From Infinity to Choreographies
Extraction for Unbounded Systems[⋆]

Bjørn Angel Kjær, Luís Cruz-Filipe[0000−0002−7866−7484], and Fabrizio Montesi[0000−0003−4666−901X]

Department of Mathematics and Computer Science, University of Southern Denmark
Campusvej 55, 5230 Odense M, Denmark
`bjoernak@gmail.com,{lcfilipe,fmontesi}@imada.sdu.dk`

Abstract. Choreographies are formal descriptions of distributed systems, which focus on the way in which participants communicate. While they are useful for analysing protocols, in practice systems are written directly by specifying each participant's behaviour. This created the need for *choreography extraction*: the process of obtaining a choreography that faithfully describes the collective behaviour of all participants in a distributed protocol.

Previous works have addressed this problem for systems with a predefined, finite number of participants. In this work, we show how to extract choreographies from system descriptions where the total number of participants is unknown and unbounded, due to the ability of spawning new processes at runtime. This extension is challenging, since previous algorithms relied heavily on the set of possible states of the network during execution being finite.

Keywords: Choreography · Extraction · Concurrency · Message passing

1 Introduction

Choreographies are coordination plans for concurrent and distributed systems, which describe the expected interactions that system participants should enact [5,14]. Languages for expressing choreographies (choreographic languages) are widely used for documentation and specification purposes, some notable examples being Message Sequence Charts [7], UML Sequence Diagrams [16], and choreographies in the Business Process Modelling Notation (BPMN) [15]. More recently, such languages have also been used for programming and verification, e.g., as in choreographic programming [13] and multiparty session types [6] respectively.

In practice, many system implementations do not come with a choreography yet. *Choreography extraction* (extraction for short) is the synthesis of a choreography that faithfully represents the specification of a system based on message passing (if it exists) [1,3,11,12]. Extraction is helpful because it gives developers

[⋆] Partially supported by Villum Fonden, grant no. 29518.

a global overview of how the *processes* (abstractions of endpoints) of a system interact, making it easier to check that they collaborate as intended. In general, though, it is undecidable whether such a choreography exists, making extraction a challenging problem.

Current methods for extraction cannot analyse systems that spawn new processes at runtime: they can only deal with systems where the number of participants is finite and statically known. This is an important limitation, because many modern distributed systems dynamically create processes for several reasons, such as scalability. The aim of this article is to address this shortcoming.

As an example of a system that cannot be analysed by previous work, consider a simple implementation of a serverless architecture (Example 1).

Example 1. A simple serverless architecture, written in pseudocode (we will formalise this example later in the article). A client sends a request to an entrypoint, which then spawns a temporary process to handle the client. The spawned process computes the response to the request. Then it offers the client to make another request, in which case a new process is spawned to handle that, since the new request may require a different service.

client:
```
    send init to server;
    loop {
        server presents worker; receive result from worker;
        if finished
           then request termination from worker; terminate;
           else request next from worker; server:=worker;
    }
```

server:
```
    receive init from client; Handle(server);

    procedure Handle(parent) {
        spawn worker with {
            parent presents client; ComputeResult();
            send result to client; receive request from client;
            switch request {
                next: Handle(worker);
                termination: terminate;
            }
        }
        introduce worker and client; terminate;
    }
```

Example 1 illustrates the usefulness of extraction: a human could manually go through the code and check that the two processes will interact as intended; however, despite it being a greatly simplified example, it is not immediately obvious whether the processes will communicate correctly or not.

Previous methods for extraction use graphs to represent the possible (symbolic) execution space of the system under consideration. The challenge presented by code as in Example 1 is that these graphs are not guaranteed to be finite anymore, because of the possibility of spawning new processes at runtime.

In this article we introduce the first method for extracting choreographies that supports process spawning, i.e., the capability of creating new processes at runtime. Our main contribution consists of a theory and implementation of extraction that use name substitutions to obtain finite representations of infinite symbolic execution graphs. Systems with process spawning have a dynamic topology, which further complicates extraction: new processes can appear at runtime, and can then be connected to other processes to enable communication. We extend the languages used for extraction in previous work with primitives for capturing these features, and show that our method can deal with them.

Structure of the paper. In Section 2, we recap the basic theory of extraction. In Section 3, we introduce the languages for representing systems (as networks of processes) and choreographies. Section 4 reports our method for extracting networks with an unbounded number of processes (due to spawning), its implementation and limitations. We conclude in Section 5.

Related work. We have already mentioned most of the relevant related work in this section. Choreography extraction has been explored for languages that include internal computation [3], process terms that correspond to proofs in linear logic [1], and session types (abstract terms without internal computation) [11,12]. Our method deals with the first case (the most general among those cited). Our primitives for modelling process spawning are inspired by [4].

2 Background

This section summarizes the framework for choreography extraction that we extend [2,3,8,17]. The remainder of the article expands upon this work to extend the capabilities of extraction, and to bring it closer to real systems.

2.1 Networks

Distributed systems are modelled as *networks*, which consist of several participants executing in parallel. Each participant is called a *process*, and the sequence of actions it executes is called a *behaviour*. Each process also includes a set of *procedures*, consisting of a name and associated behaviour.

Behaviours are formally defined by the grammar given below.

$$\begin{aligned} B ::= {} & \mathbf{0} \mid X \mid \mathsf{p}!m; B \mid \mathsf{p}?; B \mid \mathsf{p} \oplus \ell; B \mid \mathsf{p}\&\{\ell_1 : B_1, \ldots, \ell_n : B_n\} \\ & \mid \texttt{if } e \texttt{ then } B_1 \texttt{ else } B_2 \end{aligned}$$

Term $\mathbf{0}$ designates a terminated process. Term X invokes the procedure named X. Invoking a procedure must be the last action on a behaviour – in other words, we only allow tail recursion. Term $\mathsf{p}!m; B$ describes a behaviour where the executing process evaluates message m and sends the corresponding value to process p, then continues as B. The dual term $\mathsf{p}?; B$ describes receiving a message from p, storing it locally and continuing as B. Behaviour $\mathsf{p} \oplus \ell; B$ sends the selection of label ℓ to process p then continues as B, while the dual $\mathsf{p}\&\{\ell_1 : B_1, \ldots, \ell_n : B_n\}$ offers the behaviours $B_1, \ldots, B_n$ to p, which can be selected by the corresponding labels $\ell_1, \ldots, \ell_n$. Finally, `if` e `then` B_1 `else` B_2 is the conditional term: if the Boolean expression e evaluates to true, it continues as B_1, otherwise, it continues as B_2.

