

A Choreographic Calculus for Dangling-Function-Free Microservice-Serverless Systems

Saverio Giallorenzo^{1,2}, Mauro Vergnani¹, and Gianluigi Zavattaro^{1,2}

¹ Università di Bologna, Bologna, Italia

² INRIA, Sophia-Antipolis, France

{saverio.giallorenzo@,mauro.vergnani@studio.,gianluigi.zavattaro@}unibo.it

Abstract. We propose FaaSChalCore, the first formal choreographic calculus for specifying and verifying hybrid Microservice-Function-as-a-Service (FaaS) serverless systems. Specifically, FaaSChalCore captures the key primitives of serverless function creation and makes global architectural logic explicit, thereby exposing and taming FaaS coordination concerns, detecting subtle component-level mismatches that would otherwise surface only at deployment time, and enabling correctness by construction for cloud-native serverless applications. We formalise and prove correct a static-analysis discipline for well-sorted FaaSChalCore programs, which rules out classes of serverless-specific communication and lifecycle errors arising from stateless function instances, short-lived workers, and function creation and termination, while enforcing constraints on termination order and correct function usage.

1 Introduction

Function-as-a-Service (FaaS) is a state-of-the-art serverless cloud-computing model in which developers build architectures out of individual, ephemeral functions. A FaaS platform handles provisioning, scheduling, and tearing down of the functions, executed in response to triggering events [20]. The FaaS approach is a complement to the other state-of-the-art approach of microservices [12], which are typically programmed as stateful processes that persist for the whole lifetime of the system they belong to. While microservices and FaaS seem at odds, modern, real applications configure as hybrids that combine long-running microservices with short-lived serverless functions [25, 2]. For instance, microservices excel at steady-state execution, with cost-effective and predictable infrastructure, and serverless functions shine at burst absorption, where rapid scale-out matters more than per-invocation cost.

In hybrid microservice-FaaS systems, FaaS coordination concerns compound with those of classical service-oriented architectures. Function instances are ephemeral, must be explicitly created by some other running component, may need to send a response back to their creator, must eventually terminate, and their behaviour is often subject to ordering constraints relative to other components in the same workflow. Because this coordination logic is mainly implicit and spread across application code and cloud-provider configurations, subtle mismatches between components (e.g., a function terminating before satisfying a pending response obligation) tend to surface only at deployment time rather than being caught early during development.

Moreover, since serverless functions are dynamically created and stateless, they make lifecycle reasoning substantially different from (and more complex than) microservices.

In essence, hybrid architectures introduce coordination problems due to the asymmetry that emerges from the mix of the two paradigms. Hence, we investigate whether a calculus can harness that asymmetry at the level of a formal coordination language.

A Choreographic Calculus for Hybrid Microservice-Serverless Systems Choreographic Programming (CP) is a paradigm for specifying the behaviour of a distributed system from a global viewpoint, describing the interactions among all participants, called *roles*, in a single program, rather than writing the system as separate local processes [22]. The essential element of a choreography is communication, e.g., of the form $e@p \rightarrow x@q$, meaning that role p sends the evaluation of expression e against its state to role q , which stores the value in variable x . Other typical components of CP languages include sequential composition, global conditionals (where one role determines the guard but multiple roles interact in branches), and procedures, used to structure larger protocols [22].

A central result of CP is Endpoint Projection (EPP), a compilation procedure that mechanically derives, from a single global choreography, the local program that each role must run, while preserving the communication safety and deadlock-freedom of the original specification [22]. EPP is the main practical motivation for CP, whereby distributed implementations inherit correctness properties guaranteed at the global level.

In this paper, we formalise a choreographic calculus, FaaSChalCore³, for modelling and verifying correctness properties of hybrid microservices-serverless systems.

In FaaSChalCore, microservices are stateful roles and serverless functions are stateless ones, with explicit creation and termination behaviour. Before delving into the details of our contribution, in Fig. 1, we present a brief example of a hybrid architecture that serves database queries behind an OpenID-like decentralised authentication protocol [24] in FaaSChalCore. In the example, we find procedure `QueryWithOpenID`, which declares as formal parameters two stateful roles `client` and `idprov` (the other parameters of the declaration, separated by `|`, are absent). In the choreographic procedure, `client` starts a stateless role `DBwrp`, which can access a database⁴ and perform the query only if the identity provider, `idprov`, confirms `client`'s identity.

In the procedure, first `client` starts `DBwrp` through a request-response: `client` sends its `request`, which the newly created function `DBwrp` stores in `qryReq`; the fragment `>result@client` indicates that `DBwrp` needs to terminate by sending back a response, saved by `client` under `result`. Then, `DBwrp` starts an authentication session with `idprov`, sending a fresh challenge token and `client`'s username. Next, `idprov` generates an authentication challenge and sends it to `client`. `client` sends back to `idprov` the solved challenge, so that `idprov` can determine if it can authorise `client`. Note that, contrarily to what happens at line 5, where `idprov` directly sends the challenge to `client`, `idprov` cannot initiate an interaction with `DBwrp`. This

³ In designing our calculus, we draw inspiration from FaaSChal [23], a proposed choreographic programming language for hybrid microservice-serverless systems.

