Guess Who's Coming: Runtime Inclusion of Participants in Choreographies*

Maurizio Gabbrielli[1], Saverio Giallorenzo[2], Ivan Lanese[1], and Jacopo Mauro[2]

[1] University of Bologna and Focus Team INRIA
[2] University of Southern Denmark

Abstract. In Choreographic Programming, a choreography specifies in a single artefact the expected behaviour of all the participants in a distributed system. The choreography is used to synthesise correct-by-construction programs for each participant.
In previous work, we defined Dynamic Choreographies to support the update of distributed systems at runtime.
In this work, we extend Dynamic Choreographies to include new participants at runtime, capturing those use cases where the system might be updated to interact with new, unforeseen stakeholders. We formalise our extension, prove its correctness, and present an implementation in the AIOCJ choreographic framework.

Keywords: Choreographic programming · Adaptation of distributed systems · Dynamic inclusion of new software components.

1 To go or not to go ... almost an introduction

"Would you go to Padova?" Bob asked while typing on his old computer.

"Well, it depends" answered Alice, without stopping to stare at the paper on her desk. She was puzzled by the proof she was reading. The more she was going deep into it, the less she was convinced about its correctness.

"Depends on what?"

"Oh, on many factors: time and effort required, money, you know, the usual things ... I don't understand how it's been possible to publish this proof. It's completely hand-waved! And you know what the author told me when I met him at a conference in Jerusalem? He said that the proof was not accurate because ... we're not mathematicians but computer scientists! Can you believe that? Computer scientists ... then, my dear hand-waver, what is computer science? A branch of astrology or, perhaps, by any chance, something that happens to use also math?" Alice was getting visibly nervous about that paper and its author. The questions from Bob did not help in relaxing her.

* Research partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233.

"Hum, let's try this package" mumbled Bob after a short silence "this should fix these nasty Latex problems ... I believe everything in Padova is pretty standard. Nothing sensationally different from other places. So all in all those factors you mentioned do not help in the decision."

"Then, perhaps, it could be worth considering if you really want to go there. I mean, of course it could be nice, but is it really necessary?" commented Alice.

"Indeed, that is the right question. Is it necessary, or perhaps could one avoid it? Perhaps one could do something else or go to some other place. All in all, there are many places in Italy ... it's a really difficult choice."

The conversation continued for a while, with Alice and Bob working at their own desks. Then, the door of the office slammed open. "Guess who's coming!" shouted Charlie entering the room "How are things going? How many papers have you written in the last week?"

Charlie was the kind of office-mate who continuously asks questions. Always anxious about his performance, compared to the ones of his colleagues. He was a nice guy, and often his questions raised interesting problems. However, when he was at the office, it was simply impossible to do serious technical work.

"What are you talking about? Some new full abstraction proof?" pressed Charlie.

Alice raised her head, stretched her legs, looked for a while out of the window, and then answered "Oh, nothing serious, we are talking about going to Padova." Bob was baffled: "Well, if you allow me, my dear, it's a serious matter. It's actually quite important for my life!"

"Come on Bob, it's just a trip. Moreover, I was not invited to Padova. So, all in all, it's not nice of you to ask me all these questions!"

Bob stopped typing on his computer. He sat still for a few seconds. Then, he slowly turned his head towards Alice. He was genuinely surprised. He was used to some strange remarks from Alice but this time she was beyond her standards.

"Why in the world should they invite you to Padova, Alice!? You have already accepted a position at the University of Genova. Of course the University of Padova did not consider you."

"Position? University? Wait a minute! What are you talking about? Work has nothing to do with this trip to Padova. It's just a trip to go and visit that exhibition George has been talking about for months. He could have invited me too ... but that's OK, after all I've a lot of work to do in these days, so, no problem, really, no problem" replied Alice.

Bob's head was spinning "Exhibition? Why in the universe should I go and visit an exhibition in Padova? And why should I ask your opinion about going to an exhibition? I was talking about accepting or not a position at the University of Padova, that's what I was talking about!" Silence fell like a hammer.

"Well colleagues," broke Charlie "see how my simple question has solved a problem that otherwise could have lasted, unresolved, for hours? It was a kind of deadlock! So both of you owe me a beer. See you!" And with the usual slam of the door Charlie exited the room leaving Alice and Bob speechless.

In the story above, the reader might have recognised Catuscia, who played the role of Alice. Indeed this is a true anecdote, dating back to the Pisa times "some" years ago. Also the story about the wrong proof is true, but we leave to the reader the task of finding out who the author of the proof was (difficult) or guess the identity of Bob (easy). Charlie also corresponds to a real person, even though his character was changed a bit for narrative reasons. And, yes, George is . . . George!

At the time of this story choreographic languages had not been invented yet, however, in our opinion, the story has four morals: i) formalising conversations among computer scientists, as if they were distributed applications, can sometimes be useful; ii) allowing the inclusion of new participants at runtime, interacting in unforeseen ways, can also be useful; iii) as a consequence of i) and ii) dynamic choreographic languages with runtime inclusion of new participants—as presented in this paper—can be useful not only for computer applications, but also for computer scientists; iv) most importantly, never, ever, ask a question to Catuscia if she is working on a proof!

2 Introduction

Today, applications are often distributed and involve multiple participants which interact by exchanging messages. Programming the intended behaviour of such applications requires understanding how the behaviour of a participant program combines with that of others, to produce the global behaviour of the application. There is a tension between the *global* desired behaviour of a distributed application and the fact that it is programmed by developing *local* programs. Choreographic Programming [1,2] provides a good trade-off by allowing developers on the one hand to program the global behaviour directly and, on the other hand, to automatically generate the local programs that implement the global behaviour. As an example, consider the following choreography that describes the behaviour of an application composed of one `client` and one `seller`:

```
1  product@client = getInput( "Insert product name" );
2  quote: client( product ) -> seller( order )
```

The execution starts with an action performed by the `client`: an input request to the local user (line 1). The semicolon at the end of the line is a sequential composition operator, hence the user input should complete before execution proceeds to line 2. Then, a communication between the `client` and the `seller` takes place: the `client` sends a message and the `seller` receives it.

This code specifies the global behaviour and can be compiled automatically into two different local programs, one for the client and one for the seller.

Choreographies avoid by construction the presence of common errors like communication deadlocks and message races and are therefore useful to implement correct distributed systems [3]. Moreover, as testified by the dynamic choreographic language AIOCJ [2], choreographies can be extended to support the adaptation of running distributed applications. This is indeed an important task for the development of modern cloud native applications since, due to the

widely adoption of DevOps techniques for Continuous Integration and Continuous Deployment [4,5,6], the entire application needs to be updated, upgraded with new features, or patched guaranteeing no downtimes.

AIOCJ proposes a general mechanism to structure application updates. Inside applications, blocks of code delimited by *scope*s may be dynamically replaced by new blocks of code, called *update*s.