The syntax

```
p {
    def X_1 { B_1 }
    ⋮
    def X_n { B_n }
    main { B }
}
```

describes a process named p with local procedures $X_1, \ldots, X_n$ defined respectively as $B_1, \ldots, B_n$, intending to execute behaviour B.

A network is specified as a sequence of processes, all with distinct names, separated by vertical bars (|). Example 2 defines a valid network, which we use as running example throughout this section to explain the existing extraction algorithm.

Example 2. This network describes the protocol for an online store. The customer sends in items to purchase, then asks the store to proceed to checkout, or continue browsing.

Once the customer proceeds to checkout, they send their payment information to the store. The store then verifies that information, and either completes the transaction, or asks the client to re-send payment information if there where a problem.

```
customer {
    def browse{ store!item; if checkout
        then store⊕buy; purchase;
        else store⊕more; browse }
    def purchase{ store!payment; store&{accept: 0, reject: purchase} }
    main{ browse }
} |
store {
    def offer{ customer?; customer&{buy: payment, more: offer} }
    def payment{ customer?; if accepted
        then customer⊕accept; 0
        else customer⊕reject; payment }
    main { offer }
}
```

2.2 Choreographies

Global descriptions of distributed systems, specifying interactions between participants rather than their individual actions, are called *choreographies.* Similar to processes in networks, a choreography contains a set of procedure definitions, and a main body. The terms of choreography bodies are defined by the grammar below and closely correspond to the actions in process behaviours.

$$C ::= \mathbf{0} \mid X \mid \mathsf{p}.m \,\texttt{->}\, \mathsf{q}; C \mid \mathsf{p} \,\texttt{->}\, \mathsf{q}[\ell]; C \mid \texttt{if } \mathsf{p}.e \texttt{ then } C_1 \texttt{ else } C_2$$

Term $\mathbf{0}$ denotes a choreography body where all processes are terminated. Term X invokes the procedure with name X. In the communication $\mathsf{p}.m \texttt{->} \mathsf{q}; C$, process p sends message m to q, which stores the result, and the system continues as described by choreography body C. Likewise, in label selection $\mathsf{p} \texttt{->} \mathsf{q}[\ell]; C$, process p selects an action in q by sending the label ℓ, and the system continues as C. In the conditional `if` $\mathsf{p}.e$ `then` C_1 `else` C_2, process p starts by evaluating the Boolean expression e; if this resolves to true, then the choreography continues as C_1, otherwise it continues as C_2.

Example 3. The protocol described by the network in Example 2 can be written as the following choreography.

```
def Buy {
    customer.item -> store; if customer.checkout
        then customer -> store[buy]; Pay
        else customer -> store[more]; Buy
}
def Pay {
    customer.payment -> store; if store.accepted
        then store -> customer[accept]; 0
        else store -> customer[reject]; Pay
}
main {Buy}
```

2.3 Extraction algorithm

The extraction algorithm from [2,3] consists of two steps. The first step is building a graph that represents a symbolic execution of the network. The second is to traverse this graph, using its edges to build the extracted choreography.

Graph generation. The first step in extracting a choreography from a network is building a *Symbolic Execution Graph* (SEG) from the network. A SEG is a directed graph representing an abstraction of the possible evolutions of the network over time. It abstracts from the concrete semantics by ignoring the concrete values being communicated and considering both possible outcomes for every conditional. Nodes contain possible states of the network, and edges connect nodes that are related by execution of one action (the label of the edge).[1]

Edges in the SEG are labeled by *transition labels*, which represent the possible actions executable by the network: value communications (matching a send action with the corresponding receive), label selection (matching selection and offer), and conditionals. For the last there are two labels, representing the two possible outcomes (the "then" and "else" branch, respectively).

$$\lambda ::= \mathsf{p}.e \texttt{->} \mathsf{q} \mid \mathsf{p} \texttt{->} \mathsf{q}[\ell] \mid \mathsf{p}.e\ \texttt{then} \mid \mathsf{p}.e\ \texttt{else}$$

As an example, we show how to build the SEG for the network in Example 2 (see Fig. 1). The main behaviours of the two processes are procedure invocations (node on top). Expanding the corresponding definitions, we find out that the first action by customer is sending to store, while store's first action is receiving from customer. These actions match, so the network can execute an action reducing both store and customer. This results in a new network, which is placed in a new node, and we connect both nodes by the transition label describing the executed action.

The next action by customer is receiving a label from store, but store needs to evaluate a conditional expression to decide which label to send. There are two possible outcomes for these evaluation, so we create two new nodes and label the edges towards them with the corresponding possibilities (`then` or `else`). Continuing to expand the `else` branch leads to a network that is already in the SEG, so we simply add an edge to the node containing that network. The `then` branch evolves in two steps into a second conditional, whose `else` branch again creates a loop, while its `then` branch evolves into a network where all processes has terminated. This concludes the construction of the SEG.

In this example SEG generation went flawlessly, but that is not always the case. We saw that a process trying to send a message must wait for the receiving process to be able to execute a matching receive; this can lead to situations where the network is *deadlocked* – no terms can be executed. In that case, the behaviour of the network cannot be described by a choreography, and the network cannot be extracted.

This algorithm also relies on the fact that network execution is confluent: the success of extraction does not depend on which action is chosen when constructing the SEG, in case several are possible. (This can affect the algorithm's performance, though.) Furthermore, guaranteeing that all possible evolutions of the network are captured requires some care when closing loops: all processes must reduce in every cycle in the SEG. This is achieved by marking processes in the network and checking that every loop contains a node where all processes are marked. These aspects are orthogonal to the current development, and we refer the interested reader to [3] for details.