⁴ To keep the example small, we assume that the function-database interaction is implemented locally by `DBwrp`, but a straightforward extension of the example could include a forth role, `database`, to explicitly specify also that part of interaction.

```

1 def QueryWithOpenID(client,idprov||){
2   req@client  $\longleftrightarrow$  qryReq@DBwrp > result@client;
3   {tkn: genChalToken(),uname: qryReq.username}@DBwrp  $\rightarrow$  idReq@idprov;
4   genChallenge(idReq)@idprov  $\rightarrow$  challenge@client;
5   solve(challenge)@client  $\rightarrow$  challResp@idprov;
6   DBwrp  $\leftrightarrow$  checkChallenge(idReq, challResp)@idprov > isValidID@DBwrp;
7   if isValidID@DBwrp {
8     end queryDatabase( qryReq.query )@DBwrp  $\rightarrow$  client;
9   } else { end unit@DBwrp  $\rightarrow$  client; } }

```

Fig. 1. A hybrid microservice-FaaS interaction with OpenID-like access in FaaSChalCore.

limitation is typical of the FaaS paradigm, whereby, due to the ephemeral nature of running functions, one cannot directly interact with them. However, functions can interact with stateful processes, which allows *DBwrp* to fetch the result of the check from *idprov* with the instruction at line 6. Beside stateless and stateful, we distinguish between roles like *idprov* and *client*, which can both receive and send messages, calling them *conversational* (as it happens in roles of standard choreographic languages), and *inquiring*, like *DBwrp*, which can only initiate interactions with other roles (they can only respond, once, to their creator, if their creation happened through a request-response action).⁵ Closing the example, *QueryWithOpenID* terminates with *DBwrp* that controls a branching behaviour depending on the result of the authentication: either *DBwrp* runs the query and reports back to *client* the result or it sends a placeholder value (*unit*) to indicate a problem with the authentication procedure.

FaaSChalCore makes architectural dependencies visible in the choreography itself, rather than leaving them implicit in deployment code or platform configuration, and it allows static checks to rule out mismatches such as incorrect termination order or missing response obligations. Indeed, while CP languages typically exclude message deadlocks by construction – thanks to the atomic treatment of communication, where, $e@p \rightarrow x@q$ prevents one from writing unmatched send/receive instructions – CP programmers can make mistakes (e.g., passing wrong arguments in procedures). In particular, FaaSChalCore captures and surfaces the implicit configuration mistakes that programmers make on functions management (like missing/incorrect terminations and wrong workflow order), which we rule out by introducing a static analysis discipline for FaaSChalCore able to guarantee the absence of this specific category of errors.

2 Formalisation of FaaSChalCore

We define the structure of FaaSChalCore programs and their syntax and semantics.

Before defining the language, we introduce the domains of elements and variables used to range over them. **Var** is the set of variable names, ranged over by *x*, *y*, *z*,

⁵ Hence, stateful/stateless and conversational/inquiring classify different role traits, resp. *kind of processes* (e.g., microservice/serverless) and *interaction behaviours*. Practically, stateless roles are inquiring, and stateful ones can switch between conversational and inquiring depending on how they parametrise choreographic procedures.

$S ::= \overline{\text{def } X(F N T):(O)\{C\}}$	procedure definitions
$F ::= \mathbf{p}, F \mid \epsilon$	list of conversational roles (which must be stateful)
$T ::= \mathbf{f}, T \mid \epsilon$	list of terminating stateless roles: $\mathbf{f}$ sends no response to its creator
$N ::= \mathbf{n}, N \mid \epsilon$	list of nonconversational roles (stateful and non-terminating stateless)
$O ::= \mathbf{f} <: \mathbf{g}, O \mid \epsilon$	role ordering relation
$C ::= I; C \mid \mathbf{0}$	instruction sequencing
$I ::= e @ \mathbf{n} \rightarrow x @ \mathbf{q}$	$\mathbf{n}$ sends the evaluation of e to $\mathbf{q}$, which assigns it to x
$\mid \mathbf{n} \rightarrow \mathbf{q}[L]$	$\mathbf{n}$ sends the constant L to $\mathbf{q}$
$\mid \mathbf{n} \leftrightarrow e @ \mathbf{q} \triangleright x @ \mathbf{n}$	$\mathbf{n}$ fetches the value of e evaluated at $\mathbf{q}$, storing it in x
$\mid x @ \mathbf{n} = e @ \mathbf{n}$	$\mathbf{n}$ assigns the evaluation of e to x
$\mid e @ \mathbf{n} \Rightarrow x @ \mathbf{f}$	$\mathbf{n}$ sends a request, which creates a new stateless role $\mathbf{f}$, which assigns the evaluation of e to x
$\mid e @ \mathbf{n} \Longleftrightarrow x @ \mathbf{f} \triangleright y @ \mathbf{n}$	$\mathbf{n}$ sends a request-response; it acts like a request, but $\mathbf{f}$ must respond to $\mathbf{n}$, which will assign the response to y
$\mid \text{end } \mathbf{f}$	$\mathbf{f}$ terminates, without a response
$\mid \text{end } e @ \mathbf{f} \rightarrow \mathbf{n}$	$\mathbf{f}$ sends the evaluation of e to $\mathbf{n}$, which assigns it to the noted y . Then, $\mathbf{f}$ terminates
$\mid \text{if } e @ \mathbf{n} \text{ then } \{C_1\} \text{ else } \{C_2\}$	$\mathbf{n}$ evaluates the boolean expression e and continues with either C_1 or C_2 accordingly
$\mid X(F N T)$	call procedure X with conversational parameters F , nonconversational stateful and nonterminating stateless parameters N , and terminating stateless parameters T , associated with the roles they must respond to
$\mid N_P \mid T_P : X(F N T)$	the roles in N_P and T_P have yet to enter the call (runtime term)
$e ::= v \mid x \mid f(e_1, \dots, e_n)$	values, variables, and local function calls