For instance, consider the previous example and assume that the interaction between the `client` and `seller` may change. This is enabled by replacing the second line with the following code.

```
scope @client{
  quote: client( product ) -> seller( order ) }
```

In essence, a scope is a delimiter that defines which part of the application can be updated. Each scope identifies a *coordinator* of the update (`client` in this case). The coordinator is the participant responsible for ensuring that the distributed adaptation of the code within the scope, if and when needed, is performed successfully. The code change is specified by program rules that target the desired scope. For instance, assuming that the seller requires not only the name of the product but also the customer location to make a targeted offer, the previous code can be changed by applying the following rule.

```
rule { do {
  loc@client = getInput( "Insert your location" );
  quote: client( product + loc ) -> seller( order ) }
}
```

Notably, rules can be defined and inserted in the system while it is running, without downtimes. At a choreographic level, updates are applied atomically to all the involved participants. The AIOCJ framework guarantees that the compilation and the application of the adaptation rules generate correct behaviours, avoiding the inconsistencies typical of distributed code updates.

While AIOCJ [2] provides a safe and reliable mechanism to adapt the code of running applications, it has one major weakness: it does not support the introduction of new participants at run time. Unfortunately, due to the change of business or legal needs, the need to introduce another entity (e.g., auditing program, logging system) that interacts with an existing and running system may arise.

In this work, we address this issue by proposing an extension of the Dynamic Choreographic AIOCJ framework which allows AIOCJ rules to add new participants to the choreography. In particular:

- we formalise the extension of AIOCJ rules to add new participants;
- we prove that the extension with new participants satisfies the properties of deadlock freedom and (for finite traces) of correctness of compilation and adaptation;
- we extend the AIOCJ development and deployment framework to support the addition of new participants.

Structure of the paper. To exemplify the approach we start by presenting in Section 3 a simple use case. In Section 4 we formalise the extension proving that it preserves the correctness properties. In Section 5 we present the implementation strategy. Section 6 discusses related work and Section 7 draws some concluding remarks.

3 A client-seller use case

In this section we present a simple use case exemplifying how new participants can be added at run time.

For simplicity, we consider a client-seller system for online shopping where the client sends messages to the seller that processes them and returns an answer to the client. Originally the program performs just this activity. Unfortunately, due to legislation changes, the requests by the client could go towards an auditing process and therefore they must be logged by an independent authority. Hence, logging can not be performed by the seller but requires a new and dedicated service, physically deployed in a location different from the seller one.

Adaptation is performed in two stages:

1. when writing the original AIOCJ program, one should foresee which parts of the code could be adapted in the future (but not which new behaviour will be required by the adaptation), and enclose them into `scope`s;
2. while the AIOCJ program is running, one should write adaptation rules to introduce the desired new behaviour.

```
1 product@client = getInput( "Insert product name" );
2 request: client( product ) -> seller ( order );
3 scope @seller{
4   // process the order and compute result, here XXX
5   result@seller = "XXX";
6   response: seller( result ) -> client ( result )
7 }
```

Let us suppose that the main deployed and running application was created by using the choreographic program above. The original programmer foresaw the possibility that the code run by the seller to compute the answer could be subject to changes. For this reason, the code at lines 4-6, where the answer is computed and sent back to the client, is enclosed in a scope.

To enforce the new legislation, the seller now needs to log all the incoming requests from the client. The logging entity is a new participant in the choreography. Using the extension for AIOCJ that we propose, the rule triggering the adaptation can be written as shown in Listing 1.1

The rule introduces a new participant (also called a role) `logger` by using the keywords `newRoles` (line 4). New roles in AIOCJ rules are different from roles previously involved in the target AIOCJ program (if needed they are renamed to avoid clashes of names) and take part to the choreography only while the body

```
1  rule {
2   include log from "socket://localhost:8002"
3   include getTime from "socket://localhost:8003"
4   newRoles: logger
5   location@logger: "socket://independent_autority.com:8080"
6   on {} // applicability conditions, irrelevant in this example
7   do {
8    log: seller( order ) -> logger( entry );
9    {
10     time@logger = getTime();
11     _@logger = log( time + ": " + entry )
12   } | {
13     // process the order and compute result, here XXX
14     result@seller = "XXX";
15     response: seller( result ) -> client ( result )
16   }}
17 }
```

Listing 1.1. Adaptation rule adding new role.

of the rule executes. As for normal roles, the URI of new roles is declared using the keyword `location` (line 5). In this particular case we suppose that the code will be deployed at the location reachable at URL `independent_autority.com` on port 8080.

Lines 2-3 define two external services `log` and `getTime` that are used, respectively, to store a message in the database and to get the current time. These two services are supposed to be available on the facility where the `logger` code will run. The new code, as specified in lines 8-15, requires the server to send relevant information on each transaction to the logger (in parallel to the answer to the client). In particular, the server computes a timestamp for each request (line 10) and logs the transaction information together with the timestamp (line 11).

To conclude this section, we remark that for presentation purposes the running example uses only a limited subset of the AIOCJ functionalities not including, e.g., the mechanism for fine-grained rule applicability (provided by the `on` block) which can be used to specify when and whether a rule applies to a given scope. We refer the reader to [7,2] for further details and to the AIOCJ website[3] for the full code of the example (choreography and rule) and the executable programs generated by their compilation.

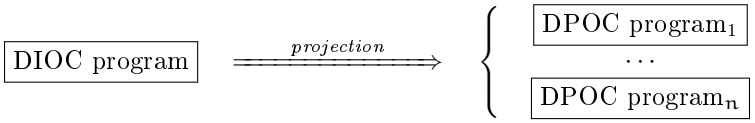
4 Theoretical Model

In this section, we give a brief overview of the theory of Dynamic Choreographies [2] and we present the changes needed to support the inclusion of new roles in runtime updates. For the sake of presentation we do not report all the formal

[3] http://www.cs.unibo.it/projects/jolie/aiocj_examples/external_roles.html

definitions detailed in [2], but we just give a general intuition of the theory presented in [2] focusing only on the formalisation of our extension. We also prove that some correctness results from [2] are preserved by the extension.

A graphical representation explaining the key elements of Dynamic Choreographies is depicted below.



From the left, we have DIOC, which stands for Dynamic Interaction-Oriented Choreographies. This is the high-level language that programmers use to specify the behaviour of the whole distributed system and models the one we have used in the code snippets in the previous sections. It will be presented in Section 4.1. The second element is the *projection* procedure which transforms a DIOC into a set of executable programs, one for every participant in the DIOC code. Since the extension in this paper does not affect the projection from [2], we just illustrate the intuition behind it in Section 4.3. Finally, we have the target language of the projection, DPOC, standing for Dynamic Process-Oriented Choreographies. DPOC is a calculus inspired by process calculi like the π-calculus and CCS, and it is equipped with primitives for runtime code update. It is designed as a common abstraction for real languages (C, Java, Python) to make the theory more general avoiding to target a specific language. DPOC is presented in Section 4.2.

4.1 Dynamic Interaction-Oriented Choreographies

We start by presenting the language to program applications, called DIOC.

