, `Bp"S" "q0"` states that some transitions expected from state `q0` of the server S cannot be fired in a configuration of the systems where the local state of S is `q0`; in fact, in configuration `q4` of the system, the local machine of S is in state `q0`, which has the transition S L!`basicLog` (not reflected in the system).

[2] This operation can be done either by *inferring* the model from the code (if possible) or by *learning* it by observing the behaviour of the new version of the component.

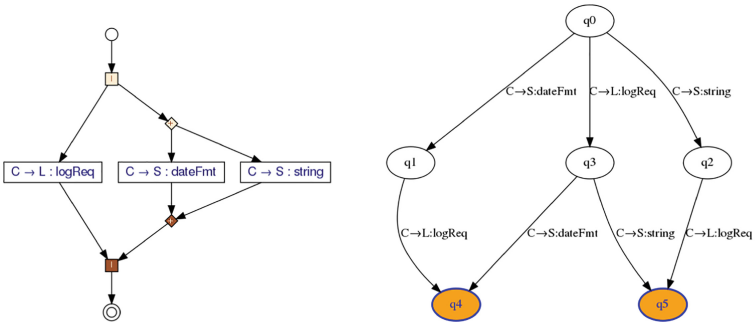


Fig. 8. G-choreography and transition system determined by bottom-up analysis

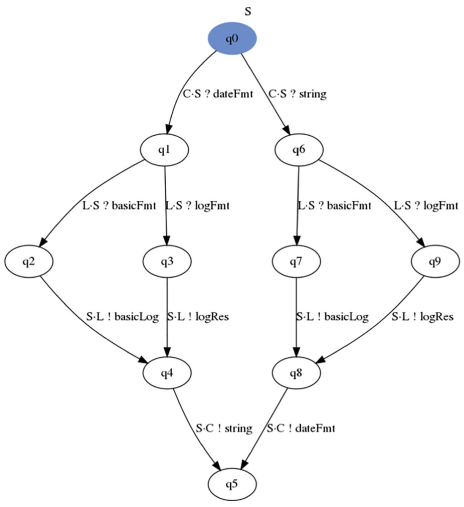


Fig. 9. An amended S

This analysis can be conducted in a more user-friendly way by inspecting the models in Fig. 8, also provided by ChorGram.

The g-choreography contains the "sound" interactions only. In fact, the transition system in Fig. 8 highlights two configurations, `q4` and `q5`, that violate the branching property. This is due to the fact that S is not aware of the choice taken in the local state `q1` of the CFSM Lfmt in Fig. 7. This problem can be solved by modifying S, as illustrated in the CFSM of Fig. 9.

8 Concluding Remarks

Tool support for analyzing realisability of global specifications is necessary to enable model-driven development of choreographies. Indeed, as observed in [1], a source of problems is that there could be some specifications that are impossible

to implement using the local views in a given communication model. Several works addressed the realisability of scenario-based global models, like *message-sequence charts* (MSCs) [2,9,14–16,22,23].

A mechanism to statically detect realisability in MSCs is proposed in [4]. The notions of non-local choices and of termination considered in [4] are less permissive than our verification conditions since intra-participant concurrency is not allowed and termination awareness is not enforced. Closure conditions for realisability have been initially proposed in [1] to study realisability of MSC and have been extended in [13] to handle sets of pomsets (the latter were reviewed in Sect. 2). This extension yields more general results and more efficient analyses, since it enables multi-threaded participants and does not require to explicitly compute all possible executions (which can be large due to the number of possible interleaving of concurrent threads) of the global model.

In the context of choreographies, the integration of ChorGram features in the CHOReVOLUTION platform [3] is of particular interest to us. In fact, CHOReVOLUTION is a rather sophisticated platform for the top-down development featuring many important aspects complementary to the functionalities of ChorGram (e.g., low-level binding of components, or security aspects).

Other possible integrations are with tools based on behavioural types [19]. The tools proposed in this context (see [10] for a survey) are typically based on theories defining constraints aimed to guarantee the soundness of the projections of global specifications (as e.g., in [5,7,18]). A peculiarity of ChorGram is that it can provide some feedback to support model-driven engineering of applications. This is not usually the case in other contexts based on behavioural types where, for instance, behavioural type checkers do not provide feedback.

Acknowledgments. We warmly thank Simon Bliudze for his helpful comments which allowed us to improve the presentation of this paper.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Softw. Eng. **29**(7), 623–633 (2003)
2. Alur, R., Holzmann, G.J., Peled, D.: An analyzer for message sequence charts. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 35–48. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_37
3. Autili, M., Di Salle, A., Gallo, F., Pompilio, C., Tivoli, M.: CHOReVOLUTION: automating the realization of highly–collaborative distributed applications. In: Riis Nielson, H., Tuosto, E. (eds.) COORDINATION 2019. LNCS, vol. 11533, pp. 92–108. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22397-7_6
4. Ben-Abdallah, H., Leue, S.: Syntactic detection of process divergence and non-local choice in message sequence charts. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 259–274. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0035393
5. Bocchi, L., Melgratti, H., Tuosto, E.: Resolving non-determinism in choreographies. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 493–512. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_26

6. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
7. Carbone, M., Honda, K., Yoshida, N.: A calculus of global interaction based on session types. Electron. Notes Theor. Comput. Sci. **171**(3), 127–151 (2007)
8. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_10
9. Gaudin, E., Brunel, E.: Property verification with MSC. In: Khendek, F., Toeroe, M., Gherbi, A., Reed, R. (eds.) SDL 2013. LNCS, vol. 7916, pp. 19–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38911-5_2
10. Gay, S., Ravara, A. (eds.): Behavioural Types: From Theory to Tools. Automation, Control and Robotics. River, Gistrup (2009)
11. Object Management Group: Business Process Model and Notation (2011). http://www.bpmn.org
12. Guanciale, R., Tuosto, E.: An abstract semantics of the global view of choreographies. In: Interaction and Concurrency Experience, pp. 67–82 (2016)
13. Guanciale, R., Tuosto, E.: Realisability of pomsets. J. Log. Algebr. Methods Program. **108**, 69–89 (2019)
14. Gunter, E.L., Muscholl, A., Peled, D.A.: Compositional message sequence charts. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 496–511. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_34
15. Gunter, E.L., Muscholl, A., Peled, D.: Compositional message sequence charts. Int. J. Softw. Tools Technol. Transfer **5**(1), 78–89 (2002). https://doi.org/10.1007/s10009-002-0085-2
16. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-19029-2
17. Harel, D., Thiagarajan, P.: Message sequence charts. In: Lavagno, L., Martin, G., Selic, B. (eds.) UML for Real, pp. 77–105. Springer, Boston (2003). https://doi.org/10.1007/0-306-48738-1_4
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). Extended version of a paper presented at POPL08
19. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
20. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 221–232 (2015)
21. Micskei, Z., Waeselynck, H.: UML 2.0 sequence diagrams' semantics. Technical report, LAAS (2008)
22. Formal description techniques (FDT) - Message Sequence Chart (MSC). Recommendation ITU-T Z.120 (2011). http://www.itu.int/rec/T-REC-Z.120-201102-I/en
23. Muscholl, A., Peled, D.: Deciding properties of message sequence charts. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 43–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11495628_3
24. Pratt, V.: Modeling concurrency with partial orders. Int. J. Parallel Prog. **15**(1), 33–71 (1986)
25. Tuosto, E., Guanciale, R.: Semantics of global view of choreographies. J. Log. Algebr. Methods Program. **95**, 17–40 (2018)