Fig. 2. Syntax of FaaSChalCore.

representing variable identifiers; $\mathbf{Val}$ is the set of values, ranged over by v, u ; $\mathbf{R}$ is the set of role names, ranged over by $\mathbf{p}, \mathbf{q}, \mathbf{r}$, used for stateful process roles; $\mathbf{J}$ is the set of role names, ranged over by $\mathbf{f}, \mathbf{g}, \mathbf{h}$, used for stateless function instances; $\mathbb{R} = \mathbf{R} \cup \mathbf{J}$, ranged over by $\mathbf{n}, \mathbf{m}, \mathbf{l}$, indicates roles of either domains. The distinction between stateless and stateful roles is important: stateful roles represent processes (microservices) that we assume running throughout a choreography, while stateless roles must be started by a running role and have an explicit instruction to end their execution.

2.1 Syntax

A FaaSChalCore program includes a series of procedure declarations S , defined using the syntax in Fig. 2, of which `main` is the entry point of execution. When executing a FaaSChalCore program, we start from the body of the `main` procedure and use a companion structure $\mathcal{C}$ to carry the definitions of the other procedures.

The peculiarity of S definitions is that we identify three distinct lists of parameters – F , N , and T – separated by $|$. These parameters represent role identifiers of three kinds: $\mathbf{p}, \mathbf{q} \in F$ are conversational roles, $\mathbf{n}, \mathbf{m} \in N$ are inquiring roles (stateful and stateless) that cannot terminate within the scope of the procedure, while $\mathbf{f}, \mathbf{g} \in T$ are inquiring roles that must terminate within the procedure’s scope – technically, T contains pairs of roles, denoted $\mathbf{f}\mathbf{n}$, meaning that $\mathbf{f}$ was created by and has to respond back to $\mathbf{n}$; a “single” role is assumed paired with $\mathbf{0}$, which represents the

nonexistent role. Programmers can choose to place stateful roles both in F and N , based on whether they implement a conversational or inquiring behaviour.

The discrimination of role parameters in procedures is instrumental to build a modular static analysis discipline (cf. Section 3), so that looking at the “signature” of a procedure definition is sufficient to compute local checks at the call site and within a procedure’s body. We associate procedure definitions to a set O of elements $f <: g$ which defines, w.r.t. T , a strict partial order relation $<:$, called *termination order*, between the terminating parameters of the procedure. Hence, when writing $f <: g$ in O , roles f and g are distinct single or left-side elements in pairs of T (e.g., f^- and g^-). The programmer can use O to specify which stateless roles should terminate before others in a procedure’s body, i.e., $f <: g$ means that f terminates before g . In this way, programmers can express the relationships among (stateless) roles to refine the judgements of the static analysis – since we impose local checks, we ignore information about role dependencies at the site of procedure definition and, without more refined constraints from the user, we can only assume the worst-case scenario in the static analysis.

We illustrate this observation with an example in Fig. 3, which also allows us to exemplify the constructs found in the body of choreographic procedures (C , cf. Fig. 2).

On the right of Fig. 3, we find the main procedure that takes only one conversational parameter, p ; within the procedure’s body, p starts the stateless role f , sending the content of p ’s variable x , which f stores within variable y . The one-way directionality of the arrow between p and f indicates that p does not require a response from f . On the contrary, in the next instruction, f starts role g with a request-response; at g ’s termination (end), f will receive the contents of g ’s response in variable y . The last instruction of the procedure calls procedure X , giving as (actual) parameters g , which must respond to role f (i.e., $g \rangle f$) and f . Looking at the definition of X , we find a similar parameter structure (using formal parameters a , b , and c); in the body of the procedure, first c terminates and then a terminates, sending its response to b .

