6

A Tool for Choreography-Based Analysis of Message-Passing Software

Julien Lange[1], Emilio Tuosto[2] and Nobuko Yoshida[1]

[1]Imperial College London, UK
[2]University of Leicester, UK

Abstract

An appealing characteristic of choreographies is that they provide two complementary views of communicating software: the *global* and the *local* views. Communicating finite-state machines (CFSMs) have been proposed as an expressive formalism to specify local views. Global views have been represented with global graphs, that is graphical choreographies (akin to BPMN and UML) suitable to represent general multiparty session specifications. Global graphs feature expressive constructs such as forking, merging, and joining for representing application-level protocols.

An algorithm for the reconstruction of global graphs from CFSMs has been introduced in [17]; the algorithm ensures that the reconstructed global graph faithfully represents the original CFSMs provided that they satisfy a suitable condition, called generalised multiparty compatibility (GMC). The CFSMs enjoying GMC are guaranteed to interact without deadlocks and other communication errors. After reviewing the basic concepts underlying global graphs, communicating machines and safe communications, we highlight the main features of **ChorGram**, a tool implementing the generalised multiparty compatibility and reconstruction of global graphs of [17]. We overview the architecture of **ChorGram** and present a comprehensive example to illustrate how it is used directly to analyse message-passing software and programs.

6.1 Introduction

Choreographic approaches are becoming popular in the "top-down" development of distributed software. In fact, a choreography-based model features two views of software: the *global view* and the *local view*. The former is a "holistic" specification of the behaviour of the composition of *all* components (and it abstracts away low level details such as asynchrony). The latter view specifies the behaviour of each component in isolation and should be obtained by *projecting* the global behaviour with respect to each component. In this framework, well-formedness of the global view and compliance of the realisation of software with respect to the corresponding projection should guarantee the soundness of the communication of the application.

The recent rise of services, cloud, and micro-services is changing the way software is produced. As a matter of fact, applications are being developed by composing (possibly distributed) independent components which coordinate their execution by exchanging messages. Modern applications offer and rely on public APIs to interact with each other, are deployed on different architectures and devices, and try to accommodate the needs of a vast number of users. The term "API economy" (see e.g., `ibm.com/apieconomy`) has been coined to refer to such applications. Existing and novel languages, as well as middlewares and programming models, foster this shift in software development. Languages such as Erlang, Elixir, Scala, and Go are paramount examples of this shift and start to be used in a wider range of application domains than the ones they were originally conceived for. For instance, Erlang plays a main role in well-known applications such as WhatsApp [24] and Facebook [21].

The trend described above is dictated by the compelling requirements of openness, scalability, and heterogeneity and caters to new challenges. Firstly, this shift pushes the applicability of top-down software development approaches to their limits. The composition mechanisms required to guarantee the interoperability of applications have to be of an order of magnitude more sophisticated than just the type signature of their APIs, as in traditional software engineering practice. More precisely, in order to attain a correct composition, it is crucial to expose (part of) the communication pattern of components. Hence, developers are responsible to guarantee the correct composition of their components. This is not an easy task. Subtle and hard to fix bugs can be introduced by inappropriate communications.

Our recent work [17] has shown that communication soundness is guaranteed when a set of communicating components enjoys the *generalised multiparty compatibility* property. Moreover, we have defined an algorithm that reconstructs a global view of a system from the composition of its local components. These results enable the realisation of an effective tool-supported approach to the design and analysis of communicating software. In fact, we have developed **ChorGram** [16], a tool supporting the theory of multiparty compatibility and choreography construction, i.e., **ChorGram** implements two fundamentals functionalities: it ensures that a system of CFSMs validates the GMC condition and if so, it returns a choreography which faithfully captures the interactions of the original system.

In this chapter, we introduce **ChorGram** and show how it supports software architects in the design and analysis of software. We first review the theoretical results underlying the tool; Section 6.2 presents our theory only informally and with the help of a simple example. Section 6.3 presents the architecture of the tool, how it integrates with the auxiliary tools it relies upon, and its data flow. Section 6.4 shows an application to a non trivial example. We start from a multiparty compatible application and show how a naive evolution could break its multiparty compatibility. We then use **ChorGram** to analyse and fix the problem. Section 6.5 gives our concluding remarks.

6.2 Overview of the Theory

Here we introduce the key ingredients of our framework which constructs choreographies, i.e., global graphs such as the one in Figure 6.2, from local specifications, i.e., communicating finite-state machines, such as the ones in Figure 6.1.

CFSMs In this framework, we use *communicating finite-state machines* [7] as behavioural specifications of distributed components (i.e., end-point specifications) from which a choreography can be built. CFSMs are a conceptually simple model and are well-established for analysing properties of distributed systems. A system of CFSMs consists of a finite number of automata which communicate with each other through unbounded FIFO channels. There are two channels for each pair of CFSMs in the system, one in each direction. We present the semantics of CFSMs informally through the example below.