Choreography construction. The main idea for generating a choreography from a SEG is that edges correspond to choreography actions, so the chore-

[1] The formal details can be found in [2,3].

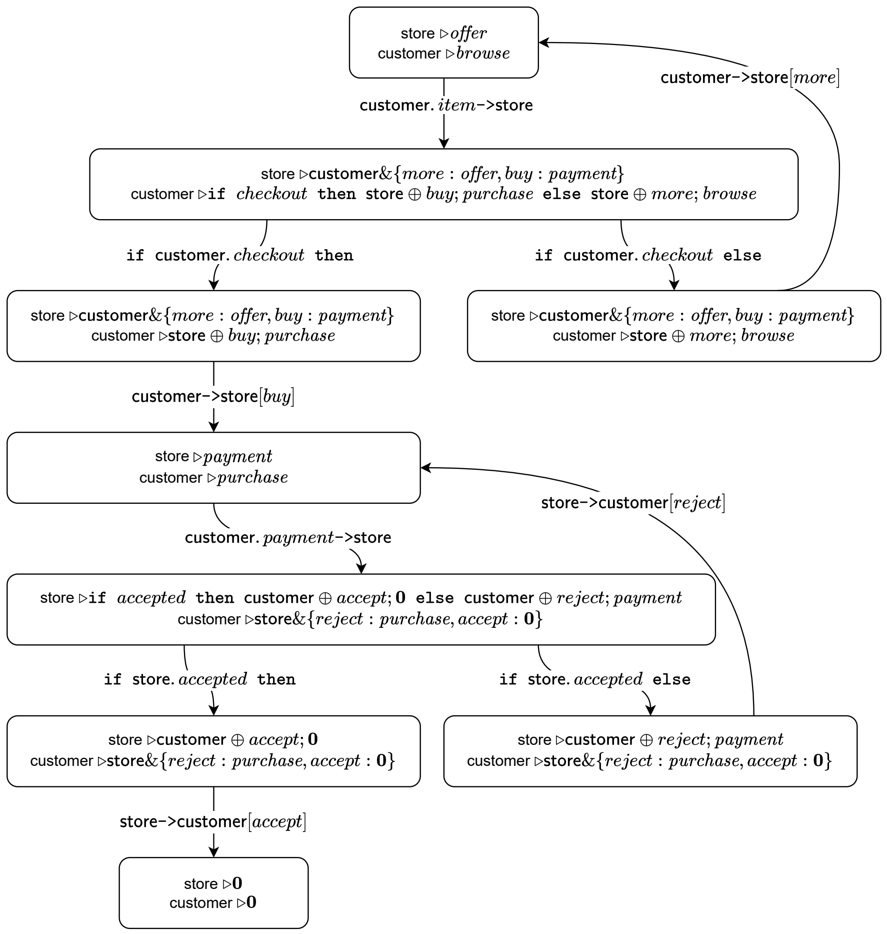

store ▷*offer*
customer ▷*browse*
customer.*item*->store
customer->store[*more*]
store ▷customer&{*more* : *offer*, *buy* : *payment*}
customer ▷if *checkout* then store ⊕ *buy*; *purchase* else store ⊕ *more*; *browse*
if customer.*checkout* then
if customer.*checkout* else
store ▷customer&{*more* : *offer*, *buy* : *payment*}
customer ▷store ⊕ *buy*; *purchase*
store ▷customer&{*more* : *offer*, *buy* : *payment*}
customer ▷store ⊕ *more*; *browse*
customer->store[*buy*]
store ▷*payment*
customer ▷*purchase*
store->customer[*reject*]
customer.*payment*->store
store ▷if *accepted* then customer ⊕ *accept*; **0** else customer ⊕ *reject*; *payment*
customer ▷store&{*reject* : *purchase*, *accept* : **0**}
if store.*accepted* then
if store.*accepted* else
store ▷customer ⊕ *accept*; **0**
customer ▷store&{*reject* : *purchase*, *accept* : **0**}
store ▷customer ⊕ *reject*; *payment*
customer ▷store&{*reject* : *purchase*, *accept* : **0**}
store->customer[*accept*]
store ▷**0**
customer ▷**0**

Fig. 1. The SEG of the network from Example 2.

ography essentially describes all paths in the SEG. The choreographic way of representing loops is by means of procedures, so each loop in the SEG should become a procedure definition. To achieve this, we first *unroll* the graph by splitting every *loop node* – the nodes that close a loop[2] – into two: a *exit node*, which is the target of all edges previously pointing to the loop node, and a *entry node*, which is the source of all edges previously pointing from the loop node. Entry nodes are given distinct procedure names, and exit nodes are associated with the corresponding procedure calls. The unrolled SEG is now a forest with each tree representing a procedure, as shown in Fig. 2.

[2] Formally, every node with at least two incoming edges – plus the starting node, if it has any incoming edges

