Choreographies for Program Understanding

Gabriele Genovese[1,2], Ivan Lanese[3(✉)], Cinzia Di Giusto[2], Emilio Tuosto[4], and Germán Vidal[5]

[1] University of Bologna, Bologna, Italy
[2] Université Côte d'Azur, CNRS, I3S, Nice, France
[3] Olas Team, University of Bologna & Inria - Université Côte d'Azur, Bologna, Italy
ivan.lanese@gmail.com
[4] Gran Sasso Science Institute, L'Aquila, Italy
[5] VRAIN, Universitat Politècnica de València, Valencia, Spain

Abstract. Current choreography-based approaches to the specification and implementation of distributed systems lack support when it comes to program understanding. In particular, we miss systematic methodologies and algorithms to take a message-passing program written in a mainstream programming language and automatically produce a global description of all its communication behaviors. This helps understanding the program interaction patterns and also highlights possible unexpected behaviors to support debugging. We discuss the requirements and difficulties of the approach we envisage. Through concrete examples we outline the kind of global descriptions we want to obtain.

1 Context and Vision

The impetus of service-oriented computing and, more recently, of microservices has determined a radical shift from monolithic applications to distributed components cooperating through message-passing. In this domain, *choreographic* coordination [17] allows designers to focus on the so-called *application-level protocols*, i.e., a description of the *communication interactions*[1] among the system's components (hereafter called *participants*). This focus is typically expressed by two distinct, yet related views of distributed computations: the so-called *global* and *local* views. The former view abstracts away from the actual communication

This work has been partially supported by French ANR project SmartCloud ANR-23-CE25-0012, by INdAM - GNCS 2024 project MARVEL, code CUP E53C23001670001, by the project FREEDA (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU, by the PRIN PNRR project DeLICE (F53D23009130001), by the MUR dipartimento di eccellenza 2023-2027, and by *Generalitat Valenciana* under grant CIPROM/2022/6 (FassLow).

[1] With interaction, we refer to a full message-passing communication, comprised of both a send of a message and the corresponding receive.

Published by Springer Nature Switzerland AG 2025
C. Ferreira and C. A. Mezzina (Eds.): FORTE 2025, LNCS 15732, pp. 173–181, 2025.
https://doi.org/10.1007/978-3-031-95497-9_10

infrastructure in order to give a blueprint of the communication protocol. The latter view provides the description of the communication behavior of participants in isolation and can guide their implementation. Choreographies can be expressed in modelling languages or formalisms (like WS-CDL [17], BPMN diagrams [26]), multiparty session types [15], message-sequence charts [1,22,28], multiparty contracts [4,5,8,18], and many others (see also the survey in [16]).

Besides offering a development methodology[2] for message-passing systems, global views yield a high-level description of application-level protocols. It is therefore crucial that global views faithfully capture all the interactions in the system. While this is relatively simple when participants' implementations are driven by the global view (e.g., in the top-down approach), the correspondence can be easily spoiled when software evolves or dynamic composition takes place (as advocated in microservice architectures). The classical top-down approach is then of little help: one needs to write a global description of the desired behavior and then use type checking or monitoring to find possible discrepancies.

In this paper, we advocate bottom-up approaches that "extract" global views from code to understand the actual, possibly bugged, behavior of the program. Note that bottom-up approaches exist [9,11,19,20,25], but they have several limitations. Firstly, they produce global views out of abstract models of local views (such as communicating-finite state machines [7] or some kind of behavioral types [16]) and not from actual code written in a mainstream programming language. Secondly, the extracted global views do faithfully capture the behavior of a local view only under some "well-formedness" conditions (e.g., multiparty compatibility in [20]). Parts of the choreography that do not respect the conditions are just dropped, hiding buggy or unexpected behavior that they may contain. We claim that this is exactly the case where a global view would be most useful: the program is buggy and, in order to fix the bug, we need to understand the application-level protocol via a global abstract description.

We claim that the full potential of choreographic approaches in building tools for program understanding can only be unleashed if top-down and bottom-up techniques are married. In particular, we advocate the need of developing bottom-up debugging tools to support the work of programmers.

2 Requirements & Difficulties

Let us discuss a few requirements that we deem decisive to realise our vision.

Push-Button Technique: the extraction of the choreography should be fully automatic, to be applicable to existing code without the need to add special annotations or any other input from the programmer.

Always Capture the Good Behaviors: even if the system is not well-behaved, hence its behavior can not be described by a choreography in the classical sense

[2] According to the so-called top-down approach, a global view (formalised, e.g., as a multiparty session type) can be algorithmically projected on a local view preserving relevant properties.

(since choreographies enforce properties such as race and deadlock freedom), a choreography should be extracted. It should contain at least all the good behaviors.