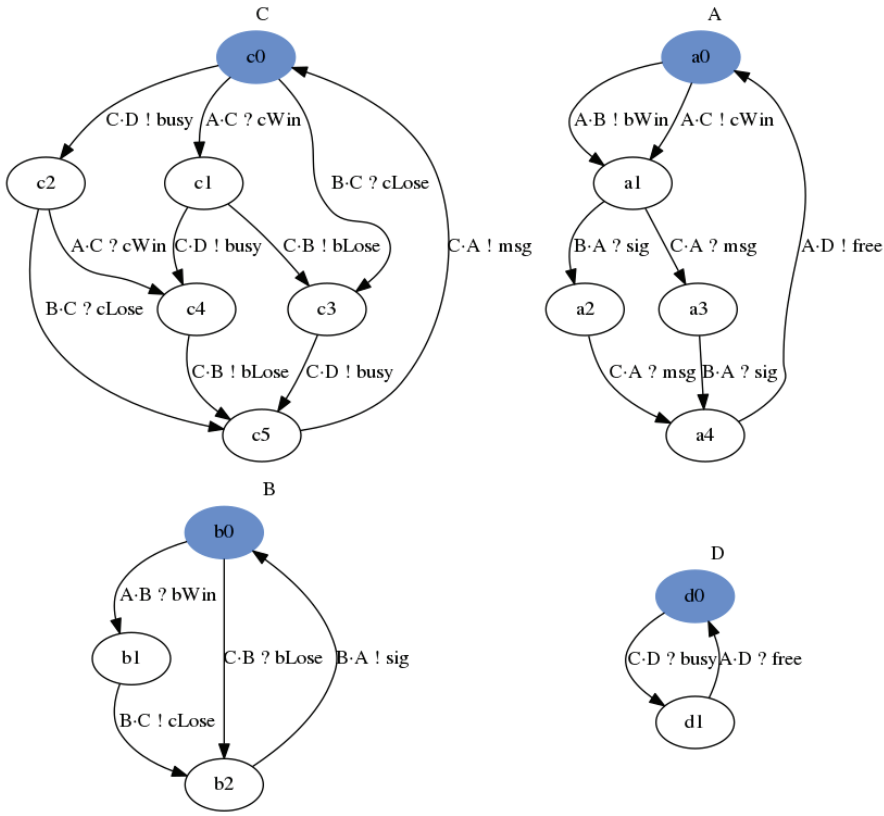


Figure 6.1 Four player game – CFSMs.

Consider the system of four machines in Figure 6.1, whose initial states are highlighted in blue. Each machine has *three* input buffers to receive messages from the other three participants and has access to three output buffers to send messages to other participants. Each transition in a machine is either a send action, e.g., $\mathsf{A} \cdot \mathsf{B}!\mathsf{bWin}$ in machine A or a receive action, e.g., $\mathsf{A} \cdot \mathsf{B}?\mathsf{bWin}$ in machine B. The system realises a protocol of a fictive game where: A*lice* (A) sends either bWin to B*ob* (B) or cWin to C*arol* (C) to decide who wins the game. In the former case, A fires the transition $\mathsf{A} \cdot \mathsf{B}!\mathsf{bWin}$ whereby the message bWin is put in the FIFO buffer AB from A to B, and likewise in the latter case. If B wins (that is the message bWin is on top of the queue AB and B consumes it by taking the transition $\mathsf{A} \cdot \mathsf{B}?\mathsf{bWin}$), then he

sends a notification (cLose) to C to notify her that she has lost. Symmetrically, C notifies B of her victory (bLose). During the game, C notifies D*ave* (D) that she is busy.

After B and C have been notified of the outcome of the game, B sends a signal (sig) to A, while C sends a message (msg) to A. Once the result is sent, A notifies D that C is now free and a new round starts.

Global graph The final product of our framework is the construction of a choreography which is equivalent to the original system of CFSMs. Global graphs [17] were inspired by the generalised global types [10] and BPMN choreography [19]. Given as input the CFSMs from Figure 6.1, our tool generates the global graph in Figure 6.2. The nodes of a global graph are labelled according to their function: a node labelled with ○ indicates the starting point of the interactions; a node labelled with ◎ indicates the termination of the interactions (not used in Figure 6.2); a node labelled with an interaction A → B : msg indicates that participant A sends a message of type msg to B; a node labelled ◈ indicates either a choice, merge, or recursion; a node labelled with ⊡ indicates either the start or the end of concurrent interactions. The graphical notation for branch and merge is inspired by process-algebraic notations; the reader familiar with BPMN should note that our ◈ -node corresponds to the ◇ and ◇ gateways in BPMN, while our ⊡ -node corresponds to the ◇ gateway in BPMN.

In the global graph of Figure 6.2, the flow of the four player game becomes much clearer. In particular, one can clearly see that either B or C win the game and that, while the results of the game are being announced, C and D are interacting.

Communication soundness properties A (runtime) configuration of a system of CFSMs, is a tuple consisting of the states in which each machine is and the content of each channel.

We say that a machine is in a sending (resp. receiving) state if all its outgoing transitions are send (resp. receive) actions. A state without any outgoing transition is said to be final. A state that is neither final, sending nor receiving is a mixed state.