Our static analysis discipline would reject the program on the left of Fig. 3, because the actual parameters of procedure X are g (first position) and f (second position) and they must both terminate inside the procedure. The procedure’s signature does not indicate an ordering between the termination of those roles – O is empty, indicated by the absence of that fragment in X ’s declaration⁶. On the contrary, the code in *main* imposes, with the instruction at line 3 (see the comment in Fig. 3) that g must terminate before f . Since f terminates in X before g , g would remain “dangling” – orphan of the creator it has to send a response back to. Our static analysis discipline prevents that occurrence by rejecting the program on the left – notably, it is not the body of X that causes the rejection, but rather the lack of information on X ’s signature; indeed, the code within X ’s body is correct w.r.t. X ’s signature.

On the right of Fig. 3, we show a fixed version (relevant changes) of the program on the left, where we add the constraint that formal parameter a must terminate before c ($a <: c$ in O), which forces us to modify the body of X , switching the two lines.

⁶ Since O concerns static analysis, we mostly omit the construct in this section, to focus on the association between procedure parameters and choreographic bodies. O will become important in Section 3, where we use it to carry information about $<:$ and T .

<pre> 1 def main(p){ 2 o@p⇒x@f; 3 x@f⇐⇒x@g▷y@f; // we infer g<:f 4 X(g)f,f); 5 } 6 def X(a)b,c){ 7 end c; 8 end x@a→b; 9 }</pre>	<pre> 1 def main(p){ 2 same as left line 3 same as left line 4 same as left line 5 } 6 def X(a)b,c):(a<:c){ 7 end x@a→b; 8 end c; 9 }</pre>
---	--

Fig. 3. Example of wrong (left) and correct (right) FaaSChalCore programs.

2.2 Semantics

In the semantics, we usually treat F , N , and T as ordered sets, and apply operations on them as such. For compactness, we use $\mathbf{n} \in N$ to range over generic roles. T contains ordered pairs $(\mathbf{f}, \mathbf{s})$, where $\mathbf{s}$ can be a generic role $\mathbf{n}$, or the nonexistent role $\mathbf{0}$.

We denote **FaaSChalCore** the set of all terms which can be derived from C , i.e., the language used to define choreographic procedures within FaaSChalCore programs.

We define the behaviour of FaaSChalCore programs through Structured Operational Semantics, by providing a corresponding Labelled Transition System (LTS). First, we introduce a few concepts that will be used in the definition of the LTS.

Process Store and Choreographic Store A process store σ is a function that represents the memory of a process, by mapping variable names to values. We write **PStore** for the set of all process stores $\sigma: \mathbf{Var} \rightarrow \mathbf{Val}$. We define notation for updating a process store, $\sigma[y \mapsto v]$, as: $\sigma[y \mapsto v](x) = v$, if $x = y$; $\sigma(x)$, otherwise. A choreographic store Σ is a function that represents the memory of an entire running system by mapping roles to process stores – $\Sigma: \mathbb{R} \rightarrow \mathbf{PStore}$. We use choreographic stores to track running stateless roles. To allow for the creation of new stateless roles and update of variable values, we define notation for updating a choreographic store: $\Sigma[y @ \mathbf{m} \mapsto v](\mathbf{n}) = \Sigma(\mathbf{n})[y \mapsto v]$, if $\mathbf{n} = \mathbf{m}$; $\Sigma(\mathbf{n})$, otherwise. We also define notation $\Sigma \setminus \mathbf{f}$ for removing terminated stateless roles from the choreographic store, which corresponds to Σ after removing $\mathbf{f}$ from its domain.

The last notation we introduce refers to the evaluation of expressions against stores, where $\sigma \vdash e \downarrow v$ means that the process store σ evaluates the expression e to the value $v \in \mathbf{Val}$. Thus, $\Sigma(\mathbf{n}) \vdash e \downarrow v$ means that the choreographic store Σ evaluates expression e to value $v \in \mathbf{Val}$ in role $\mathbf{n}$. If $\mathbf{n} \notin \text{dom}(\Sigma)$, we say that $\Sigma(\mathbf{n}) \vdash e \downarrow ()$, where $()$ is the unit value, used as a placeholder for a value that carries no information.

We assume, without defining it, a system $\vdash f(v_1, \dots, v_n) \downarrow v$ to derive expressions, whereby calling local function f with arguments $v_1, \dots, v_n$ evaluates to v , for all $v_1, \dots, v_n$.

[VAL] $\sigma \vdash v \downarrow v$ **[VAR]** $\sigma \vdash x \downarrow \sigma(x)$ **[CALL]** $\sigma \vdash f(e_1, \dots, e_n) \downarrow v \Leftrightarrow \sigma \vdash e_1 \downarrow v_1 \dots \sigma \vdash e_n \downarrow v_n \wedge \vdash f(v_1, \dots, v_n) \downarrow v$.