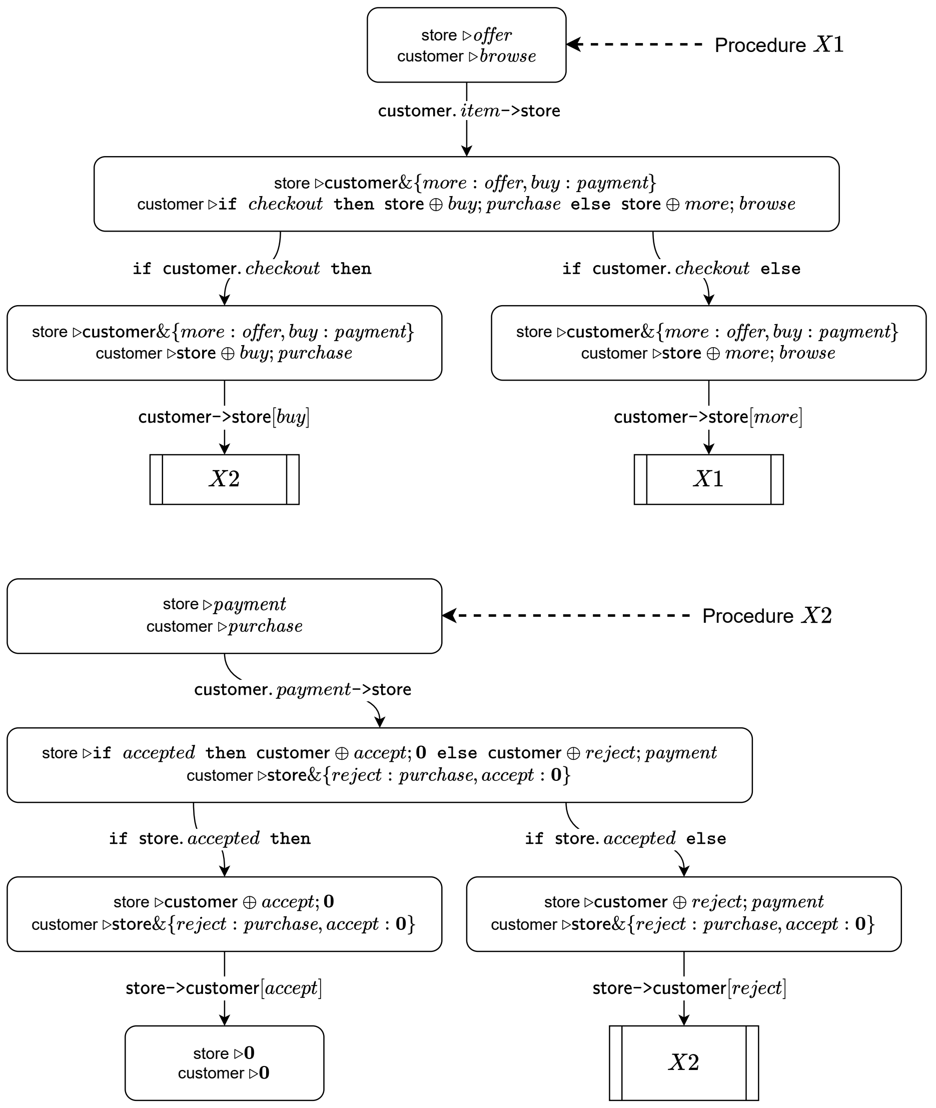

store ▷offer
customer ▷browse
Procedure X1
customer. item->store
store ▷customer&{more : offer, buy : payment}
customer ▷if checkout then store ⊕ buy; purchase else store ⊕ more; browse
if customer. checkout then
if customer. checkout else
store ▷customer&{more : offer, buy : payment}
customer ▷store ⊕ buy; purchase
store ▷customer&{more : offer, buy : payment}
customer ▷store ⊕ more; browse
customer->store[buy]
customer->store[more]
X2
X1
store ▷payment
customer ▷purchase
Procedure X2
customer. payment->store
store ▷if accepted then customer ⊕ accept; 0 else customer ⊕ reject; payment
customer ▷store&{reject : purchase, accept : 0}
if store. accepted then
if store. accepted else
store ▷customer ⊕ accept; 0
customer ▷store&{reject : purchase, accept : 0}
store ▷customer ⊕ reject; payment
customer ▷store&{reject : purchase, accept : 0}
store->customer[accept]
store->customer[reject]
store ▷0
customer ▷0
X2

Fig. 2. The unrolling of the graph of Fig. 1 resulting in two trees, each corresponding to a procedure in the SEG to be extracted. For readibility, the exit node corresponding to the invocation of $X2$ is depicted twice.

Since transition labels are similar to the choreography body terms, it is simple to read choreography bodies directly from each tree of the unrolled graph. This is done recursively, starting from the root of each tree and proceeding as follows: when encountering a node with no outgoing edges, then either all processes have terminated, in which case we return $\mathbf{0}$, or the node is an exit node, in which case we return the corresponding procedure invocation. If there is one outgoing edge, that edge represents an interaction, so we return the choreography body that starts with the transition label for that edge and continues as the result of the recursive invocation on the edge's target. If there are two outgoing edges, then we return a conditional choreography body whose two continuations are the results of the recursive calls targets of the two edges, as dictated by those edges' labels.

Example 4. By reading the trees in Fig. 2 in the manner described, we obtain the choreography given earlier in Example 3, where procedures *Buy* and *Pay* are now called $X1$ and $X2$, respectively.

3 Networks and Choreographies with Process Spawning

In this section, we extend the theories of networks and choreographies with support for spawning new processes at runtime.

We start by adding three primitives to the language of behaviours, following ideas from [4].

1. *Spawning of processes.* The language of behaviours is extended with the construct `spawn q with` B_{q} `continue` B, which adds a new process to the network with a new, unique name. The new process gets main behaviour B_{q}, and inherits its parent's set of procedures, while the parent continues executing B. This term also binds q (a process variable) in B.
2. *Advertising processes.* Since names of newly spawned processes are only known to their parents, we need terms for communicating process names. Process p can "introduce" q and r to each other (send each of them the other's name) by executing term q <-> $\mathsf{r}; B$, while q and r execute the dual actions $\mathsf{p?t};\ B_{\mathsf{q}}$ and $\mathsf{p?t};\ B_{\mathsf{r}}$. Here t is again a variable, which is bound in the continuations B_{q} and B_{r}.
3. *Parameterised procedures.* To be able to use processes spawned at runtime in procedures, their syntax is changed so that they can take process names as parameters.

We do not distinguish process names from process variables syntactically, as this simplifies the semantics. We assume as usual that all binders in the same term bind distinct variables, and work up to α-renaming. However, we allow a variable to occur both free and bound in the same term – this is essential for our algorithm.

As previously, the semantics includes a state function σ, mapping each process to a value (its memory state). The new ingredient is a graph of connections $\mathcal{G}$ between processes, connecting pairs of process that are allowed to communicate. The choice of the initial graph allows for modelling different network topologies. We use the notation $\mathsf{p} \leftrightarrow \mathsf{q} \in \mathcal{G}$ to denote that p and q are connected in $\mathcal{G}$, and $\mathcal{G} \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}$ to denote the graph obtained from $\mathcal{G}$ by adding an edge between p and q.