We say that a configuration is a *deadlock* if all the buffers are empty, there is at least one machine in a receiving state, and all the other machines are either in a receiving state or a final state. A system has the *eventual reception property* [5] if whenever a message has been sent by a participant,

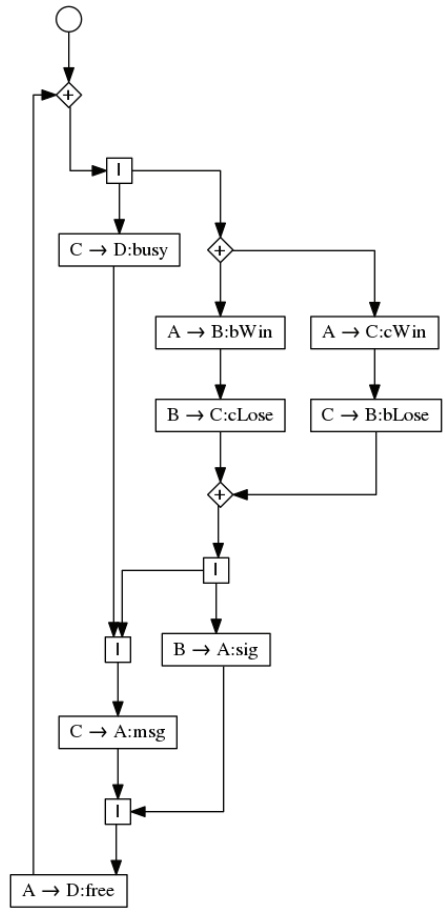


Figure 6.2 Four player game – Global graph.

that message will be eventually received. We say that a system of CFSMs is *communication sound* if none of its reachable configuration is a deadlock and it validates the eventual reception property.

Ensuring communication soundness Our tool checks that the CFSMs validate *generalised multiparty compatibility* (GMC) [17] which guarantees that (*i*) the projections of the generated global graph are equivalent to the original system and (*ii*) the system is *communication sound* (as defined above).

The GMC condition consists of two parts: *representability* and *branching property*. Both parts are checked against the machines and their synchronous executions, i.e., the finite labelled transition system (dubbed TS_0) of the machines executing with the additional constraint that a message can be sent only if its partner is ready to receive it and no other messages are pending in other buffers. For instance, all the synchronous executions of our running example are modelled in the finite labelled transition system in Figure 6.3.

The representability condition essentially requires that for each participant, the projection of TS_0 onto that participant yields an automaton that is bisimilar to the original machine. The branching property condition requires that whenever a branching occurs in TS_0 then either (*i*) the branching commutes, i.e., it corresponds to two independent (concurrent) interactions, or (*ii*) it corresponds to a choice and the following constraints must be met:

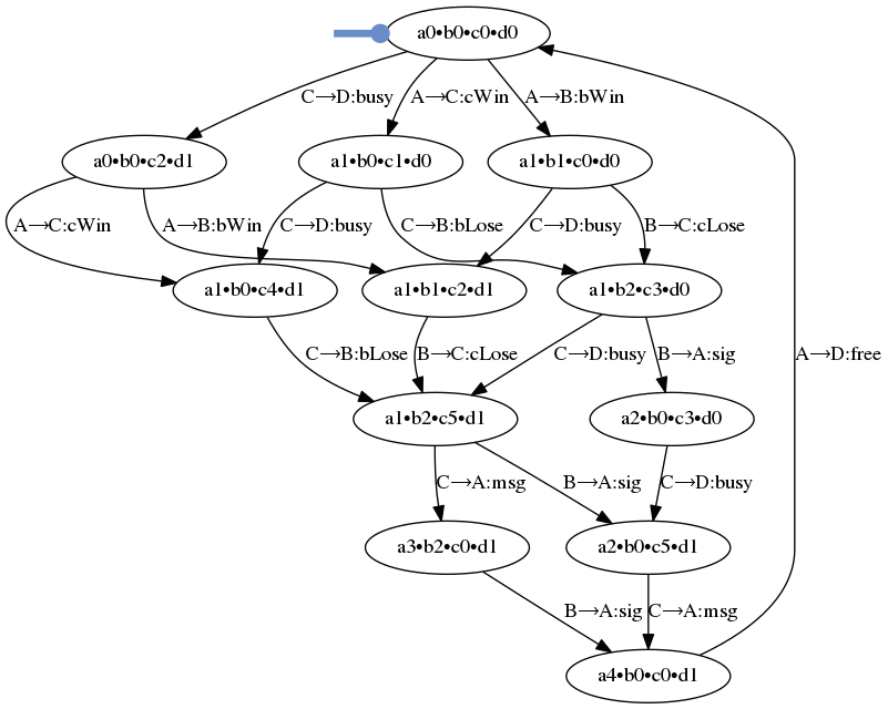


Figure 6.3 Four player game – TS_0.

1. The choice is made by a single participant (the selector).
2. If a participant is not the selector but behaves differently in two branches of the choice, then it must receive different messages in each branch (before the point where its behaviours differ).

Item (1) guarantees that every choice is located at exactly one participant (this is crucial since we are assuming *asynchronous* communications). Item (2) ensures that all the participants involved in the choice are made aware of which branch was chosen by the selector.

Besides guaranteeing communication soundness, our GMC condition ensures that if a system of CFSMs validates it, then we can construct a global graph which is equivalent to the original system, i.e., the global graph contains exactly the same information than the system of CFSMs.

6.3 Architecture

The structure and the work-flow of our tool is illustrated in Figure 6.4. Before commenting on the diagram, we explain its graphical conventions. Dashed arrows represent files used to exchange data; the input files are provided by the user, those of **hkc** are generated by the Haskell module **Representability**. Solid arrows do not represent invocations but rather control/data flow. For instance, the arrow from **TS** represents the fact that the check of the GMC property is made by concurrent threads on the results produced by **TS**.