Response Store A response store $\Psi: \mathbf{J} \rightarrow \mathbb{R} \times \mathbf{Var}$ is a function that stores information for the serverless roles in a choreography about which role they should send a response to, and where this role should save the response. Stateless roles created without requiring a response do not appear in the response store.

Stateless roles within the domain of a response store must respond according to that store's information – and stateless roles outside the store's domain should not respond. Similarly to Σ , we use Ψ to provide information about stateless roles' termination. To handle the creation of new roles and their termination, we define a notation for updating response stores $\Psi[g \mapsto y@n](f) = \langle n, y \rangle$, if $f = g$; $\Psi(f)$, otherwise. We also denote with $\Psi(f) \mapsto y@n$ that f must respond to n , which will save the result in variable y . Similarly to stores, $\Psi \setminus f$ denotes Ψ after removing f from its domain.

Process Names To define the semantics, we need to refer to the roles which appear in instructions, choreographies, and transition labels. Hence, we define the pn function, which maps instructions, choreographies, lists of parameters, and transition labels to the set of role names that appear within them. For brevity, we omit the formal definition of pn , reported in the companion technical report [15].

Name Substitution Lastly, we define a name-substitution operator for roles in choreographies and lists of parameters, such that $C[1/m]$ corresponds to choreography C where 1 replaces every instance of m . We use a similar notation for pairs and sets of roles, with a condition that prevents incorrect positional replacements of $\emptyset$ roles – cf. [15].

Labelled Transition System FaaSChalCore's semantics is defined as follows. The set of states is the set of tuples $\langle C, \Sigma, \Psi, \mathcal{C} \rangle$, called *choreographic configurations*. We define *initial* a choreographic configuration where C corresponds to the body of procedure $\text{main}(F||)$, as contained in $\mathcal{C}$, and Σ and Ψ are both empty. The transition relation is the smallest relation that satisfies the inference rules defined in Fig. 4, such that the set of transition labels μ is the set of all possible labels obtained by instantiating the variable names that appear in the inference system. For compactness, when elements of a configuration are immaterial to the rule, we replace them with $\cdot$. Rules [LOC], [COM], [SEL] are standard [22]; [FTC] is similar to [COM].

Rule [REQ] executes a serverless function request without asking for a response, which creates a new stateless role f , adding f to Σ 's domain and setting the value of x in f to v (the value sent by the requester). Rule [REQ-RES] is similar to [REQ], except that the requester expects a response. In this case, we store in Ψ that f must send its response to n , which will save it under its variable y . Rule [END] handles the termination of a stateless role without response, which essentially entails the removal of the role from Σ 's domain. Rule [END-RES] is similar to [END] except that f sends to n the value v from the evaluation of expression e , saved in variable y .

Rule [CALL-FIRST] executes a procedure call for an initial role n . Note that we use $\circ$ to concatenate the procedure body with the remainder of the caller ($\circ$ is inductively defined: $\emptyset \circ C \triangleq C$ ($I; C'$) $\circ C \triangleq I; (C' \circ C)$). The first premise retrieves the parameters and body C_x of procedure x , while the second selects the initial role that enters the procedure. The next four premises prevent name capture when expanding the body of the procedure. In particular, we find the set H_x of “hidden” roles in the procedure's body, not declared as parameters. These roles are necessarily stateless, so we create a fresh set of stateless roles, H , of the same size, distinct from the existing roles (in Σ), from the rest of the choreography C , and the body of x ; the notation $a \# b$ is a shorthand for $a \cap b = \emptyset$. The last premise expands the procedure's body – replacing the original hidden roles in H_x with the new roles in H (we slightly abuse notation since these are unordered sets) and the formal parameters with the actual