Fig. 3 includes some representative rules of this extended semantics.[3]

$$\frac{\mathsf{p} \leftrightarrow \mathsf{q} \in \mathcal{G} \qquad e \downarrow^{\sigma}_{\mathsf{p}} v}{\mathsf{p} \triangleright \mathsf{q}!e; B_1 \mid \mathsf{q} \triangleright \mathsf{p}?; B_2, \sigma, \mathcal{G} \xrightarrow{\mathsf{p}.v \texttt{->} \mathsf{q}} \mathsf{p} \triangleright B_1 \mid \mathsf{q} \triangleright B_2, \sigma[\mathsf{q} \mapsto v], \mathcal{G}} \ \text{N|Com}$$

$$\frac{\mathsf{p} \leftrightarrow \mathsf{q} \in \mathcal{G} \qquad \mathsf{p} \leftrightarrow \mathsf{r} \in \mathcal{G}}{\begin{array}{c}\mathsf{p} \triangleright \mathsf{q} \texttt{<->} \mathsf{r}; B_{\mathsf{p}} \mid \mathsf{q} \triangleright \mathsf{p}?\mathsf{r}; B_{\mathsf{q}} \mid \mathsf{r} \triangleright \mathsf{p}?\mathsf{q}; B_{\mathsf{r}}, \sigma, \mathcal{G} \\ \xrightarrow{\mathsf{p.q} \texttt{<->} \mathsf{r}} \\ \mathsf{p} \triangleright B_{\mathsf{p}} \mid \mathsf{q} \triangleright B_{\mathsf{q}} \mid \mathsf{r} \triangleright B_{\mathsf{r}}, \sigma, \mathcal{G} \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\}\end{array}} \ \text{N|Intro}$$

$$\frac{}{\begin{array}{c}\mathsf{p} \triangleright \texttt{spawn q with } B_q \texttt{ continue } B, \sigma, \mathcal{G} \\ \xrightarrow{\texttt{p spawns q}} \\ \mathsf{p} \triangleright B \mid \mathsf{q} \triangleright B_q, \sigma, \mathcal{G} \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}\end{array}} \ \text{N|Spawn}$$

Fig. 3. New semantics of networks, selected rules.

Rule N|Com describes a communication. Process p wants to send the result of evaluating e to process q, and q is expecting to receive from p. These processes can communicate, and the result v of evaluating e at p is sent and stored in q (premise $e \downarrow^{\sigma}_{\mathsf{p}} v$). The difference from the previous semantics is the presence of the additional premise $\mathsf{p} \leftrightarrow \mathsf{q} \in \mathcal{G}$, which checks that these two processes are allowed to communicate.

Rule N|Intro is similar, but process names are communicated instead, and the communication graph is updated. For simplicity, instead of explicitly substituting variables for process names, we assume that the behaviours of q and r have previous been α-renamed appropriately (this kind of simplifications based on α-renaming are standard in process calculi [18]).

Rule N|Spawn creates a new process q into the network with a unique name, and adds an edge between it and its parent to the network. Note that q is distinct from other process names in the network.

Choreographies get two corresponding actions: `p spawns q`; C and $\mathsf{p.q} \texttt{<->} \mathsf{r};\ C$. Procedures also become parameterized. At the choreography level we do not require process variables except in procedure definitions (which are replaced by process names when called); we assume that all names of spawned processes are unique (again treating `spawn` actions as binders and α-renaming in procedure bodies when needed at invocation time).

The corresponding rules for the semantics are given in Fig. 4.

Example 5. We illustrate a network with process spawning by writing Example 1 in our language. The client sends a request to an entry-point, which then spawns an instance to handle the request, which gets introduced to the client. The

[3] For the complete semantics, see the technical report [9].

$$\frac{\mathsf{p} \leftrightarrow \mathsf{q} \in \mathcal{G} \qquad e \downarrow^{\sigma}_{\mathsf{p}} v}{\mathsf{p}.e \mathrel{\texttt{->}} \mathsf{q}; C, \sigma, \mathcal{G} \xrightarrow{\mathsf{p}.v \texttt{->} \mathsf{q}} C, \sigma[\mathsf{q} \mapsto v], \mathcal{G}} \ \text{C}|\textsc{Com}$$

$$\frac{\mathsf{p} \leftrightarrow \mathsf{q} \in \mathcal{G} \qquad \mathsf{p} \leftrightarrow \mathsf{r} \in \mathcal{G}}{\mathsf{p}.\mathsf{q} \mathrel{\texttt{<->}} \mathsf{r}; C, \sigma, \mathcal{G} \xrightarrow{\mathsf{p}.\mathsf{q} \texttt{<->} \mathsf{r}} C, \sigma, \mathcal{G} \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\}} \ \text{C}|\textsc{Intro}$$

$$\frac{}{\mathsf{p}\ \texttt{spawns}\ \mathsf{q}; C, \sigma, \mathcal{G} \xrightarrow{\mathsf{p}\ \texttt{spawns}\ \mathsf{q}} C, \sigma, \mathcal{G} \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}} \ \text{C}|\textsc{Spawn}$$

Fig. 4. New semantics of choreographies, selected rules.

instance sends the client the result, and the client either makes another request or terminates the connection. Additional requests spawn new instances, since requests may differ in kind – this is handled by the previous instance to reduce load on the entry-point.

```
client {
    def X(s){
        s?w; w?; if more
            then w⊕next; X(w)
            else w⊕end; 0
    }
    main{ entry!req; X(entry) }
}
```

```
| entry {
    def X(this){
        spawn worker with
            this?client; client!res;
                client&{ next: X
                (worker), end: 0 }
        continue worker<->client; 0
    }
    main{ client?; X(entry) }
}
```