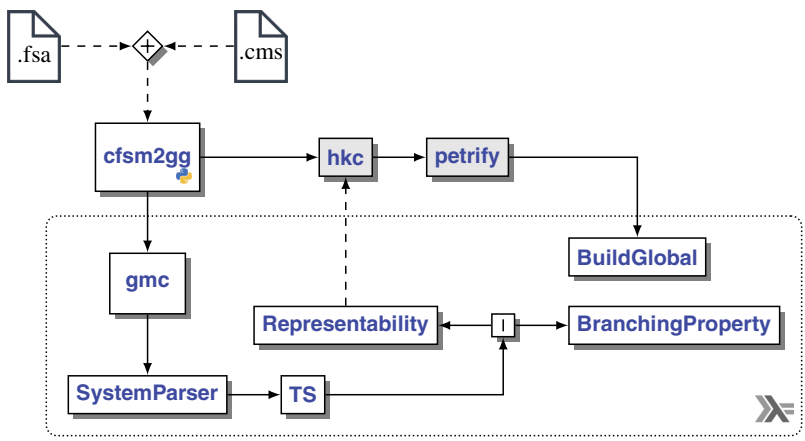


Figure 6.4 Architecture of **ChorGram**.

Data- and control-flow The Python script **cfsm2gg** provides a command line interface to the application and connects it with the external tools **hkc** [6] and **petrify** [8], respectively used to check language equivalence between projections and their corresponding CFSMs and to extract a Petri net from a transition system. The script takes a description of the system in (a file that is in) either of the two formats described in the following paragraph and triggers all the other activities.

The core functionalities are implemented in the Haskell modules (within the dotted box) and are described below.

gmc is the main program; it is invoked by **cfsm2gg**, which passes over the input file (after having set some parameters according to the flags of the invocation). After invoking **SystemParser**, the internal Haskell representation of the system of CFSMs is passed by **gmc** to **TS**, which computes the synchronous transition system – TS_0 (and the bounded one if required with the `-b` flag of **cfsm2gg**). The synchronous transition system is then checked for generalised multiparty compatibility [17, Definitions 3.4(ii) and 3.5] (but for the language equivalence part [17, Definition 3.4(i)] later checked by invoking **hkc** from **cfsm2gg**). This check is performed in parallel and has the side effect of producing the files to give in input to **hkc**.

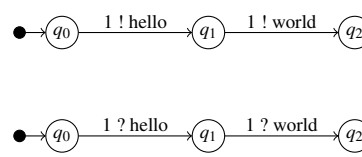
cfsm2gg invokes **hkc**, once it has obtained the control back from **gmc**, to check the language equivalence of the original CFSMs with respect to the corresponding projections of the synchronous transition system. Finally, **petrify** is invoked and its output is then transformed by **BuildGlobal** as described in [17] to obtain a global graph (in dot format) of the system. Besides, **cfsm2gg** generates also graphical representation of the communicating machines and the transition systems (again in the `dot` format).

Input formats The syntax of the input files of gmc can be specified either in the *fsa* (after finite state automata) or *cms* format, the latter being a simple process-algebraic syntax (described below). The format to be used depends on the extension of the file name (.fsa or .cms respectively, and for file names without extensions the default format is *fsa*).

A system consists of a list of automata, each described by specifying an (optional) identifier, its initial state, and its transitions. (Identifiers of CFSMs are strings starting with a letter.) We refer to the example in Figure 6.5 to

```
.outputs A
.state graph
q0 1 ! hello q1
q1 1 ! world q2
.marking q0
.end

.outputs
.state graph
q0 0 ? hello q1
q1 0 ? world q2
.marking q0
.end
```



The first automaton has an identifier A while for the second no identifier is specified, so the automaton is identified by 1, its position in the file (automata positions start from 0). The lines following each `.state graph` line yield the transitions followed by the specification of the initial state with the line starting with `.marking`, and finally with the end of the automaton specification (line starting with `.end`). Transitions are written as *src m act msg tgt*, where *src* and *tgt* are respectively the source and target state, *m* is the position of the partner CFSM, *act* is the action (`!` and `?` respectively for output and input actions), and *msg* is the message.

Figure 6.5 HelloWorld example – fsa representation.

describe the *fsa* format. Consider the text on the left of Figure 6.5 specifying the (system consisting of) two simple automata depicted on the right.

It is sometimes more convenient to have a more concrete syntax to represent machines. Therefore we define the alternative *cms* format. The idea is that each CFSM of a system is described by a process in the syntax that we now describe.

The *cms* format of a system is a term of the following grammar:

$$S \quad ::= \quad \mathsf{system}\ \mathsf{id}\ \mathsf{of}\ \mathsf{A}_1, \cdots, \mathsf{A}_n : \mathsf{A}_1 = M_1 \parallel \cdots \parallel \mathsf{A}_m = M_m$$

where id is a string used to identify the system, $\mathsf{A}_1, \cdots, \mathsf{A}_n$ are the names of the machines forming the system, and for each $1 \leqslant i \leqslant m$ (with $m \geqslant n \geqslant 2$) we have a unique defining equation assigning an expression that specifies the behaviour of A_i. We can now model the HelloWorld example of Figure 6.5, as follows:

$$\mathsf{system}\ \mathsf{helloWorld}\ \mathsf{of}\ \mathsf{A}, \mathsf{B} : \mathsf{A} = \cdots \parallel \mathsf{B} = \cdots$$

(where the ellipsis will be defined in a moment). The list of defining equations specify the behaviour M_i of each role A_i, with $1 \leqslant i \leqslant n$, of the system and

the behaviour of some auxiliary machines. For each $1 \leqslant i \leqslant m$, the identity A_i cannot appear in the communication actions of the behaviour M_i of the defining equation $\mathsf{A}_i = M_i$.