$$\begin{array}{c}
\text{[LOC]} \quad \frac{\Sigma(\mathbf{n}) \vdash e \downarrow v}{\frac{\langle x @ \mathbf{n} = e @ \mathbf{n}; C, \Sigma, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{n}} \langle C, \Sigma[x @ \mathbf{n} \mapsto v], \cdot, \cdot \rangle}} \\
\text{[COM]} \quad \frac{\Sigma(\mathbf{n}) \vdash e \downarrow v}{\frac{\langle e @ \mathbf{n} \rightarrow x @ \mathbf{q}; C, \Sigma, \cdot, \cdot \rangle \xrightarrow{v @ \mathbf{n} \rightarrow x @ \mathbf{q}} \langle C, \Sigma[x @ \mathbf{q} \mapsto v], \cdot, \cdot \rangle}} \\
\text{[SEL]} \quad \frac{}{\langle \mathbf{n} \rightarrow \mathbf{q}[\mathbf{L}]; C, \cdot, \cdot, \cdot \rangle \xrightarrow{\mathbf{n} \rightarrow \mathbf{q}[\mathbf{L}]} \langle C, \cdot, \cdot, \cdot \rangle} \\
\text{[FTC]} \quad \frac{\Sigma(\mathbf{q}) \vdash e \downarrow v}{\frac{\langle \mathbf{n} \leftrightarrow e @ \mathbf{q} > x @ \mathbf{n}; C, \Sigma, \cdot, \cdot \rangle \xrightarrow{\mathbf{n} \leftrightarrow e @ \mathbf{q} > x @ \mathbf{n}} \langle C, \Sigma[x @ \mathbf{n} \mapsto v], \cdot, \cdot \rangle}} \\
\text{[REQ]} \quad \frac{\Sigma(\mathbf{n}) \vdash e \downarrow v}{\frac{\langle e @ \mathbf{n} \rightarrow x @ \mathbf{f}; C, \Sigma, \cdot, \cdot \rangle \xrightarrow{v @ \mathbf{n} \rightarrow x @ \mathbf{f}} \langle C, \Sigma[x @ \mathbf{f} \mapsto v], \cdot, \cdot \rangle}} \\
\text{[REQ-RES]} \quad \frac{\Sigma(\mathbf{n}) \vdash e \downarrow v}{\frac{\langle e @ \mathbf{n} \iff x @ \mathbf{f} > y @ \mathbf{n}; C, \Sigma, \Psi, \cdot \rangle \xrightarrow{v @ \mathbf{n} \iff x @ \mathbf{f}} \langle C, \Sigma[x @ \mathbf{f} \mapsto v], \Psi[\mathbf{f} \mapsto y @ \mathbf{n}], \cdot \rangle}} \\
\text{[END]} \quad \frac{}{\langle \text{end } \mathbf{f}; C, \Sigma, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{f}} \langle C, \Sigma \setminus \mathbf{f}, \cdot, \cdot \rangle} \\
\text{[END-RES]} \quad \frac{\Psi(\mathbf{f}) \mapsto y @ \mathbf{n} \quad \mathbf{n} \in \mathbf{R} \cup \text{dom}(\Sigma) \quad \Sigma(\mathbf{f}) \vdash e \downarrow v}{\langle \text{end } e @ \mathbf{f} \rightarrow \mathbf{n}; C, \Sigma, \Psi, \cdot \rangle \xrightarrow{\mathbf{n} \leftarrow v @ \mathbf{f}} \langle C, \Sigma[y @ \mathbf{n} \mapsto v] \setminus \mathbf{f}, \Psi \setminus \mathbf{f}, \cdot \rangle} \\
\text{[CALL-FIRST]} \quad \frac{\begin{array}{l} \chi(F_{\mathbf{x}} | N_{\mathbf{x}} | T_{\mathbf{x}}) = C_{\mathbf{x}} \in \mathcal{C} \quad \mathbf{n} \in \text{pn}(F \cup N \cup T) \quad H_{\mathbf{x}} = \text{pn}(C_{\mathbf{x}}) \setminus \text{pn}(F_{\mathbf{x}} \cup N_{\mathbf{x}} \cup T_{\mathbf{x}}) \quad H \subseteq \mathbf{H} \quad |H| = |H_{\mathbf{x}}| \\ H \# \text{dom}(\Sigma) \cup \text{pn}(C) \cup \text{pn}(C_{\mathbf{x}}) \quad \langle (F \cup N) | T : \chi(F | N | T); C_{\mathbf{x}}[H, F, N, T / H_{\mathbf{x}}, F_{\mathbf{x}}, N_{\mathbf{x}}, T_{\mathbf{x}}]_{\mathbf{g}} C, \Sigma, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{n}} \langle C', \Sigma, \cdot, \cdot \rangle \end{array}}{\langle \chi(F | N | T); C, \Sigma, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{n}} \langle C', \Sigma, \cdot, \cdot \rangle} \\
\text{[COND-TRUE]} \quad \frac{\Sigma(\mathbf{n}) \vdash e \downarrow \text{true}}{\langle \text{if } e @ \mathbf{n} \text{ then } \{C_1\} \text{ else } \{C_2\}; C, \Sigma, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{n}} \langle C_1_{\mathbf{g}} C, \Sigma, \cdot, \cdot \rangle} \\
\text{[COND-FALSE]} \quad \text{mirror of [COND-TRUE], omitted} \\
\text{[CALL-ENTER]} \quad \frac{\mathbf{n} \in \text{pn}(N_P \cup T_P) \quad (N_P \cup T_P) \setminus \mathbf{n} \neq \emptyset}{\frac{\langle N_P | T_P : \chi(F | N | T); C, \cdot, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{n}} \langle N_P \setminus \mathbf{n} | T_P \setminus \mathbf{n} : \chi(F | N | T); C, \cdot, \cdot, \cdot \rangle}} \\
\text{[CALL-LAST]} \quad \frac{\text{pn}(N_P \cup T_P) = \{\mathbf{n}\}}{\langle N_P | T_P : \chi(F | N | T); C, \cdot, \cdot, \cdot \rangle \xrightarrow{\tau @ \mathbf{n}} \langle C, \cdot, \cdot, \cdot \rangle} \\
\text{[DELAY]} \quad \frac{\langle C, \Sigma, \Psi, \cdot \rangle \xrightarrow{\mu} \langle C', \Sigma', \Psi', \cdot \rangle \quad \text{pn}(I) \# \text{pn}(\mu)}{\langle I; C, \Sigma, \Psi, \cdot \rangle \xrightarrow{\mu} \langle I; C', \Sigma', \Psi', \cdot \rangle} \\
\text{[DELAY-COND]} \quad \frac{\langle C_1, \Sigma, \Psi, \cdot \rangle \xrightarrow{\mu} \langle C'_1, \Sigma', \Psi', \cdot \rangle \quad \langle C_2, \Sigma, \Psi, \cdot \rangle \xrightarrow{\mu} \langle C'_2, \Sigma', \Psi', \cdot \rangle \quad \mathbf{n} \notin \text{pn}(\mu)}{\langle \text{if } e @ \mathbf{n} \text{ then } \{C_1\} \text{ else } \{C_2\}; C, \Sigma, \Psi, \cdot \rangle \xrightarrow{\mu} \langle \text{if } e @ \mathbf{n} \text{ then } \{C'_1\} \text{ else } \{C'_2\}; C, \Sigma', \Psi', \cdot \rangle}
\end{array}$$