DIOCs rely on a set of *Roles*, ranged over by $\mathbf{R}, \mathbf{S}, \ldots$, to identify the participants in the choreography. Each role has its own local state. Roles exchange messages over labels, called *operations* and ranged over by o. We require expressions, ranged over by e, to include at least values, belonging to a set *Val* ranged over by v, and variables, belonging to a set *Var* ranged over by $x, y, \ldots$. *DIOC processes* are ranged over by $\mathcal{I}, \mathcal{I}', \ldots$. We present below the DIOC production rules.

$$
\begin{array}{rlllllll}
\mathcal{I} ::= & \mathsf{o} : \mathbf{R}(e) \to \mathbf{S}(x) & \textit{(interact)} & | & x@\mathbf{R} = e & \textit{(assign)} & | & \mathcal{I};\mathcal{I}' \ \textit{(seq)} \\
& \textbf{if } \mathsf{b}@\mathbf{R}\,\{\mathcal{I}\}\ \textbf{else}\ \{\mathcal{I}'\} & \textit{(cond)} & | & \textbf{scope}\ @\mathbf{R}\ \{\mathcal{I}\} & \textit{(scope)} & | & \mathcal{I}|\mathcal{I}' \ \textit{(par)} \\
& \textbf{while } \mathsf{b}@\mathbf{R}\,\{\mathcal{I}\} & \textit{(while)} & | & \mathbf{0} & \textit{(end)} & | & \mathbf{1} \ \textit{(skip)}
\end{array}
$$

Interaction $\mathsf{o} : \mathbf{R}(e) \to \mathbf{S}(x)$ means that role $\mathbf{R}$ sends a message on operation o to role $\mathbf{S}$. The sent value is obtained by evaluating expression e in the local state of $\mathbf{R}$ (evaluation of expressions is always atomic) and it is then stored in the local variable x of $\mathbf{S}$. Assignment $x@\mathbf{R} = e$ assigns the evaluation of expression e in the local state of $\mathbf{R}$ to its local variable x. Processes $\mathcal{I};\mathcal{I}'$ and $\mathcal{I}|\mathcal{I}'$ denote the standard sequential and parallel composition of processes. The conditional $\textbf{if } \mathsf{b}@\mathbf{R}\,\{\mathcal{I}\}\ \textbf{else}\ \{\mathcal{I}'\}$ and the iteration $\textbf{while } \mathsf{b}@\mathbf{R}\,\{\mathcal{I}\}$ are guarded by the evaluation

of the boolean expression b in the local state of $\mathbf{R}$. The construct **scope** $@\mathbf{R}\ \{\mathcal{J}\}$ delimits a subterm $\mathcal{J}$ of the DIOC process that may be updated in the future. In **scope** $@\mathbf{R}\ \{\mathcal{J}\}$, role $\mathbf{R}$ is the coordinator of the update, which ensures that either none of the participants update, or they all apply the same update. Finally, $\mathbf{1}$ defines a DIOC process that can only terminate while $\mathbf{0}$ represents a terminated DIOC. The latter is needed for the definition of the operational semantics and not intended to be used by the programmer. We call initial a DIOC where $\mathbf{0}$ never occurs.

DIOC processes $\mathcal{J}$ execute within a DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{J} \rangle$, which pairs them with a *global state* Σ (disjoint union of the local states) and a set of available updates $\mathbf{I}$, i.e., a set of DIOCs that may replace scopes. Set $\mathbf{I}$ may change at runtime. The semantics of DIOC systems is defined in terms of a labelled transition system (LTS) of the shape $\langle \Sigma, \mathbf{I}, \mathcal{J} \rangle \xrightarrow{\mu} \langle \Sigma', \mathbf{I}', \mathcal{J}' \rangle$ where we use μ to range over the labels. Notably, changes of set $\mathbf{I}$ are visible in the labels, thus allowing to track or restrict the set of available updates if needed.

To support inclusion of new roles in DIOCs we need to change one rule in their semantics, namely rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ (reported below). As done in [2], we annotate the **scope** (as well as other constructs) with an index $\mathsf{i} \in \mathbb{N}$.[4] The only requirement over annotated DIOCs is that indexes within the same program must to be distinct.

To apply an update, we need to make sure that the roles marked as new in the update are not present in the running DIOC. While in principle one could consider as new all roles not present in the target scope, in practice one needs to declare the location only for new roles, hence they need to be distinguished. For this reason, from now on updates include a set of roles expected to be new. Thus, we introduce function $\mathsf{newRoles}$ that, given an update, returns the set of new roles. Function $\mathsf{newRoles}$ is similar to function roles (cf. [2]), which instead extracts the set of roles present in a given DIOC $\mathcal{J}$.

We report below the form of rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ used by our extension.

$$\frac{\mathcal{J}' \in \mathbf{I} \quad \mathsf{roles}(\mathcal{J}') \setminus \mathsf{newRoles}(\mathcal{J}') \subseteq \mathsf{roles}(\mathcal{J}) \quad \mathcal{J}'' = \mathsf{fresh}(\mathcal{J}') \quad \mathsf{connected}(\mathcal{J}') \quad \mathsf{freshIndexes}(\mathcal{J}')}{\langle \Sigma, \mathbf{I}, \mathsf{i}\colon \mathbf{scope}\ @\mathbf{R}\ \{\mathcal{J}\} \rangle \xrightarrow{\mathcal{J}''} \left\langle \Sigma, \mathbf{I}, \mathcal{J}'' \right\rangle} \lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$$

First, condition $\mathcal{J}' \in \mathbf{I}$ selects an update in the set of updates. Then, condition $\mathsf{roles}(\mathcal{J}') \setminus \mathsf{newRoles}(\mathcal{J}') \subseteq \mathsf{roles}(\mathcal{J})$ ensures that roles which are not declared as new were already present in the scope. This weakens the condition $\mathsf{roles}(\mathcal{J}') \subseteq \mathsf{roles}(\mathcal{J})$ in [2], which required *all* roles to be already present in the scope. The update which is actually applied (and advertised in the label of the transition), $\mathcal{J}''$, is obtained by renaming new roles so that they are fresh for the whole DIOC. Renaming is performed by function fresh that generates new names for roles

[4] Annotated DIOC constructs are useful in the definition of the projection and the related proofs to avoid interference between different constructs.

w.r.t. the whole DIOC. [5] Finally, the predicate $\mathsf{connected}(\mathcal{I}')$ checks that the DIOC is well formed while the predicate $\mathsf{freshIndex}(\mathcal{I}')$ checks that there are no interaction interferences between the rule and the original DIOC. We refer to [2] for the detailed description of both the predicates, since their behaviour is orthogonal to the addition of new roles.

4.2 Dynamic Process-Oriented Choreographies

DPOC is the abstract and formally defined language used as target by the projection function. Hence, this is the language of the programs that implement the DIOC specification.

DPOCs include *processes*, ranged over by $\mathsf{P}, \mathsf{P}', \ldots$, describing the behaviour of participants. A process P for *DPOC role* $\mathbf{R}$ executing in a local state Γ is denoted as $(\mathsf{P}, \Gamma)_{\mathbf{R}}$. A collection of executing processes for different roles is a *Network*, ranged over by $\mathcal{N}$, $\mathcal{N}'$. Finally, a DPOC system is a DPOC network equipped with a set of updates $\mathbf{I}$, namely a pair $\langle \mathbf{I}, \mathcal{N} \rangle$.

Like in DIOCs, DPOC processes communicate over operations o. Among all the communications, there are some auxiliary ones that the projection uses to implement the synchronisation mechanisms needed to realise the global choreography. We use o^* to range over auxiliary operations and $\mathsf{o}^?$ to range over both normal and auxiliary operations.