Basically, the behaviour of a machine[1] is specified as a regular expression on an alphabet of actions. We impose some syntactic restrictions to keep out some meaningless terms and define:

M	::=	$B+M$	branching	B	::=	pre; end	prefix
pre	::=	A!m	output		|	pre; M	prefix
	|	A?m	input		|	pre do A	iteration

A machine M is a sum of branches B. A branch is a prefix-guarded behaviour (a machine or end) or it is the invocation to the behaviour of a machine A specified in the set of defining equations of the system. Prefixes yield the possible actions of a participant: in A!m (resp. A?m), the message m is sent to (resp. received from) participant A. The equations for the participants of the helloWorld system are:

$$\mathsf{A} = \mathsf{B}!\mathsf{hello};\ \mathsf{B}!\mathsf{world};\ \mathsf{end} \qquad \text{and} \qquad \mathsf{B} = \mathsf{A}?\mathsf{hello};\ \mathsf{A}?\mathsf{world};\ \mathsf{end}$$

Trailing occurrences of end can be omitted, e.g., writing A = B!hello;B!world. Finally, + is right-associative and gives precedence to all the other operators except $\|$, which has the lowest precedence.

6.4 Modelling of an ATM Service

We use a simple scenario to showcase **ChorGram**. We want to design the protocol of a service between an ATM (A), a bank (B), and a customer (C), where, after a successful authentication, the customer C can withdraw cash or check the balance of their account. Such services are enabled only after the ATM has successfully checked the credentials of C. We also require that bank B monitors the usage of the cards of its customers, so that unsuccessful attempts to use it are reported to C (e.g., via an SMS to the customers' mobile).

[1] The *cms* format provides a richer and more flexible syntax which we omit here because not used in the examples. The full syntax is described at `https://bitbucket.org/emlio_tuosto/chorgram/wiki/Home`

6.4.1 ATM Service – Version 1

For the moment, we will assume that the protocol repeats only after a successful withdrawal. Let us start with the description of the bank B:

```
1  B = A ? accessFailed;
2        C ! failedAttempt
3    +                                  .. Notification of authentication outcome
4    A ? accessGranted; (
5        A ? checkBalance;
6        A ! balance
7        +
8        A ? withdraw; (
9            A ! deny
10           +                          .. B decides if to allow the withdrawal
11           A ! allow do B
12       )
13       +
14       A ? quit
15   )
```

The bank B is notified of the outcome of the authentication by the ATM A. If the access fails, B sends a message to the customer C (lines 1–2); otherwise, the bank waits to be told which service has been requested by the customer and acts accordingly (lines 4–14). (The symbol ".." is for single-line comments.)

The specification for the customer C is as follows:

```
1  C = A ! auth; (
2        A ? authPass; (                .. Services are now enabled
3            A ! checkBalance;
4                A ? balance
5            +
6            A ! withdraw do Cw         .. after the request C continues as Cw
7            +
8            A ! quit;
9                A ? card
10       )
11       +
12       A ? authFail;
13           A ? card;
14           B ? failedAttempt
15   )
16 ||
17 Cw = A ? card
18      +
19      A ? money do C
```

Firstly, C provides the ATM A with their credentials by sending the auth message (line 1). If the authentication fails, the ATM replies with the authFail message; in this case the customer also expects their card back and the message failedAttempt from the bank (line 14). On successful authentication, C can select one of the services offered by the ATM or quit the protocol (lines 3–9). In the latter case, C receives their card and terminates (line 9). To check their balance, C sends the message checkBalance to A and waits for the result (line 3). If C sends A the message withdraw, then C continues to Cw (line 6), namely they expects to receive their cash (in which case the protocol restarts) or their card back.

The most complex participant is the ATM A. It can be specified as follows:

```
1  A = C ? auth; (
2          C ! authFail;
3              B ! accessFailed;
4              C ! card
5          +
6          C ! authPass;
7              B ! accessGranted; (
8                  C ? checkBalance do Ac                    .. Ac is specified below
9                  +
10                 C ? withdraw do Aw                        .. Aw is specified below
11                 +
12                 C ? quit;
13                     B ! quit;
14                     C ! card
15             )
16     )
```

The structure of participant A is very similar to the one of C with the addition of the interactions to liaise with the bank. In case A receives the request for a service from C, it will behave according to `Ac` (for checking the balance) or to `Aw` (for withdrawing money). These behaviours are specified below.

```
1  Ac = B ! checkBalance; B ? balance ;C ! balance
2  ||
3  Aw = B ! withdraw; (
4           B ? deny;
5               C ! card
6           +
7           B ? allow;
8               C ! money do A
9       )
```

Auxiliary machine `Ac` forwards the checkBalance message to B, waits for the balance, and returns it to the customer (line 1). Similarly, auxiliary machine `Aw` forwards the request for withdrawal to B, and waits for the outcome (lines 3–8). If the withdrawal is denied (line 4), then the card is returned to the customer, otherwise the customer receives the money and the protocol restarts (line 8).

Executing **ChorGram** on the system

```
1  system atm of C, A, B:
2      C = ··· ||   Cw = ···
3      ||
4      A = ··· ||   Ac = ··· ||   Aw = ···
5      ||
6      B = ···
```

we verify that the system is GMC and the resulting global graph is reported in Figure 6.6, where the overall protocol becomes apparent.