Highlight Misbehaviors: debugging is our key reason to extract choreographies from code; therefore, choreographies for ill-behaved systems, beyond describing the correct behaviors as mentioned above, should flag, as much as possible, misbehaviors due to communications such as deadlocks, orphan messages, unspecified receptions, etc.

Applicable to Mainstream Languages: one should be able to extract the choreography from a real program written in a mainstream language. Natural targets are languages with a clean concurrency/distribution model and dedicated primitives for message-passing such as Erlang, Go, and Scala.

Support Creation and Termination of Participants: in real message-passing systems new processes can be spawned, and some processes may terminate. Hence, a choreographic description should allow for a dynamic number of participants.

Support Races: often choreographic models do not admit races, yet races are commonplace in concurrent programs. Therefore, races should not be forbidden, but possibly flagged as potentially wrong in line with the requirement of highlighting misbehaviors, since they could lead to unexpected behavior.

Accessible yet Precise Notation: choreographies should be represented with an intuitive, possibly graphical, formalism to improve readability. Instead of the usual algebraic formalisms one should appeal to graph-like notations such as labelled transition systems or finite state automata that pair a graphical representation with a well-defined mathematical definition.

We have given above a number of requirements that the approach we envisage should satisfy, but the reader familiar with the topic may have already found a number of potential difficulties. Indeed, we are aware of a few problems that need to be solved in order to make such an approach feasible. We describe them below, together with possible mitigation measures:

Undecidability: extracting a precise description of all the behaviors of a system is in general impossible, since it would require, e.g., deciding termination. Hence, one can focus on extracting a precise choreography in simple cases and giving approximations of the behavior otherwise (e.g., when the system is infinite state). An over-approximation may exhibit spurious behavior w.r.t. the actual behavior of the system. Thus, one can understand the actual behavior, including bugs; however, it is necessary to verify if the reported bugs are false positives. Therefore, care should be taken to limit the number of false positives, since a too high number would make the approach not viable. Over-approximations can be too coarse; e.g., if it is not possible to statically determine which is the expected recipient of a message, an over-approximation may yield a huge number of spurious communications by adding an interaction for each participant in the system. When, as in the case discussed above, over-approximations are not suitable, an under-approximation may be more useful,

possibly paired with warnings highlighting issues. The problem in this case is to ensure that false negatives, i.e., cutting off misbehavior from extracted choreographies, are avoided.

Huge descriptions: choreographies of real programs may be huge, thus hindering their usefulness for program understanding. We believe this issue should be tackled by providing tools to abstract, explore, or better visualize the choreography. For instance, one may decide that in order to understand a particular behavior interaction, some participants are immaterial, hence they can be abstracted away (e.g., like ϵ-transitions in automata based approaches). Another option to reduce the size of the description could be to collapse behaviors which are equal up to swap of concurrent actions (as in partial order reduction techniques within model checking [14]), or collapse the behaviors of multiple processes executing the same code.

3 Motivating Examples

In order to better illustrate our point, we consider two simple examples. We use Erlang-like pseudocode for which we provide a global view in terms of choreography automata [3], a formal model of global views featuring a graphical representation akin to finite-state automata. Here, a *global view* is an abstract description of all the possible behaviors of the full system. We consider a language where receive statements are blocking operations. A state where each participant has completed its task and terminated is called *final*. As such, a state that is not final and has no outgoing transitions is a *deadlock*.

The first example is a concise reproduction of the dining philosophers problem, which highlights a possible deadlock. The second example shows a possible mutual exclusion error when operating a simple bank account. Both examples are available online [12]; the corresponding choreography automata have been obtained with the help of our prototype tool Chorer [10].

In the examples, we exploit two operations for sending and receiving messages, respectively. More precisely, send msg to proc sends message msg to process proc, while receive pat_1 from $\mathsf{proc}_1 \rightarrow \mathsf{e}_1$; ... ; pat_n from $\mathsf{proc}_n \rightarrow \mathsf{e}_n$ represents a branching point where the process receives the first message that matches a pattern pat_i and continues with the execution of e_i. As in Erlang, pattern matching is tried from top to bottom. When a receive has only one clause, we abbreviate it as receive pat from proc (and continues with the execution of the next sentence).