Following [2], for technical reasons, DPOC constructs are annotated using indexes $\mathsf{i}, \iota \in \mathbb{N}$. Indexes are also used to disambiguate operation names. The syntax of DPOCs is the following.

$$
\begin{array}{llllll}
\mathsf{P} ::= & \iota\colon \mathsf{i}.\mathsf{o}^? : \mathsf{x}\ \mathbf{from}\ \mathbf{R} & \textit{(receive)} & \mid \mathsf{i}\colon \mathbf{if}\ \mathsf{b}\ \{\mathsf{P}\}\ \mathbf{else}\ \{\mathsf{P}'\} & \textit{(cond)} & \mid \mathsf{P}; \mathsf{P}' \quad \textit{(seq)} \\
\mid & \iota\colon \mathsf{i}.\mathsf{o}^? : \mathsf{e}\ \mathbf{to}\ \mathbf{R} & \textit{(send)} & \mid \iota\colon \mathbf{scope}\ @\mathbf{R}\ \{\mathsf{P}\}\ \mathbf{roles}\ \{\mathsf{S}\} & \textit{(coord)} & \mid \mathsf{P} \mid \mathsf{P}' \quad \textit{(par)} \\
\mid & \iota\colon \mathsf{i}.\mathsf{o}^* : \mathsf{X}\ \mathbf{to}\ \mathbf{R} & \textit{(send-up)} & \mid \iota\colon \mathbf{scope}\ @\mathbf{R}\ \{\mathsf{P}\} & \textit{(scope)} & \mid \mathbf{1} \quad \textit{(skip)} \\
\mid & \iota\colon \mathsf{x} = \mathsf{e} & \textit{(assign)} & \mid \iota\colon \mathbf{spawn}\ @\mathbf{R}\ \{\mathsf{P}\} & \textit{(spawn)} & \mid \mathbf{0} \quad \textit{(end)} \\
\mid & \mathsf{i}\colon \mathbf{while}\ \mathsf{b}\ \{\mathsf{P}\} & \textit{(while)} & & &
\end{array}
$$

$$
\mathsf{X} ::= \quad \mathbf{no} \quad \mid \quad \mathsf{P} \qquad\qquad \mathcal{N} ::= \quad (\mathsf{P}, \Gamma)_{\mathbf{R}} \quad \mid \quad \mathcal{N} \parallel \mathcal{N}'
$$

DPOC processes include the action of receiving a message written as $\iota\colon \mathsf{i}.\mathsf{o}^? : \mathsf{x}\ \mathbf{from}\ \mathbf{R}$ and meaning that a message from role $\mathbf{R}$ is received on a specific operation $\mathsf{i}.\mathsf{o}^?$ (either normal or auxiliary) and the value stored in variable x. Similarly, the send action $\iota\colon \mathsf{i}.\mathsf{o}^? : \mathsf{e}\ \mathbf{to}\ \mathbf{R}$ sends the value of an expression e to operation $\mathsf{i}.\mathsf{o}^?$ of role $\mathbf{R}$. DPOC offers also a higher-order send action that instead of sending a value sends a process. This operation, written as $\iota\colon \mathsf{i}.\mathsf{o}^* : \mathsf{X}\ \mathbf{to}\ \mathbf{R}$, means that the higher-order argument X is sent to role $\mathbf{R}$ and is used to distribute the updated code. In particular, X may be either the new DPOC process P

[5] A compositional formalisation of function fresh would require to check freshness at all the steps of the derivation of the transition, to avoid name clashes with roles which are not in the scope but only in the context. To simplify the presentation, here we assume that function fresh has access to the set of roles of the whole DIOC.

that $\mathbf{R}$ has to execute or a token no notifying that no update is needed and the execution can continue with the pre-existing code. Processes also feature assignment $\iota\colon x = e$ of the value of expression e to the variable x.

As standard for process calculi, $P; P'$ and $P|P'$ denote the sequential and parallel composition of P and P'. DPOC processes also include conditionals $i : \textbf{if } b \{P\} \textbf{ else } \{P'\}$ and loops $i : \textbf{while } b \{P\}$. We also have the process $\mathbf{1}$ that can only successfully terminate, and the terminated process $\mathbf{0}$. Peculiar for DPOC is instead the scope constructor for delimiting a block of code. There are two versions of it: one for the process leading the possible adaptation and one for a process involved in the adaptation but not leading it. Construct $\iota\colon \textbf{scope } @\mathbf{R} \{P\} \textbf{ roles } \{S\}$ defines a scope with body P and set of participants S, and may occur only inside role $\mathbf{R}$, which acts as coordinator of the update. The shorter version $\iota\colon \textbf{scope } @\mathbf{R} \{P\}$ is used instead inside the code of some role $\mathbf{R}_1$, which is not the coordinator $\mathbf{R}$ of the update. The difference is due to the fact that the coordinator $\mathbf{R}$ needs to know the set S of involved roles to be able to send to them their updates.

All these constructs were already present in the original DPOC described in [2]. For our extension, we introduce only the new construct $\iota\colon \textbf{spawn } @\mathbf{R} \{P\}$, which indicates the runtime creation of a new role $\mathbf{R}$ running behaviour P.

The semantics of DPOCs is defined in terms of a labelled transition system composed of two layers. One is the semantics of DPOC roles, which specifies the local actions of each process and has the shape $(P, \Gamma)_{\mathbf{R}} \xrightarrow{\delta} (P', \Gamma')_{\mathbf{R}}$. The second is the semantics of DPOC systems, which defines how roles interact with each other and has the shape $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}', \mathcal{N}' \rangle$.

We now present the changes introduced in the semantics of DPOCs to support runtime role inclusion. We updated three rules and introduced two new ones. We start describing the revised rule $\lfloor^{\text{DPOC}}{}_{|\text{Lead-Up}}\rceil$, which defines the semantics of the process coordinating the update in the case where an update takes place. Main novelties are: *i*) it supports the application of updates with roles not present in the body of the **scope** and *ii*) it includes in the behaviour of the coordinator the **spawn** instructions needed to create the new participants (if any) present in the rule. Below, we use greyed-out circled number to ease the description of the reductum.

$$\frac{Q = \mathsf{newRoles}(\mathcal{J}) \quad \mathsf{roles}(\mathcal{J}) \setminus Q \subseteq S \quad \mathsf{connected}(\mathcal{J}) \quad \mathsf{freshIndexes}(\mathcal{J})}{(\iota\colon \textbf{scope } @\mathbf{R} \{P\} \textbf{ roles } \{S\}, \Gamma)_{\mathbf{R}} \xrightarrow{\mathcal{J}}} \lfloor^{\text{DPOC}}{}_{|\text{Lead-Up}}\rceil$$

$$\left(\begin{array}{ll} \textcircled{1} & \prod_{\mathbf{R}_j \in Q} \iota\colon \textbf{spawn } @\mathsf{R}_j \{\iota\colon \textbf{scope } @\mathsf{R} \{\mathbf{1}\}\}; \\ \textcircled{2} & \prod_{\mathbf{R}_j \in (S \cup Q) \setminus \{\mathbf{R}\}} \iota\colon \iota.\mathsf{sb}^*_\iota : \pi(\mathcal{J}, \mathbf{R}_j) \textbf{ to } \mathbf{R}_j; \\ \textcircled{3} & \pi(\mathcal{J}, \mathbf{R}); \\ \textcircled{4} & \prod_{\mathbf{R}_j \in (S \cup Q) \setminus \{\mathbf{R}\}} (\iota\colon \iota.\mathsf{se}^*_\iota : \_ \textbf{ from } \mathbf{R}_j), \Gamma \end{array} \right)_{\mathbf{R}}$$

As done in $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ for DIOCs, in the updated version of $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$ we make sure that the roles included in an update which are not new (i.e., not contained in Q) are already present in the scope (i.e., contained in S). We also check that the update satisfies the predicates connected and freshIndexes. In DIOCs, rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ also α-converts new roles to ensure their freshness. In DPOCs the same conversion is performed at the level of DPOC networks, in rule $\lfloor^{\text{DPOC}}|_{\text{LIFT-UP}}\rceil$, described at the end of this section.