6.4.2 ATM Service – Version 2

The previous specification is GMC, but has several drawbacks, the most evident of which is the fact that when the protocol is repeated the customer

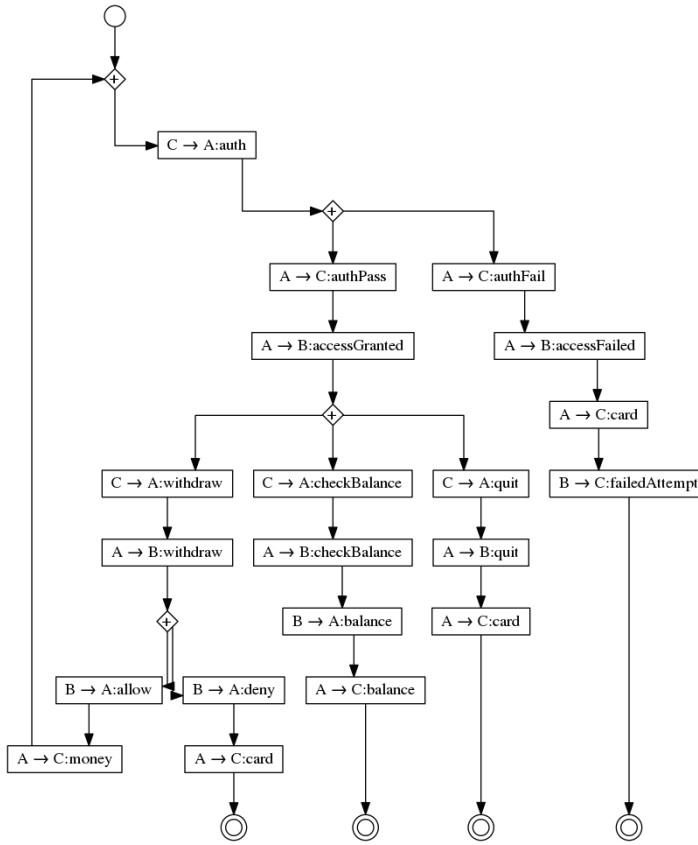


Figure 6.6 Global graph for ATM service v1.

has to re-authenticate. We therefore replace the previous participants C and A with the following ones:

```
C = A!auth;(
      A?authPass do Ca                    .. Now C loops back after authentication
      +
      A?authFail ;A?card; B?failedAttempt
    )
||
Ca = A!checkBalance; A?balance do Cf      .. After successful requests, C decides
     +                                    .. whether to continue or not to behave as Cf
     A!withdraw ;(
           A?card
           +
           A?money do Cf
           )
||
Cf = A!newService do Ca
     +
     A!quit; A?card
```

```
18 ||
19 A = C ? auth; (
20         C ! authPass; B ! accessGranted do Aa
21         +
22         C ! authFail; B ! accessFailed ;C ! card
23       )
24 ||
25 Aa = C ? checkBalance do Ac
26       +
27       C ? withdraw do Aw
28 ||
29 Ac = B ! checkBalance; B ? balance ;C ! balance do Af
30 ||
31 Aw = B ! withdraw; (
32           B ? deny; C ! card
33           +
34           B ? allow ;C ! money do Af
35         )
36 ||
37 Af = (C ? quit; C ! card) +(C ? newService do Ac)
```

Now, after successful authentication, the customer C decides which service to invoke, behaving as specified by `Ca` (lines 7–12). Once the request has been served, the customer executes `Cf` deciding whether to quit or ask for a new service (lines 15–17). Accordingly, A reacts to service requests as per `Aa` on lines 25–27 of the above snippet, similarly to the previous version of ATM, but after the completion of each request, A behaves as per `Af` on line 37 and returns the card to C if a quit message is received or repeats from `Aa` when a new service is requested.

The verification of the new version of the system with **ChorGram** now highlights some problems as shown by the following output message (slightly manipulated for readability):

```
1 ...
2 gmc:    Branching Property (part (ii)): [Rp ([qCf,qAf,qBa],···)]
3                                          (qCf,qAf,C,A,Tau,newService)
4                                          (qCf,qAf,C,A,Tau,quit)   No choice awareness
5 ...
```

The above message reports that a reachable configuration where participants C, A, and B respectively are in state `qCf`, `qAf`, and `qBa` is a 'No choice awareness' configuration. This configuration is highlighted in yellow in the synchronous transition system, which is reported in Figure 6.7. Inspecting the synchronous transition system, we note that this configuration leads to deadlocks (the configurations highlighted in orange in Figure 6.7), due to the fact that the participant B is not notified when the quit branch is taken, i.e., B is *not aware* of which branch of the protocol was chosen by C.

Notice that **ChorGram** builds a global graph also when the system violates GMC (not shown for space restrictions). Such a synthesised global graph reflects some of their possible communication sound executions while leaving out traces where communication misbehaviour happen. The global graph of