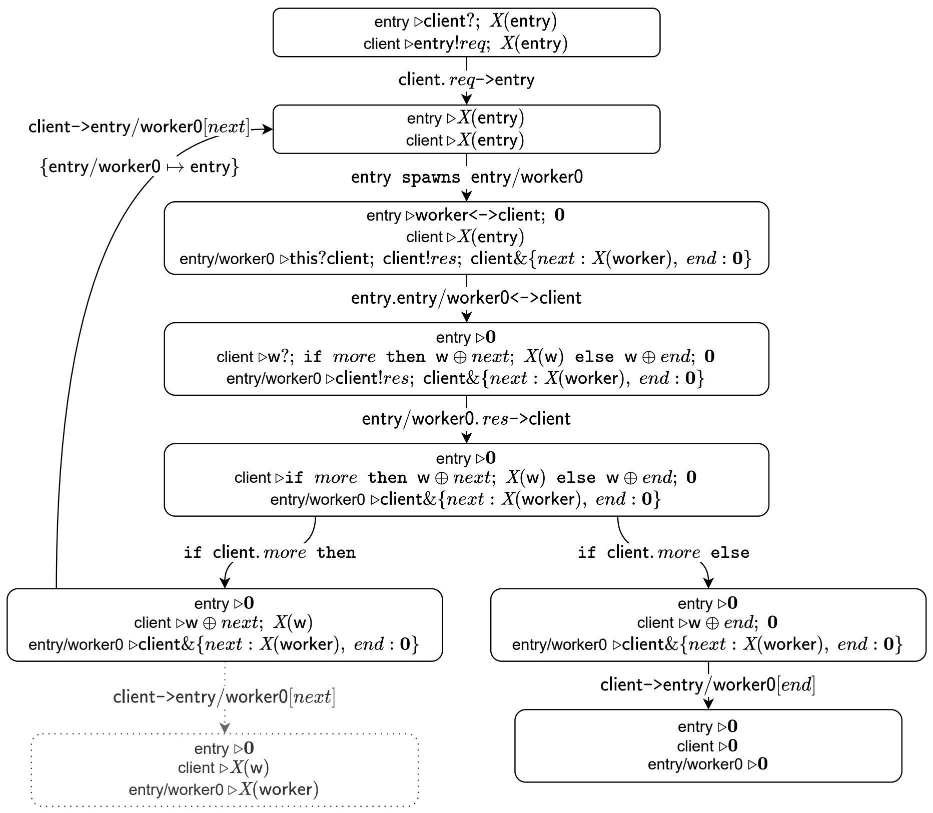
After the initial communication and procedure call, variable w in client needs to be renamed to worker in order for the next communication to reduce.

4 Extraction with process spawning

In the presence of spawning, the intuitive process of extraction described earlier no longer works: since a network can generate an unbounded number of new processes, there is no guarantee that the SEG is finite and, as a consequence, that the procedure terminates.

However, we observe that since networks are finite and can only reduce to networks built from their subterms, there is only a finite number of possible behaviours for processes that are spawned at runtime. Therefore we can keep SEGs finite if we allow renaming processes when connecting nodes. This intuition is key to our development.

Example 6. Consider the network from Example 5 and its SEG, shown in Fig. 5. The dotted part shows the network that would be generated by symbolic execution as described earlier. By allowing renaming of processes, we can close a loop by applying the mapping $\{\mathsf{client} \mapsto \mathsf{client}, \mathsf{entry/worker0} \mapsto \mathsf{entry}\}$. The parameters w and worker are both variables mapping to $\mathsf{entry/worker0}$, and since entry has terminated, the remapping makes the dotted node equivalent to the second node of the SEG, as shown by the loop. For simplicity we only show the process variables that are changed by the mapping, i.e., we omit $\mathsf{client} \mapsto \mathsf{client}$.



entry ▷client?; X(entry)
client ▷entry!req; X(entry)
client.req->entry
entry ▷X(entry)
client ▷X(entry)
client->entry/worker0[next]
{entry/worker0 ↦ entry}
entry spawns entry/worker0
entry ▷worker<->client; 0
client ▷X(entry)
entry/worker0 ▷this?client; client!res; client&{next : X(worker), end : 0}
entry.entry/worker0<->client
entry ▷0
client ▷w?; if more then w ⊕ next; X(w) else w ⊕ end; 0
entry/worker0 ▷client!res; client&{next : X(worker), end : 0}
entry/worker0.res->client
entry ▷0
client ▷if more then w ⊕ next; X(w) else w ⊕ end; 0
entry/worker0 ▷client&{next : X(worker), end : 0}
if client.more then
if client.more else
entry ▷0
client ▷w ⊕ next; X(w)
entry/worker0 ▷client&{next : X(worker), end : 0}
entry ▷0
client ▷w ⊕ end; 0
entry/worker0 ▷client&{next : X(worker), end : 0}
client->entry/worker0[next]
client->entry/worker0[end]
entry ▷0
client ▷X(w)
entry/worker0 ▷X(worker)
entry ▷0
client ▷0
entry/worker0 ▷0

Fig. 5. The SEG of the network in Example 5.

4.1 Generating SEGs

Formally, we define an abstract semantics for networks that makes two changes with respect to the concrete semantics given above. First, we remove all information about states σ (and as a consequence about the actual values being communicated), as in [2]. Secondly, we now treat *all* process names as variables, and replace the communication graph $\mathcal{G}$ by a partial function γ mapping pairs of a process name and a process variable to process names: intuitively, $\gamma(\mathsf{p}, \mathsf{q})$

returns the name of the actual process that p locally identifies as q. If $\gamma(\mathsf{p},\mathsf{q})$ is undefined, then p does not know who to communicate with. We assume that initially $\gamma(\mathsf{p},\mathsf{p}) = \mathsf{p}$ for all p, and that, for all p and q, either $\gamma(\mathsf{p},\mathsf{q}) = \mathsf{q}$ (meaning that p knows q's name and is allowed to communicate with it) or $\gamma(\mathsf{p},\mathsf{q})$ is undefined (meaning that p is not connected to q and cannot communicate with it) – this allows us to model different initial network topologies. Fig. 6 shows the abstract versions of the rules previously shown.