3.1 Dining Philosophers Example

Let us consider a program with two participants playing the role of a dining philosopher who shares two forks with the other participant. The behavior of the philosophers is given by the pseudocode on the right while pseudocode for the behavior of the forks is discussed below. Each philosopher first acquires the forks (starting with the one on its right, that is the one with the same index), then eats (with the function call `eat()` representing some terminating local computation), and finally releases the forks before recurring.

```
philosopher(Fork1, Fork2) →
  send req to Fork1,
  receive ack from Fork1,
  send req to Fork2,
  receive ack from Fork2,
  eat(),
  send release to Fork1,
  send release to Fork2,
  philosopher(Fork1, Fork2).
```

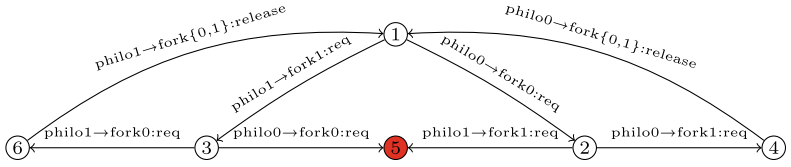

Fig. 1. Global view of the dining philosophers example.

```
fork() →
  receive req from Phil,
  send ack to Phil,
  receive release from Phil,
  fork().
```

Parameters Fork1 and Fork2 are references to processes executing the behavior of forks described by the pseudocode on the left that repeatedly waits for the request from process Phil, ack the request, and waits for the release message from Phil.

There are two possible behaviors of the system. The first (good) one where the philosophers alternate eating infinitely and a second (bad) behavior where both philosophers manage to take only one fork each resulting in a deadlock. Figure 1 depicts the global Choreography Automaton representing the program above. We have two recursive executions where both philosophers eat, represented as loops, and two executions which end in the same deadlock state. The deadlock is visible since State 5 is not final, and it has no outgoing transitions. The automaton is simplified as it does not display the `ack` messages, and the `release` messages are merged since their order is irrelevant. The focus here is solely on the order of the `req` messages. One can imagine to first extract the full choreography automaton, and then merge equivalent behaviors and abstract away from uninteresting transitions.

3.2 Bank Account Example

We now consider a system where a bank account is accessed by two clients, dubbed C1 and C2.

The pseudocode on the right yields the behavior of the bank account, where Value represents its current balance. This process waits for requests from a client. A request can either be a read access to know the current balance or an update request of such value to a NewValue.

```
account(Value) →
  receive
    read from Client →
      send Value to Client,
      account(Value);
    NewValue from Client →
      account(NewValue).
```

Symmetrically, client processes C1 and C2 behave according to the pseudocode on the left: the process reads the current balance from the account, performs some internal operations based on such value, and updates the balance. The global view of the communicating system is depicted in Fig. 2. We can observe two correct executions where the operations are performed in a read-update-read-update order (taking the path via states 1-2-4-7-11 or the one via states 1-3-6-10-14). However, there is also a read-read-update-update order on the highlighted paths. Although the choreography is not inherently incorrect, these highlighted paths could represent a violation of mutual exclusion which may be undesirable for developers in certain contexts. The choreography automaton in Fig. 2 helps in spotting this issue.

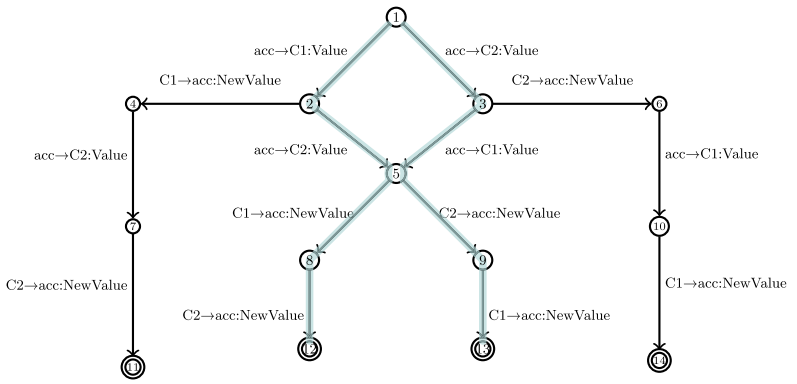

Fig. 2. Global view of the bank account example.

```
client() →
  send read to Acc,
  receive Value from Acc,
  send NewValue to Acc.
```

As for the Dining Philosophers example, the automaton is simplified as we omit the read requests since they are not relevant to spot the undesired behavior (but may be needed for further analysis of its causes).

4 Related Work and Conclusion

Software industry is increasingly devoting attention to choreographic approaches [2,6,13,26] because they naturally support modularization and decoupling. In fact, distributed components coordinate according to a global description without the need of an explicit coordinator. In this context, global specifications are crucial for guaranteeing correctness (since they are blueprints of complex distributed systems and feature model-driven development) as well as for program understanding. To the best of our knowledge, the first attempts to distil global specifications for message-passing systems goes back to [19,21,25]. These approaches aim to identify "meaningful" global specifications according to general properties such as deadlock-freedom or absence of orphan messages [7]. A limitation of these approaches is that they do not start from components written in a full-fledged programming language. Rather, distributed components are specified in [19,21] as abstract models (respectively, π-calculus processes [23,24,27] and communicating finite-state machines [7]).

In this paper we have discussed the use of choreographies for program understanding, highlighting both the requirements of such an approach and contrasting them with the more common approaches we are aware of. We have also discussed challenges to be overcome and shown some examples to make our discussion more concrete.