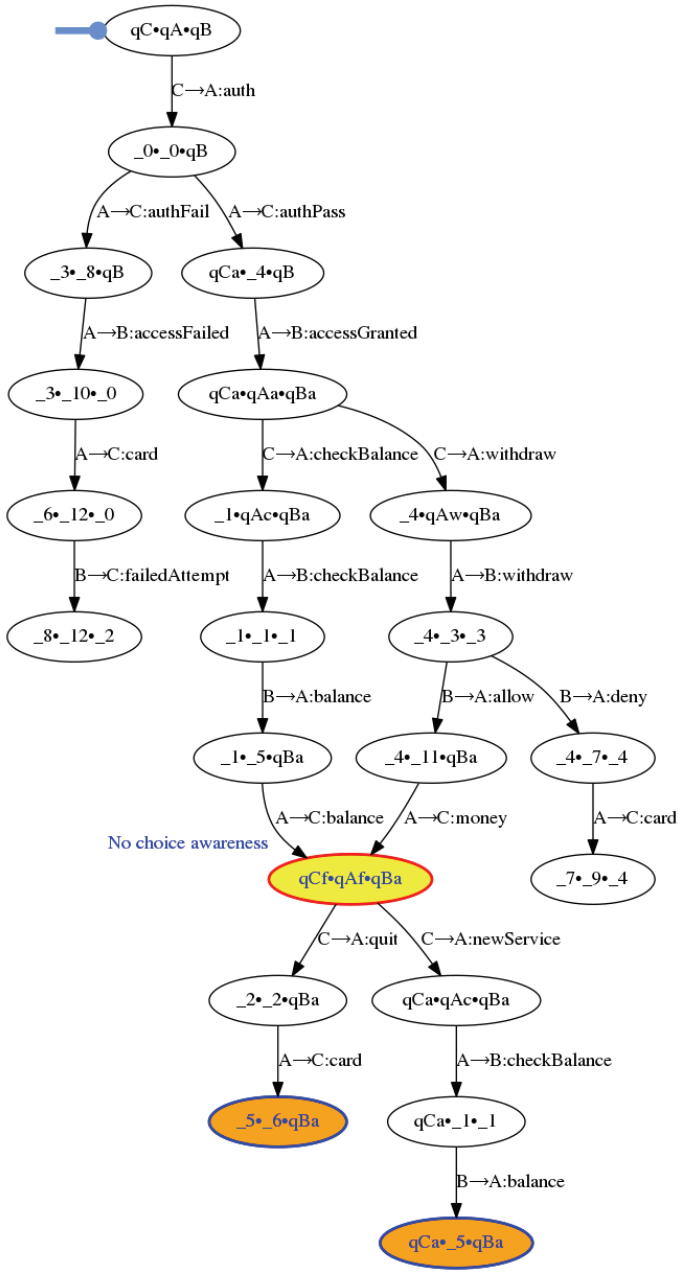


Figure 6.7 Synchronous transition system of ATM service v2.

our second version of the ATM system can also be used to understand what goes wrong in the overall choreography.

6.4.3 ATM Service – Version 3 (fixed)

Besides making the refined specification of Section 6.4.2 GMC, in the next version we also want to let the customer quit the protocol immediately after the authentication. This change makes `Cf` and `Af` unnecessary: so, we replace `Ca` and `Cf` with the following new versions:

```
1  Ca = A ! checkBalance; A ? balance do Ca
2       +
3       A ! withdraw; (
4                      A ? card
5                      +
6                      A ? money do Ca
7                     )
8       +
9       A ! quit
10 ||
11 Aa = C ? checkBalance do Ac
12      +
13      C ? withdraw do Aw
14      +
15      C ? quit;
16          B ! quit
```

Note that now A notifies B when the protocol quits (line 15). This modification requires also to modify the bank, which is now:

```
1  B = A ? accessFailed; C !failedAttempt      .. The bank tells the customer the attempt failed
2      +
3      A ? accessGranted do Ba
4  ||
5  Ba = A ? checkBalance; A ! balance do Ba
6       +
7       A ? withdraw; (
8            A ! deny
9            +
10           A ! allow do Ba
11        )
12      +
13      A ? quit
```

The above changes re-establish GMC, hence communication soundness of the system, as verified by **ChorGram**, which returns the global graph of Figure 6.8.

6.5 Conclusions and Related Work

Conclusions & future work We presented **ChorGram**, a tool supporting the analysis and design of choreography-based development. We have discussed only part of the features of **ChorGram**, those strictly related to

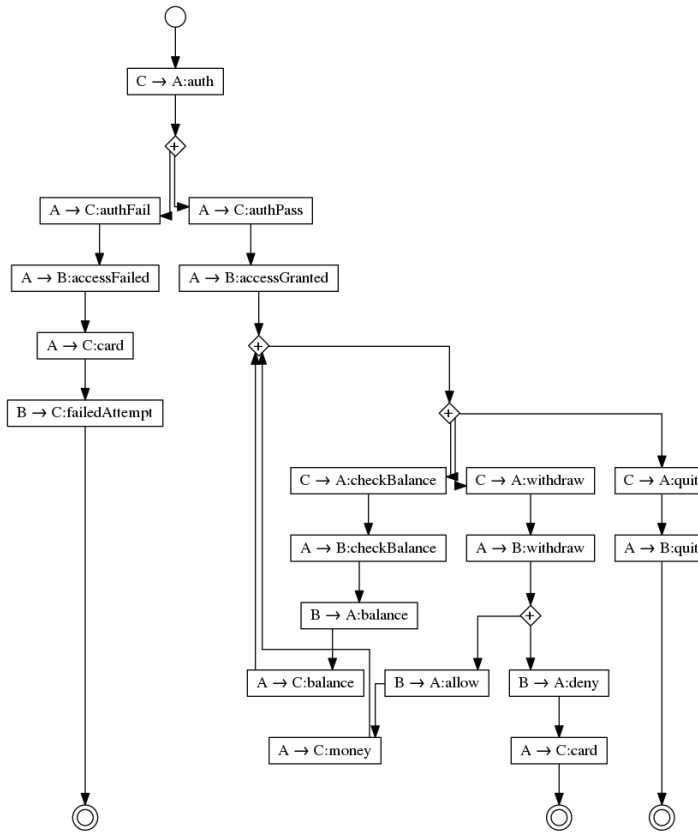


Figure 6.8 Global graph for ATM service v3.

the bottom-up development based on our theory [17], which is itself an extension of previous work on synthesising global types from local specifications [9, 13, 15]. Recently, **ChorGram** has been extended with new functionalities for top-down development. These new functionalities rely on a new semantic framework [11]. We are also planning to plug the "bottom-up" approach advocated here with the classical "top-down" approach [12] as advocated by, e.g., the Scribble specification language [22, 25]. Such an integration would give the flexibility of designing protocols at the global level and obtain the local level automatically, and vice-versa.