In the reductum of rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$, described below, π is the process-projection function (see Section 4.3, cf. [2]) that generates the code from choreography $\mathcal{I}$ for role $\mathbf{R}$. In the reductum, first (1) the coordinator requires the creation of all the new roles. All new spawned roles have the same behaviour $\mathfrak{i}$: **scope** @R {**1**}, which means that, when started, the new roles wait to receive from the coordinator of the update the code they need to execute (exactly as the old roles do when reaching a scope). Then, (2) the coordinator sends, using a high-order communication on auxiliary operation $\mathsf{sb}^*_{\mathfrak{i}}$, the new code that the other roles need to execute. After having updated all the coordinated roles, (3) also the coordinator executes its own part of the update $\pi(\mathcal{I}, \mathbf{R})$. Finally (4) the coordinator waits for a message from each role involved in the update to make sure that the updated code has been completed.

To support the **spawn** construct, we define its semantics at the level of DPOC roles with the two rules denoted $\lfloor^{\text{DPOC}}|_{\text{SPAWN}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{NEWROLE}}\rceil$. The former triggers the **spawn** at the level of DPOC roles, while the latter creates the new role at the level of DPOC systems.

$$\frac{}{(\mathfrak{i}\text{: }\mathbf{spawn}\ @\mathbf{R}'\ \{\mathsf{P}\}, \Gamma)_{\mathbf{R}} \xrightarrow{\mathbf{R}'\{\mathsf{P}\}} (1, \Gamma)_{\mathbf{R}}}\ \lfloor^{\text{DPOC}}|_{\text{SPAWN}}\rceil$$

$$\frac{\mathcal{N} \xrightarrow{\mathbf{R}'\{\mathsf{P}\}} \mathcal{N}'}{\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\tau} \langle \mathbf{I}, \mathcal{N} \parallel (\mathsf{P}, \Gamma_0)_{\mathbf{R}'}\rangle}\ \lfloor^{\text{DPOC}}|_{\text{NEWROLE}}\rceil$$

Note that new roles are paired with an empty state Γ_0.

To complete the update on the semantics of DPOCs, we need to revise the original lifting rule to avoid capturing the new label $\mathbf{R}'\{\mathsf{P}\}$, which is instead handled by rule $\lfloor^{\text{DPOC}}|_{\text{NEWROLE}}\rceil$.

$$\frac{\mathcal{N} \xrightarrow{\delta} \mathcal{N}' \quad \delta \notin \{\mathcal{I}, \mathbf{R}'\{\mathsf{P}\}\}}{\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\delta} \langle \mathbf{I}, \mathcal{N}'\rangle}\ \lfloor^{\text{DPOC}}|_{\text{LIFT}}\rceil$$

Finally, we need to change rule $\lfloor^{\text{DPOC}}|_{\text{LIFT-UP}}\rceil$ to perform the α-conversion of role names. Note that, while the rule selects $\mathcal{I}' \in \mathbf{I}$ as the DIOC update, reductions happen on its α-converted version $\mathcal{I}$.

$$\frac{\mathcal{I}' \in \mathbf{I} \quad \mathcal{N} \xrightarrow{\mathcal{I}} \mathcal{N}' \quad \mathcal{I} = \mathsf{fresh}(\mathcal{I}')}{\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\mathcal{I}} \langle \mathbf{I}, \mathcal{N}'\rangle}\ \lfloor^{\text{DPOC}}|_{\text{LIFT-UP}}\rceil$$

4.3 Projection

Given a DIOC program, the projection function proj returns a DPOC network (i.e., a combination of interacting DPOC processes) that implements the semantics of the originating DIOC program. Each process is obtained by projecting the DIOC behaviour on a specific role using the process projection function π. Since **spawn** constructs are introduced during the execution of scopes, there was no need to extend the original definition of the projection function.

For the sake of presentation here we provide just an example of application of the projection, referring the interested reader to [2] for the full definition. In particular, in the following we show the application of π to DIOC interactions. Function π, given an annotated DIOC process and a role $\mathbf{R}$, returns a DPOC process for role $\mathbf{R}$. Below, in case (1) the input role is the sender $\mathbf{R}_1$ and π returns the corresponding send action (indexed by i and on operation $\mathsf{i.o}$) towards the receiver $\mathbf{R}_2$. Case (2) is complementary to (1) for the reception while case (3) produces $\mathbf{1}$, i.e., the role has no part in the interaction and skips that action.

$$
\begin{array}{ll}
(1)\ \pi(\ \mathsf{i}\colon \mathsf{o} : \mathbf{R}_1(e) \to \mathbf{R}_2(\mathsf{x}), \mathbf{R}_1\ ) = \mathsf{i}\colon \mathsf{i.o} : e \textbf{ to } \mathbf{R}_2 & \\
(2)\ \pi(\ \mathsf{i}\colon \mathsf{o} : \mathbf{R}_1(e) \to \mathbf{R}_2(\mathsf{x}), \mathbf{R}_2\ ) = \mathsf{i}\colon \mathsf{i.o} : \mathsf{x} \textbf{ from } \mathbf{R}_1 & \\
(3)\ \pi(\ \mathsf{i}\colon \mathsf{o} : \mathbf{R}_1(e) \to \mathbf{R}_2(\mathsf{x}), \mathbf{R}'\ ) = \mathbf{1} & \text{if } \mathbf{R}' \notin \{\mathbf{R}_1, \mathbf{R}_2\}
\end{array}
$$

4.4 Example of projection and adaptation

To better clarify how our framework works, we provide here a minimal example of the projection and the adaptation step for an excerpt of the use case in Section 3, where the new role **logger** enters the choreography upon adaptation. We start by annotating the code from Section 3, to be able to project it since the projection function requires well-annotated DIOCs. For brevity, we consider the excerpt from lines 3–7 where the **seller** role coordinates the adaptation of the scope. We monotonically annotate every instruction starting from 1. This results in the well-annotated DIOC below.

```
1: scope @seller{
   2: result@seller = "XXX";
   3: response : seller(result) → client(result)}
```

From the annotated DIOC, the projection generates two DPOC processes, one for the **seller** and one for the **client**. We report below the projection on the **seller**, together with a step derived using rule $\lfloor^{\text{DPOC}}|_{\text{Lead-Up}}\rceil$, where new role **logger** enters the system. In the projection of the program, for simplicity, we omit the indexes that prefix the operations. On the left, we show the DPOC code that is obtained by projecting the previous choreography on the **seller** role. On the right, we present the DPOC process of the **seller** after the adaption step which applies the rule in Listing 1.1, whose body is denoted in the following by $\mathcal{J}$.