Tool support is paramount to successfully employ choreographies for program understanding. Indeed, we are working on a tool, called Chorer, to extract choreographies from programs written in the Erlang programming language [10], which follows the ideas outlined above.

Chorer. The tool is currently under active development. At this stage, it is capable of parsing relatively simple Erlang programs, such as those presented in Sect. 3. It exploits *static analysis* techniques to construct a best-effort global view of the system, formalized as an extension of choreography automata [3]. This is achieved through a bottom-up approach, as discussed in detail in Sect. 2. Chorer can be used via a Command Line Interface. Upon execution, the tool generates automata that represent both local and global behaviors. These automata are represented in Graphviz's DOT language format [29], facilitating their easy visualization. Chorer is also able to identify and highlight potential deadlock states. However, the correctness of these results has not yet been formally proven, and further verification steps are required to establish the overall soundness of the approach.

Chorer relies on an over-approximation algorithm. This technique should preserve correct behaviors in the analyzed system but may also introduce spurious behaviors that would not occur in actual executions. Over-approximation is required for instance when it is not possible to decide statically whether a given message matches a given pattern (currently, approximation on values of variables is quite rough): the algorithm then considers both the options of matching succeeding and failing, while in practice maybe only one of them can actually happen. Creation and termination of participants are analyzed statically. The

tool is currently unable to analyze programs that dynamically spawn actors within recursive functions, since this may result in an infinite state space.

Despite these constraints, the tool satisfies the majority of the requirements outlined in Sect. 2, e.g., it considers (a subset of) the mainstream language Erlang, it is a push-button technique, and it is based on the accessible yet precise notion of finite state automata. Beyond improving the coverage of the language construct and patterns, a main item of future work is studying abstraction techniques to make the resulting choreography automata easier to understand.

Acknowledgements. We thank the anonymous reviewers for their useful comments and suggestions.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theor. Comput. Sci. **331**(1), 97–114 (2005)
2. Autili, M., Inverardi, P., Tivoli, M.: Automated synthesis of service choreographies. IEEE Softw. **32**(1), 50–57 (2015)
3. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 86–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_6
4. Bartoletti, M., Tuosto, E., Zunino, R.: Contracts in distributed systems. In: Silva, A., Bliudze, S., Bruni, R., Carbone, M., (eds.) ICE, vol. 59, 130–147 (2011)
5. Bartoletti, M., Tuosto, E., Zunino, R.: On the realizability of contracts in dishonest systems. In: Sirjani, M., (ed.) COORDINATION, pp. 245–260 (2012)
6. Bonér, J.: Reactive Microsystems - The Evolution Of Microservices At Scale. O'Reilly, Sebastopol (2018)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
8. Bravetti, M., Zavattaro, G.: A theory of contracts for strong service compliance. Math. Struct. Comput. Sci. **19**(3), 601–638 (2009)
9. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distri. Comput. **31**, 51–67 (2018)
10. Chorer github repository. https://github.com/gabrielegenovese/chorer
11. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 424–440. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_25
12. Two simple examples. Dining example and Account example, https://github.com/gabrielegenovese/chorer/tree/master/examples/dining. https://github.com/gabrielegenovese/chorer/tree/master/examples/account
13. Frittelli, L., Maldonado, F., Melgratti, H., Tuosto, E.: A choreography-driven approach to APIs: the OpenDXL case study. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 107–124. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_7
14. Godefroid, P.: Model checking for programming languages using verisoft. In: POPL, pp. 174–186 (1997)

15. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1-9:67 (2016)
16. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1-3:36 (2016)
17. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web services choreography description language version 1.0. Technical report, W3C (2005). http://www.w3.org/TR/ws-cdl-10/
18. Lange, J., Scalas, A.: Choreography synthesis as contract agreement. In: Proceedings 6th Interaction and Concurrency Experience, ICE 2013, pp. 52–67 (2013)
19. Lange, J., Tuosto, E.: Synthesising choreographies from local session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 225–239. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_17
20. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: Rajamani, S.K., Walker, D., (eds.) POPL, pp. 221–232. ACM (2015)
21. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL15, pp. 221–232 (2015)
22. Lohrey, M.: Realizability of high-level message sequence charts: closing the gaps. Theor. Comput. Sci. **309**(1–3), 529–554 (2003)
23. Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge University Press, Cambridge (1999)
24. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Inf. Comp. **100**(1), 1–40 (1992)
25. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_23
26. OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011. https://www.omg.org/spec/BPMN
27. Sangiorgi, D., Walker, D.: The π-Calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2002)
28. Stutz, F.: Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In: Ali, K., Salvaneschi, G., (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States, vol. 263, LIPIcs, pp. 32:1–32:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
29. The Graphviz Authors. Dot format. https://graphviz.org/doc/info/lang.html