Fig. 4. Rules of FaaSChalCore semantics.

parameters – and reduces it with an action on $\mathbf{n}$, applying either rule [CALL-ENTER] or [CALL-LAST], as described below. Since the replacements on lists are applied sequentially, we consider procedure declarations up to the renaming of their formal parameters, which we assume distinct from the roles in the configuration, so that these replacements do not interfere with one another. Similarly, since rules [DELAY] and [DELAY-COND] (described below) let a call execute before the instructions that precede it, or within the branches of a conditional, the premises of [CALL-FIRST] cannot inspect all the roles of the choreography; hence, we assume that the roles in H are distinct from all the roles that occur in the configuration. Rule [CALL-ENTER] handles the entrance of a role into the body of a started procedure by simply removing it from the list of roles that have yet to enter. We abuse the notation when writing $T_P \setminus \mathbf{n}$, which means that we replace each $(\mathbf{f}, \mathbf{n}) \in T_P$ with $(\mathbf{f}, \emptyset)$, each $(\mathbf{n}, \mathbf{m}) \in T_P$ with $(\emptyset, \mathbf{m})$, and we remove each $(\mathbf{n}, \emptyset) \in T_P$ and $(\emptyset, \mathbf{n}) \in T_P$. Rule [CALL-LAST] executes the entrance of the last role into a started procedure by removing the runtime term.

The other rules are standard: [COND-TRUE] and [COND-FALSE] (complementary, omitted) evaluate expression e at role $\mathbf{n}$, and execute the related branch,

while $\llbracket \text{DELAY} \rrbracket$ and $\llbracket \text{DELAY-COND} \rrbracket$ support out-of-order execution instructions in sequences and within if statements. These last two rules allow interleaved parallel execution of actions that appear after a given instruction if they regard different roles – if statements are a bit more involved, since we need to make sure that the same out-of-order action can happen in both branches and does not concern the guard evaluator.

3 Static Analysis

Recalling the examples in Section 2, Fig. 3 reports a wrong workflow composition (the one on the left). There, a serverless function $\mathbf{f}$ created a second function, $\mathbf{g}$, which had to respond back to $\mathbf{f}$ but could not do so because $\mathbf{f}$ terminated before $\mathbf{g}$ could send back its response. In this section, we define a static analysis discipline that rejects these kinds of ill-behaved programs. Before formalising the analysis rules, we define “bad” choreographic configurations as containing dangling functions.

Definition 1 (Dangling Functions). *Given a choreographic configuration $\langle C, \Sigma, \Psi, \mathcal{C} \rangle$, it contains a dangling function if there exists $\mathbf{f} \in \mathbf{R}$ such that either:*

- $\mathbf{f}$ is a leak function, i.e., $C = \mathbf{0} \wedge \mathbf{f} \in \text{dom}(\Sigma) \cup \text{dom}(\Psi)$; or
- $\mathbf{f}$ is an orphan function, i.e., $\mathbf{f} \in \text{dom}(\Psi) \wedge \Psi(\mathbf{f}) \mapsto \mathbf{x} @ \mathbf{n} \wedge \mathbf{n} \notin (\mathbf{R} \cup \text{dom}(\Sigma))$.

Let $\xrightarrow{\vec{\mu}}$ be the reflexive, transitive closure of $\xrightarrow{\mu}$, with $\vec{\mu}$ sequence of transition labels.