$$
\begin{array}{l}
\text{1: } \textbf{scope } @\textbf{seller}\{ \\
\quad \mathsf{result} = \text{"XXX"}; \\
\quad \mathsf{response} : \mathsf{result} \textbf{ to client} \\
\}\ \textbf{roles}\ \{\ \textbf{client}\ \}
\end{array}
\quad \xrightarrow{\mathcal{J}} \quad
\begin{array}{l}
\text{1: } \textbf{spawn } @\textbf{logger}\{1 : \textbf{scope}@\textbf{seller}\{1\}\}; \\
\{1{:}\ \ \mathsf{sb}_1^* : \pi(\mathcal{J}, \textbf{client}) \textbf{ to client} \mid \\
\quad 1{:}\ \ \mathsf{sb}_1^* : \pi(\mathcal{J}, \textbf{logger}) \textbf{ to logger}\}; \\
\pi(\mathcal{J}, \textbf{seller}); \\
\{1{:}\ \ \mathsf{se}_1^* : \_ \textbf{ from client} \mid \\
\quad 1{:}\ \ \mathsf{se}_1^* : \_ \textbf{ from logger}\}
\end{array}
$$

As can be seen, the projection of a DIOC scope is a DPOC scope. If no adaptation rule is applied, the body of the scope is executed, hence the **seller** simply computes the answer storing it in its variable result and then sends it to the **client**. If, instead, an adaptation rule is applied, the **seller** first spawns new roles (if any) found in the adaptation rule. In the case of the adaptation rule in Listing 1.1, the only new role is **logger**. Once spawned, the **logger** executes the code **scope**@**seller**{1}. Hence, the **logger** waits to get from the **seller** the code to be executed. It is up to the seller, as leader of the adaptation, to send the new code, obtained from the body of the adaptation rule using the π operator, to both the **client** and the **logger**. This is done by performing the communications on operations 1: sb_1^*. Then, the **seller** executes its own adapted code $\pi(\mathcal{J}, \textbf{seller})$. When the **seller** terminates the execution of its adapted code, it waits for all the led roles to notify that they also terminated the execution of their adapted code (this is done by communications on operations 1: se_1^*) before starting to execute the rest of the choreography.

4.5 Properties

Our model, extended to support the inclusion of new roles in runtime updates, preserves the correctness properties of [2]. In particular, we will show in Theorem 1 that a DIOC system and its projection are weak trace equivalent (for finite traces), that is they have the same behaviour up to internal τ actions and communications on auxiliary operations. As shown in [2], this property implies deadlock freedom, which is indeed formalised by requiring finite internal traces (an internal trace is obtained by removing all transitions with label $\mathcal{J}$ from a trace) to end with $\surd$.

Definition 1 (DIOC traces). *A* (strong) trace *of a DIOC system* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{J}_1 \rangle$ *is a sequence (finite or infinite) of labels* $\mu_1, \mu_2, \ldots$ *such that there is a sequence of DIOC system transitions* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{J}_1 \rangle \xrightarrow{\mu_1} \langle \Sigma_2, \mathbf{I}_2, \mathcal{J}_2 \rangle \xrightarrow{\mu_2} \ldots$.

A weak trace *of a DIOC system* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{J}_1 \rangle$ *is a sequence of labels* $\mu_1, \mu_2, \ldots$ *obtained by removing all silent labels* τ *from a trace of* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{J}_1 \rangle$.

Definition 2 (DPOC traces). *A* (strong) trace *of a DPOC system* $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$ *is a sequence (finite or infinite) of labels* $\eta_1, \eta_2, \ldots$ *with*

$$\eta_i \in \{\tau, o^? : \mathbf{R}_1(v) \to \mathbf{R}_2(x), o^* : \mathbf{R}_1(X) \to \mathbf{R}_2(), \surd, \mathcal{J}, \textbf{no-up}, \mathbf{I}\}$$

such that there is a sequence of transitions $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle \xrightarrow{\eta_1} \langle \mathbf{I}_2, \mathcal{N}_2 \rangle \xrightarrow{\eta_2} \ldots$.
A weak trace *of a DPOC system* $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$ *is a sequence of labels* $\eta_1, \eta_2, \ldots$ *obtained by removing all the labels corresponding to auxiliary communications, i.e.,*

of the form $\mathsf{o}^* : \mathbf{R}_1(\nu) \to \mathbf{R}_2(\mathsf{x})$ *or* $\mathsf{o}^* : \mathbf{R}_1(\mathsf{X}) \to \mathbf{R}_2()$*, and the silent labels* τ*, from a trace of* $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$*.*

DPOC traces do not allow send and receive actions. Indeed these actions represent incomplete interactions, thus they are needed for compositionality reasons, but they do not represent relevant behaviours of complete systems. Note also that these actions have no correspondence at the DIOC level, where only whole interactions are allowed.

Definition 3 (Finite trace equivalence). *Two DIOC systems, or two DPOC systems, or a DIOC and a DPOC system are* finite (weak) trace equivalent *iff their sets of finite (weak) traces do coincide.*

Lemma 1 (Projection preserves weak traces [2, Corollary 7.5]). *Consider the semantics without new roles. For each initial, connected DIOC process* $\mathcal{I}$*, each state* Σ*, each set of updates* $\mathbf{I}$*, the DIOC system* $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ *and the DPOC system* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ *are weak trace equivalent.*

We now show that a similar result holds in the system with new roles. We will show this only for finite traces. We conjecture that this holds also for infinite traces, however, while the proof for finite traces can be done using the result in [2, Corollary 7.5] as a black box, we have not been able to do the same for infinite traces. The alternative of redoing all the proofs in [2, Corollary 7.5] (including the preliminary results therein) is beyond the scope of this paper. This last option would also allow us to prove race freedom and orphan message freedom.

Theorem 1 (Projection preserves finite weak traces, with new roles). *For each initial, connected DIOC process* $\mathcal{I}$*, each state* Σ*, each set of updates* $\mathbf{I}$*, the DIOC system* $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ *and the DPOC system* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ *are finite weak trace equivalent.*

Proof (sketch). For each finite weak trace we provide a translation from the DIOC $\mathcal{I}$ generating it to a DIOC $\mathcal{I}'$ where the same finite weak trace does not require new roles. We show that $\mathcal{I}$ and $\mathcal{I}'$ are finite weak trace equivalent, and the same holds for their projections. Hence, given a finite weak trace of $\mathcal{I}$, $\mathcal{I}'$ also has it. Thanks to Lemma 1 this means that the projection of $\mathcal{I}'$ has the trace thus implying that the trace also belongs to the traces of the projection of the original system $\mathcal{I}$. Dually, given a finite weak trace of the projection of the original system $\mathcal{I}$, then the projection of $\mathcal{I}'$ has it thus implying that $\mathcal{I}'$ and $\mathcal{I}$ have the same trace.

Let us consider a finite fixed weak trace. It is clear that the trace uses only a finite amount of new roles. The translation takes a DIOC $\mathcal{I}$ with these new roles and replaces each scope $\mathbf{scope}\ @\mathbf{R}\ \{\mathcal{I}\}$ with

$$\mathbf{scope}\ @\mathbf{R}\ \{\mathbf{if}\ \mathtt{false}@\mathbf{R}\ \{\prod_{\mathbf{R}'} \mathtt{unused}@\mathbf{R}' = 0\}\ \mathbf{else}\ \{\mathbf{1}\}; \mathcal{I}\}$$

where the product stands for n-ary parallel composition, and ranges over all new roles that are created by instantiating the considered scope in the trace under analysis. Variable $\mathtt{unused}$ is an unused variable.