$$\frac{\mathsf{r} \downharpoonright^{\gamma}_{\mathsf{p}} \mathsf{q} \qquad \mathsf{s} \downharpoonright^{\gamma}_{\mathsf{q}} \mathsf{p}}{\mathsf{p} \triangleright \mathsf{r}!e; B_1 \mid \mathsf{q} \triangleright \mathsf{s}?; B_2, \gamma \xrightarrow{\mathsf{p}.e\ \text{->}\ \mathsf{q}} \mathsf{p} \triangleright B_1 \mid \mathsf{q} \triangleright B_2, \gamma}\ \text{N}|\textsc{Com}$$

$$\frac{\mathsf{s} \downharpoonright^{\gamma}_{\mathsf{p}} \mathsf{q} \qquad \mathsf{t} \downharpoonright^{\gamma}_{\mathsf{p}} \mathsf{r} \qquad \mathsf{u} \downharpoonright^{\gamma}_{\mathsf{q}} \mathsf{p} \qquad \mathsf{v} \downharpoonright^{\gamma}_{\mathsf{r}} \mathsf{p}}{\begin{array}{c}\mathsf{p} \triangleright \mathsf{s} \text{<->} \mathsf{t}; B_{\mathsf{p}} \mid \mathsf{q} \triangleright \mathsf{u}?\mathsf{w}; B_{\mathsf{q}} \mid \mathsf{r} \triangleright \mathsf{v}?\mathsf{x}; B_{\mathsf{r}}, \gamma \\ \xrightarrow{\mathsf{p}.\mathsf{q}\ \text{<->}\ \mathsf{r}} \\ \mathsf{p} \triangleright B_{\mathsf{p}} \mid \mathsf{q} \triangleright B_{\mathsf{q}} \mid \mathsf{r} \triangleright B_{\mathsf{r}}, \gamma[\langle \mathsf{q},\mathsf{w}\rangle \mapsto \mathsf{r}][\langle \mathsf{r},\mathsf{x}\rangle \mapsto \mathsf{u}]\end{array}}\ \text{N}|\textsc{Intro}$$

$$\frac{}{\begin{array}{c}\mathsf{p} \triangleright \texttt{spawn q with } B_q \texttt{ continue } B, \gamma \\ \xrightarrow{\texttt{p spawns q}} \\ \mathsf{p} \triangleright B \mid \mathsf{r} \triangleright B_q, \gamma[\langle \mathsf{p},\mathsf{q}\rangle \mapsto \mathsf{r}][\langle \mathsf{r},\mathsf{p}\rangle \mapsto \mathsf{p}]\end{array}}\ \text{N}|\textsc{Spawn}$$

Fig. 6. Abstract semantics of networks, selected rules

Example 7. It is easy to check that the SEG shown in Fig. 5 follows the rules in Fig. 6. Initially, the variable mapping is the identity, and this remains unchanged after the first communication.

γ	entry	client	worker	w
entry	entry	client	—	—
client	entry	client	—	—

When entry spawns the new process entry/worker0, this name is associated to entry's local variable worker according to rule N|Spawn. So the variable mapping is now given by the following table.

γ	entry	client	worker	w
entry	entry	client	entry/worker0	—
client	entry	client	—	—
entry/worker0	entry	—	—	—

Next, entry introduces entry/worker0 and client to each other; client uses the local name w for the new process. According to rule N|Intro, the variable mapping is now the following.

γ	entry	client	worker	w
entry	entry	client	entry/worker0	—
client	entry	client	—	entry/worker0
entry/worker0	entry	client	—	—

To determine whether we can close a loop in the SEG, we need the following definitions.

Definition 1. *Two networks N and N' are* equivalent *if there exists a total bijective mapping M from processes in N to processes in N', such that, for all processes* p*:*

- *if* p *has main behaviour B, then $M(\mathsf{p})$ has main behaviour $M(B)$ (where M is extended homeomorphically to behaviours);*
- *if* p *has not terminated and $X(\tilde{\mathsf{q}}) = B$ is a procedure definition in* p*, then $X = M_{\mathsf{q}}(B)$ is a procedure definition in $M(\mathsf{p})$, where M_{q} maps every process in $\tilde{\mathsf{q}}$ to itself and every other process* r *to $M(\mathsf{r})$.*

To close a loop in the SEG, we also need to look at the variable mappings γ.

Definition 2. *Two nodes N, γ and N', γ' in a SEG are* behaviourally equivalent *if:*

- *There exists a series of reductions $\tilde{\lambda}$ in the SEG such that $N, \gamma \xrightarrow{\tilde{\lambda}}^{*} N', \gamma'$;*
- *There exists a mapping function M that proves N and N' are equivalent.*
- *If $M(\mathsf{p}) = \mathsf{q}$, $\gamma(\mathsf{p}, \mathsf{a}) = \mathsf{b}$, $\gamma'(\mathsf{q}, \mathsf{a}) = \mathsf{c}$, and there is a reduction accessible from N, γ that evaluates $\gamma^*(\mathsf{p}, \mathsf{a})$ for some intermediary γ^*, then $M(\mathsf{b}) = \mathsf{c}$.*

The last point of the definition only applies to variables actually evaluated in reductions of N. This makes extraction more efficient (as more nodes are equivalent): a variable might have been used for a previous step in the evolution up to N, σ, but if it remains unused thereafter it does not affect the behaviour anymore, and can be ignored.

Lemma 1. *Let N, γ and N', γ' be behaviourally equivalent nodes in a SEG for a given network. The graph obtained by redirecting all edges coming into N', γ' to N, γ and removing the nodes that are no longer accessible from the root is also a SEG for the original network.*

Our implementation simply looks for a suitable N, γ when N', γ' is generated, and records the mapping M in the edge leading to N, γ.

Termination. As we mentioned at the start of this section, our algorithm can only generate a finite number of behaviours for each process involved in the network. This is enough to guarantee termination, following the arguments in [2], unless the number of processes in the network is allowed to grow unboundedly.

This situation can occur if processes are spawned in a loop, faster than they terminate, making the number of processes increase for every iteration. Such networks embody a resource leak, and they cannot be extracted by our theory. To ensure termination, our algorithm must be able to detect resource leaks, which is an undecidable problem. We deal with it as follows: when a new candidate node is generated, we check whether there is a *surjective* mapping with the properties described above such that there are at least two process values mapped to the same process name. If this happens, the algorithm returns failure. The interested reader can find some examples of networks with resource leaks in the technical report [9].