As illustrated in Section 6.4, our approach can be used to give feedback to protocol designers. Hence, we are considering integrating **ChorGram** with

a framework [20] allowing programmers to obtain real-time feedback wrt. the multiparty compatibility of the system they are designing. Currently, the prototype highlights communication mismatches at the local level and it is sometimes difficult to identify the real cause of such errors [20]. However, it appears that a (possibly partial) global graph can help giving precise feedback to the developer so that they can fix the error(s) easily.

Existing extensions and applications Recent work extends the theory underlying **ChorGram** [17] to communicating timed automata (CTA) [5], i.e., CFSMs which use clocks to constrain when send and receive actions may take place. The authors show that if a system validates some conditions on communication soundness and deadlines, it is possible to construct a choreography with time constraints which is equivalent to the original system of CTAs.

The synthesis of global graphs from local specifications has been applied thus far in two programming languages. A tool to statically detect deadlocks in the Go programming language is available [18]. The tool first extracts CFSMs from each Go-routine in the source code, then feeds them into a slight variation of **ChorGram** (for synchronous semantics) which checks whether the system is multiparty compatible and generates the corresponding global graph (which may be used to track down what may have caused deadlocks). Also, **ChorGram** has been used to model and analyse **genserver** [23], a part of the Erlang OTP standard library widely used in the Erlang community for the development of client/server applications. The analysis highlighted possible coordination errors and was conducted following the pattern showed in Section 6.4. The main difference was that the participants and the corresponding CFSMs had to be extracted from the API documentation of **genserver**.

An interesting use [2, 14] of multiparty compatibility is to support an orchestration mechanism based on the *agreement* of behavioural contracts [4]. Recently this theoretical framework has been used to develop **Diogenes** [1], a middleware supporting designers (and developers) to write *honest* programs [3], namely programs that respect *all* their contracts in *all* their execution contexts. An interesting future work is to integrate **Diogenes** and **ChorGram** in order to adapt components when they are not multiparty compatible. In such cases, (as discussed at the end of Section 6.4.2) **ChorGram** synthesises a choreography which, although not faithfully reflecting the behaviour of participants, represents some of their possible communication sound executions. Such a synthesised choreography could then be used to obtain projections that help to attain honesty.

Acknowledgements This work is partially supported by EU FP7 612985 (UPSCALE) and by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1.

References

[1] Nicola Atzei and Massimo Bartoletti. Developing honest Java programs with Diogenes. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 52–61, 2016.

[2] Massimo Bartoletti, Julien Lange, Alceste Scalas, and Roberto Zunino. Choreographies in the wild. *Sci. Comput. Program.*, 109:36–60, 2015.

[3] Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. Honesty by typing. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 305–320. Springer, 2013.

[4] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. Contract-oriented computing in CO_2. *Scientific Annals in Comp. Sci.*, 22(1):5–60, 2012.

[5] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR 2015*, pages 283–296, 2015.

[6] Filippo Bonchi and Damien Pous. HKC. `http://perso.ens-lyon.fr/damien.pous/hknt/`

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

[8] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Petrify. `http://www.lsi.upc.edu/~jordicf/petrify/`

[9] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP 2013*.

[10] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP 2012*, pages 194–213, 2012.

[11] Roberto Guanciale and Emilio Tuosto. An Abstract Semantics of the Global View of Choreographies. In *ICE 2016*, pages 67–82, 2016.

[12] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[13] Julien Lange. *On the Synthesis of Choreographies*. PhD thesis, Department of Computer Science, University of Leicester, 2013.

[14] Julien Lange and Alceste Scalas. Choreography synthesis as contract agreement. In *ICE*, volume 131 of *EPTCS*, pages 52–67, 2013.
[15] Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In *CONCUR 2012*, pages 225–239, 2012.
[16] Julien Lange and Emilio Tuosto. **ChorGram**: tool support for choreographic development. Available at `https://bitbucket.org/emlio_tuosto/chorgram/wiki/Home`, 2015.
[17] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232, 2015.
[18] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *CC 2016*, pages 174–184, 2016.
[19] Object Management Group. Business Process Model and Notation. `http://www.bpmn.org`
[20] Roly Perera, Julien Lange, and Simon J. Gay. Multiparty compatibility for concurrent objects. In *PLACES 2016*, pages 73–82, 2016.
[21] Chris Piro. Chat stability and scalability. `https://goo.gl/Z1tpgA`
[22] Scribble. `http://www.scribble.org`
[23] Ramsay Taylor, Emilio Tuosto, Neil Walkinshaw, and John Derrick. Choreography-based analysis of distributed message passing programs. In *PDP 2016*, pages 512–519, 2016.
[24] Paolo D'Incau's blog. `https://goo.gl/eXKng1`, 2013.
[25] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC 2013*, pages 22–41, 2013.