The evaluation of the conditional always selects the else branch, hence it only causes one additional τ step. Apart from this, DIOC $\mathcal{I}$ and its translation generate the same traces. Also, the trace under analysis can be generated by the new DIOC without the need for dynamically generating new roles.

Let us now consider the projections. The projection of $\mathcal{I}$ does not have the auxiliary steps needed for the conditional (a τ to evaluate the guard and some auxiliary communications and τ steps to notify the result of the evaluation to all the involved roles), and has some additional τ steps to spawn the new roles. However, these are all auxiliary steps, hence the weak traces do coincide. □

We remark that even if the proof above relies on the system without new roles, this does not mean that the two systems are equivalent. Indeed, the proof requires one system without new roles for each possible trace, hence infinitely many of them. One single system with new roles captures all these behaviours.

Finally, we note that the DIOC and DPOC are not image-finite [8]. Indeed, both of them allow for changing in an arbitrary way the set of available updates in one step. Hence, each system can have an infinite amount of successors. If we allow only for a finite number of rules at a time, given that the evaluation of expressions is deterministic, the system becomes image-finite and therefore the (infinite) weak trace equivalence can be derived directly from the finite weak trace equivalence [8].

5 Implementation

In this section we overview the components and functionalities of the AIOCJ runtime support and present the main design and implementation choices made to support the inclusion of new roles in choreographies.

AIOCJ is a framework that allows the development of choreographies and update rules that can be projected to runnable and deployable distributed programs. To improve usability, as exemplified in Section 3, the AIOCJ syntax used to define the choreographies is not the formal DIOC one, but an embellished version. The target language for the projection function is Jolie [9]. This language, often used to develop microservices [10], was adopted because it offers programming primitives very close to the DPOC ones, thus making the definition of the projection function easier.

The basic AIOCJ runtime support includes three kinds of components: the Adaptation Manager, the Rule Server, and the Environment. These components are depicted as rectangles in Fig.1. Additionally, to support inclusion of new roles at runtime, we introduced a new component, called the Role Supporter; depicted in Fig.1 as a rectangle with a darker colour.

All runtime support components are optional, meaning that an AIOCJ choreography without adaptation scopes does not need the presence of any of the runtime support components to execute. When scopes are present, the only mandatory component is the Adaptation Manager that interacts with the projected code to find possible applicable rules and retrieve their code. In turn, the

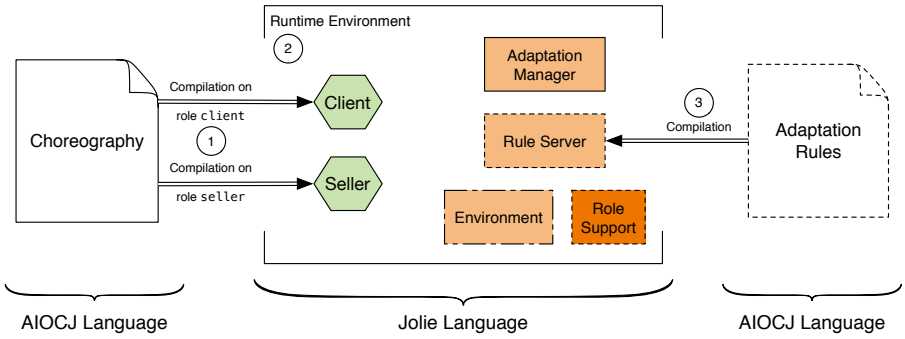


Fig. 1. AIOCJ components. Left and right, choreographic artefacts. Centre, executable components: projected roles (hexagons) and runtime support programs (rectangles).

Adaptation Manager works as registry for Rule Servers, which store the adaptation rules. Every time a new set of adaptation rules is projected with AIOCJ, the AIOCJ compiler synthesises a new Rule Server that contains the executable code corresponding to the projection of the body of the rule on each participant occurring there. A Rule Server registers to the Adaptation Manager, making its rules available.

When managing an adaptation step, the Adaptation Manager invokes the registered Rule Servers to check which rules are applicable. If at least one rule is applicable, the Adaptation Manager selects one and obtains the code update for the selected rule. Since in AIOCJ applicability conditions of rules may refer to properties of the execution environment (e.g., time, temperature), the AIOCJ runtime offers an Environment service that stores and publishes data on the status of the execution environment.

The Role Supporter component, as the name entails, supports the deployment of new roles. New roles are meant to add a new participant into a pre-existing choreography accessing new functionalities and useful, e.g., for system integration and evolution [11]. A Role Supporter component has to be deployed in the premises of the location of each new participant. The deployment information that is abstracted away in DIOCs, is instead explicitly defined in AIOCJ rules. New roles are marked with the keyword `newRoles`. Their location is stated using the syntax shown at line 5 of Listing 1.1 (`location@logger:"..."`).

The protocol to coordinate the instantiation of new roles is depicted in Fig. 2. The protocol starts with the request made by an adaptation leader (R_1) to the Adaptation Manager (1). After receiving the request, the Adaptation Manager contacts its registered Rule Servers to look for applicable rules (2). If there is an adaptation rule that is applicable, the Rule Servers find it (3). Now, if the selected rule contains new roles, the Rule Server checks if the new roles can be deployed (i.e., there are Role Supporters available at the locations of the new

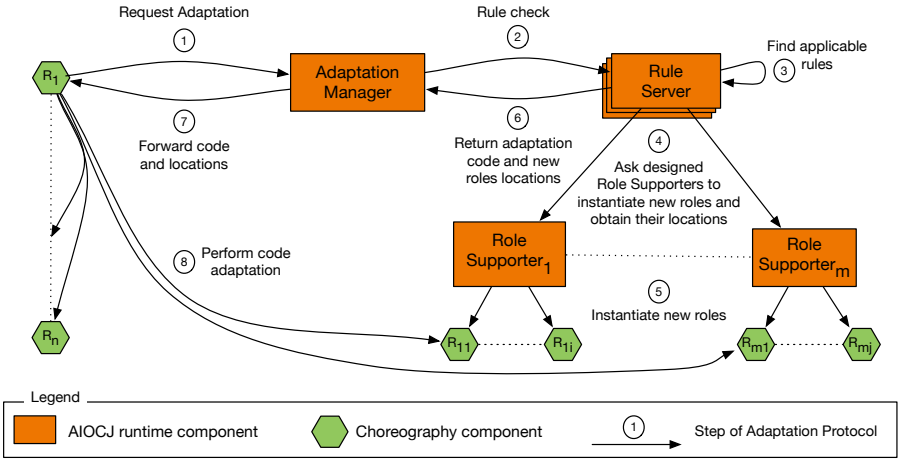


Fig. 2. Representation of the steps of adaptation with new roles.

roles).[6] Assuming new roles are needed, at step (4) the Rule Server contacts the interested Role Supporters and invokes the instantiation of dedicated new roles (e.g., $R_{11} \ldots R_{mj}$). The new roles are instantiated at step (5). Each new role is located at a unique, fresh address (prefixed by the location defined in the rule). In this way, parallel executions of the same rule can run without the need for any coordination among parallel rule applications. After each new role has started, its Role Supporter responds to the invoking Rule Server with the address of the instantiated role. Hence, the Rule Server collects the locations of all new roles, which the adaptation leader (R_1) will use to contact them to finalise the adaptation process. After this, or if no new roles are needed, the protocol continues with steps (6) and (7) that forward the adaptation code of each participant back to the Adaptation Manager and, immediately after, to the adaptation leader. Finally, at step (8), the adaptation leader (R_1) distributes the adaptation code to the participants (the previously present $R_1, \cdots, R_n$ and the newly instantiated $R_{11} \cdots R_{mj}$), to proceed executing the updated behaviour.