4.2 Generating the choreography

Building a choreography from a SEG is similar to the original case. The main change deals with procedure calls: we use the variable mappings in edges that close loops to determine their parameters and arguments.

When unrolling the SEG, each node corresponding to a procedure definition gets a list of parameters corresponding to the process names[4] that appear in the co-domain of any variable mapping in an edge leading to that node. Each procedure call is then appended to the reverse images of these processes by the mapping in the edge leading to it. Note that the edges do not contain process names that are mapped to themselves; as such, processes that are the same in all maps will not appear as arguments to the extracted procedure. A consequence of this is that some procedures may get an empty set of arguments.

After this transformation the choreography can again be extracted by recursively traversing the resulting forest.

We formulate the correctness of our extraction procedure in terms of strong bisimilarity [19].

Theorem 1. *If C is a choreography extracted from a network N, then $C \sim N$.*

Example 8. We return to the network in Example 5, whose SEG was shown in Fig. 5. The only loop node has two incoming edges, one with empty (identity) mapping and another with $\{\mathsf{entry/worker0} \mapsto \mathsf{entry}\}$. Therefore this node is extracted to a procedure $X1$ with one process variable entry. The original call simply instantiates this parameter as itself, while the recursive call replaces it with entry/worker0.

The choreography extracted from this SEG is thus the following.

```
def X1(entry) {
    entry spawns entry/worker0; entry.entry/worker0 <-> client;
    entry/worker0.res -> client;
    if client.more
        then client -> entry/worker0[next]; X1(entry/worker0)
        else client -> entry/worker0[end]; 0
}
main { client.req -> entry; X1(entry) }
```

[4] Assuming some predefined ordering of process names.

4.3 Implementation and limitations

The extension of the original extraction algorithm to networks with process spawning has been implemented in Java. It can successfully extract the networks in the examples given here, as well as a number of randomly generated tests following the ideas from [3]. Due to space constraints, we do not report on the details of our testing strategy, which is an extension of the strategy presented in detail in [3], extended in the natural way to include networks with process spawning and introduction.

Since our language only allows for tail recursion, divide-and-conquer algorithms such as mergesort are currently still not extractable, and our next plan is to extend the algorithm to deal with general recursion. This is not a straightforward extension, as our way of constructing the SEG has no way of getting past a potentially infinite recursive subterm to its continuation.[5]

Another example of an unextractable network, which does not use general recusion, is the following.

```
s {
    def X(p,t){
        if cont then spawn q with X(t,q) continue q?; p!m; 0 else p!m; 0
    }
    main{ X(p,s) }
} |
p{s?; stop}
```

Although the spawned processes behave as their parent, the entire network never repeats itself, and extraction fails: extracting a choreography would require closing a loop where some processes did not reduce. This is essentially the same limitation already discussed in [3], and cannot be avoided: given that the problem of determining whether a network can be represented by a choreography is in general undecidable [2], soundness of our algorithm implies that such networks will always exist.

5 Conclusion

We showed how the state-of-the-art algorithm for choreography extraction [2,3] could be extended to accommodate for networks with process spawning. This adaptation requires allowing processes names to change dynamically, so that the total number of networks that needs to be consider remains finite. The resulting theory captures examples including loops where processes that are spawned at runtime take over for other processes that terminate in the meantime. This extension also required adding parameterised procedures to the network and choreography language, and including a form of resource leak detection to ensure termination.

A working implementation of choreography extraction with process spawning is available at [10].

[5] This was also the reason for only including tail recursion in the original work [2].

References

1. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distributed Comput. **31**(1), 51–67 (2018). `https://doi.org/10.1007/s00446-017-0295-1`
2. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: Esparza, J., Murawski, A.S. (eds.) Proceedings of FoSSaCS. Lecture Notes in Computer Science, vol. 10203, pp. 424–440 (2017)
3. Cruz-Filipe, L., Larsen, K.S., Montesi, F., Safina, L.: Implementing choreography extraction. CoRR **abs/2205.02636** (2022), `https://arxiv.org/abs/2205.02636`, submitted for publication.
4. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: Bouajjani, A., Silva, A. (eds.) Proceedings of FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 92–107. Springer (2017)
5. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020). `https://doi.org/10.1016/j.tcs.2019.07.005`
6. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. J. ACM **63**(1), 9 (2016). `https://doi.org/10.1145/2827695`
7. International Telecommunication Union: Recommendation Z.120: Message sequence chart (1996)
8. Kjær, B.A.: Implementing Choreography Extraction in Java. Bachelor thesis, University of Southern Denmark (2020)
9. Kjær, B.A., Cruz-Filipe, L., Montesi, F.: From infinity to choreographies: Extraction for unbounded systems. CoRR **abs/2207.08884** (2022), `https://arxiv.org/abs/2207.08884`, technical report.
10. Kjær, B.A.: Choreographic extractor (May 2022). `https://doi.org/10.5281/zenodo.6554763`
11. Lange, J., Tuosto, E.: Synthesising choreographies from local session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7454, pp. 225–239. Springer (2012). `https://doi.org/10.1007/978-3-642-32940-1_17`
12. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 221–232. ACM (2015). `https://doi.org/10.1145/2676726.2676964`
13. Montesi, F.: Choreographic Programming. Ph.D. Thesis, IT University of Copenhagen (2013), `https://www.fabriziomontesi.com/files/choreographic-programming.pdf`
14. Montesi, F.: Introduction to Choreographies. Accepted for publication by Cambridge University Press (2022)
15. Object Management Group: Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/` (2011)
16. Object Management Group: Unified modelling language, version 2.5.1 (2017)
17. Safina, L.: Formal Methods and Patterns for Microservices. Ph.D. thesis, University of Southen Denmark (2019)
18. Sangiorgi, D.: Pi-i: A symmetric calculus based on internal mobility. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings. Lecture Notes in Computer

Science, vol. 915, pp. 172–186. Springer (1995). https://doi.org/10.1007/3-540-59293-8_194, https://doi.org/10.1007/3-540-59293-8_194

19. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2011). https://doi.org/10.1017/CBO9780511777110