All the components for managing adaptation described above are written in Jolie [9] and can be easily deployed using some scripts we provide. For the extension, the code of the Rule Server has been updated to take into account the possibility to use Role Supporters. This last component has been instead created from scratch. The code of the Adaptation Manger and Environment did not require any changes. For more technical details on AIOCJ, an explanation on how to deploy and use AIOCJ with the new extension and its actual implementation we refer the interested reader to [7,12].

[6] Note that in case the new roles cannot be instantiated, the adaptation rule is considered not applicable and the process of rule selection proceeds discarding this rule.

6 Related work

This paper extends dynamic choreographic programming [2] to support the introduction at runtime of new participants. While referring to the related work in [2] for further details, in this section we describe the main distinctive features of our approach and the work closest to it. As far as we know, the approach in [2] is the only one encompassing i) adaptation for distributed systems, ii) guarantees of relevant correctness properties by construction, and iii) a working implementation. To the best of our knowledge, existing proposals share only two of those qualities.

In the literature we can find several middlewares and architectures enabling run-time adaptation [13,14,15,16,17] (see also the related survey [18]). These proposals provide tools for programming adaptive systems, but they do not offer by construction correctness guarantees on the behaviour of the system during and after adaptation. Some of them, however, such as [17], allow one to check correctness properties using techniques such as model checking. In order to do this, they assume knowledge of all the possible available adaptations at the moment of writing the adaptable application.

Other approaches are based on session types [19,20,21,22], choreography languages [23,24], behavioural contracts [25], and ad-hoc scripting languages [26]. Those works provide high-level specifications to describe the expected behaviour of a distributed system, ensuring relevant correctness properties, however they assume a static system and are not suitable for runtime adaptation.

There are also approaches based on adaptive choreographies [27,28,29], however they are not implemented and they concentrate on correctness checking more than on code generation. In particular, [27] concentrates on systems that autonomously switch among a set of pre-defined behaviours, [28] supports system update when no protocol is ongoing, and [29] requires to check global conditions on the system to ensure correctness of adaptation and therefore it is not suitable for large and complex distributed systems.

Finally, among the existing proposals based on choreographic programming, we note [30], where the authors define a compositionality mechanism for choreographies that, although not specifically targeted for adaptation, constitutes a first technical step to support it.

7 Conclusion and future work

We presented an extension of dynamic choreographic programming [2] to support the runtime introduction of new roles, extending both the related theory and the AIOCJ programming language [7].

Directions for future research include optimizing code generation to reduce the number of auxiliary communications, introducing in choreographic programs more structured forms of adaptation such as aspects, and developing or exploiting state-of-the-art DevOps tools to automatise the deployment of the different services generated by the AIOCJ framework.

References

1. F. Montesi, "Kickstarting choreographic programming," in *WS-FM*, pp. 3–10, Springer, 2015.
2. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro, "Dynamic choreographies: Theory and implementation," *Logical Methods in Computer Science*, vol. 13, no. 2, 2017.
3. S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, pp. 329–339, ACM, 2008.
4. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.
5. M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, and F. Montesi, "Self-reconfiguring microservices," in *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pp. 194–210, Springer, 2016.
6. M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, and G. Zavattaro, "Optimal and automated deployment for microservices," in *FASE*, pp. 351–368, Springer, 2019.
7. "AIOCJ website." http://www.cs.unibo.it/projects/jolie/aiocj.html.
8. J. A. Bergstra, A. Ponse, and S. A. Smolka, eds., *Handbook of Process Algebra*. New York, NY, USA: Elsevier Science Inc., 2001.
9. F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with Jolie," in *Web Services Foundations*, pp. 81–107, Springer, 2014.
10. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering.*, pp. 195–216, Springer, 2017.
11. S. Giallorenzo, I. Lanese, and D. Russo, "Chip: A choreographic integration process," in *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, pp. 22–40, Springer, 2018.
12. S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli, "Programming adaptive microservice applications: An AIOCJ tutorial," in *Behavioural Types: from Theory to Tools*, pp. 147–167, River Publishers, 2017.
13. A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik, "Dynamic Adaptation of Fragment-Based and Context-Aware Business Processes," in *ICWS*, pp. 33–41, IEEE, 2012.
14. W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing Adaptive Software in Distributed Systems," in *ICDCS*, vol. 6084 of *LNCS*, pp. 635–643, Springer, 2001.
15. C. Ghezzi, M. Pradella, and G. Salvaneschi, "An evaluation of the adaptation capabilities in programming languages," in *SEAMS*, pp. 50–59, ACM, 2011.
16. I. Lanese, A. Bucchiarone, and F. Montesi, "A Framework for Rule-Based Dynamic Adaptation," in *TGC*, vol. 6084 of *LNCS*, pp. 284–300, Springer, 2010.
17. J. Zhang, H. Goldsby, and B. H. C. Cheng, "Modular Verification of Dynamically Adaptive Systems," in *AOSD*, pp. 161–172, ACM, 2009.
18. L. A. F. Leite *et al.*, "A systematic literature review of service choreography adaptation," *Service Oriented Computing and Applications*, vol. 7, no. 3, pp. 199–216, 2013.

19. M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, p. 8, 2012.
20. M. Carbone and F. Montesi, "Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming," in *POPL*, pp. 263–274, ACM, 2013.
21. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani, "On global types and multiparty session," *Logical Methods in Computer Science*, vol. 8, no. 1, 2012.
22. K. Honda, N. Yoshida, and M. Carbone, "Multiparty Asynchronous Session Types," in *POPL*, pp. 273–284, ACM, 2008.
23. S. Basu, T. Bultan, and M. Ouederni, "Deciding choreography realizability," in *POPL*, pp. 191–202, ACM, 2012.
24. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro, "Bridging the Gap between Interaction- and Process-Oriented Choreographies," in *SEFM*, pp. 323–332, IEEE, 2008.
25. M. Bravetti and G. Zavattaro, "Towards a unifying theory for choreography conformance and contract compliance," in *SC*, vol. 4829 of *LNCS*, pp. 34–50, Springer, 2007.
26. J. A. Bergstra and P. Klint, "The discrete time TOOLBUS - A software coordination architecture," *Sci. Comput. Program.*, vol. 31, no. 2-3, pp. 205–229, 1998.
27. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri, "Self-adaptive multiparty sessions," *Service Oriented Computing and Applications*, vol. 9, no. 3-4, pp. 249–268, 2015.
28. C. Di Giusto and J. A. Pérez, "Event-based run-time adaptation in communication-centric systems," *Formal Asp. Comput.*, vol. 28, no. 4, pp. 531–566, 2016.
29. G. Anderson and J. Rathke, "Dynamic software update for message passing programs," in *APLAS*, vol. 7705 of *LNCS*, pp. 207–222, Springer, 2012.
30. F. Montesi and N. Yoshida, "Compositional choreographies," in *CONCUR*, vol. 8052 of *LNCS*, pp. 425–439, Springer, 2013.