Definition 2 (Dangling-Function-Free Configurations). *Let $\langle C, \Sigma, \Psi, \mathcal{C} \rangle$ be an initial choreographic configuration. We call $\langle C, \Sigma, \Psi, \mathcal{C} \rangle$ dangling-function-free if $\forall \vec{\mu} . \langle C, \Sigma, \Psi, \mathcal{C} \rangle \xrightarrow{\vec{\mu}} \langle C', \Sigma', \Psi', \mathcal{C}' \rangle$, $\langle C', \Sigma', \Psi', \mathcal{C}' \rangle$ has no dangling functions.*

Note that, since FaaSChalCore is parametric with respect to expression evaluation, it is generally undecidable to verify whether an initial configuration is dangling-function-free. Thus, we propose a static analysis that characterises a decidable subset of initial configurations, called *well-sorted*, which are guaranteed to be dangling-function-free under every interpretation of the expression-evaluation function.

3.1 Well-sortedness

We start by introducing two functions that allow us to handle sets of tuples of the kind $D \subseteq S_1 \times \dots \times S_i \times \dots \times S_n$, which are the projection π_i and its preimage $\pi_{D,i}^{-1}$:

$$\pi_i(D) = \{ x_i \mid (x_1, \dots, x_i, \dots, x_n) \in D \} \quad \pi_{D,i}^{-1}(x) = \{ (x_1, \dots, x_i, \dots, x_n) \in D \mid x_i = x \}$$

The notation $\langle \mathcal{C}, \mathbf{F}, \mathbf{N}, \mathbf{T}, <, \mathbf{R}, C \rangle \checkmark$ means that C is well-sorted in the context of the set of procedure definitions $\mathcal{C}$, the set of stateful roles $\mathbf{F} \subseteq \mathbf{R}$, the set of non-terminating roles $\mathbf{N} \subseteq \mathbb{R} \setminus \mathbf{F}$, and the set of terminating roles $\mathbf{T} \subseteq (\mathbf{R} \setminus \mathbf{N}) \times (\mathbb{R} \cup \{\mathbf{0}\})$, ranged over by $(\mathbf{f}, \mathbf{s})$ and $(\mathbf{g}, \mathbf{t})$, the termination order $<: \subseteq (\mathbf{R} \setminus \mathbf{N}) \times \mathbb{R}$, and the set of stateless roles $\mathbf{R} \supseteq \mathbf{N} \cup \pi_1(\mathbf{T})$ in its scope. We use $\text{def}(\mathbf{n})$ as a shorthand for $\mathbf{n} \in \mathbf{F} \cup \pi_1(\mathbf{T})$, meaning that a role is defined if it is a stateful or nonterminating role which the choreography can include, or if it is the first element of a terminating pair.

$\forall (f, s) \in T. f \neq s$	a terminating role cannot be its own creator
$\forall s \in \pi_2(T). \text{def}(s) \vee s = \emptyset$	the second element in each terminating pair is either a role in FUN , the first element in a terminating pair, or $\emptyset$.
$\nexists \{(f, s), (f, t)\} \subseteq T. s \neq t$	a terminating role in T cannot have different creators
$<: = (T \setminus \pi_{T,2}^{-1}(\emptyset))^+$	the termination order corresponds to the transitive closure of the pairs with a respondent terminating role
$\forall (f, n) \in (T \setminus \pi_{T,2}^{-1}(\emptyset)). n \not<: f$	no responding stateless role outlasts its creator

Fig. 5. Well-sortedness: assumed invariants on F, N, T and $<:$.

We define the inference system in Fig. 6 to derive these statements. As done in Fig. 4, we omit elements of the context that are immaterial to the rule. We assume the invariants in Fig. 5, which are required at the root of derivations and maintained by the inference rules (cf. [15]).

Henceforth, we consider the complete form of $\mathcal{C}$, as a set of pairs $(X(F_X | N_X | T_X) = C_X, <:_X) \in \mathcal{C}$, where each procedure definition is paired with its corresponding termination order. Formally, given a procedure declaration $\text{def } X(F_X | N_X | T_X) : (O_X)\{C_X\}$, we define $<:_X = (O_X \cup (T_X \setminus \pi_{T_X,2}^{-1}(\emptyset)))^+$, i.e., $<:_X$ is the transitive closure of the non- $\emptyset$ second-element pairs in O_X and T_X .

Rule $[\text{WSCLOSE}]$ states that the terminated choreography is well-sorted when all terminating roles have actually terminated (i.e., T is empty). Nonterminating stateless roles can exist, which supports well-sortedness for procedure definitions.

$[\text{WSLOC}]$ checks that the involved role is defined in the choreography and that the continuation is well-sorted. Rules $[\text{WSCOM}]$, $[\text{WSSEL}]$, and $[\text{WSFTC}]$ are similar, but we make sure that the receiver (q) is stateful and distinct from the sender (n).

A well-sorted request, as per $[\text{WSREQ}]$, must involve a defined role (n), and a role not in the current scope (f). The continuation, given the additional terminating pair $(f, \emptyset)$, must be well-sorted. The conditions for request responses are similar, in $[\text{WSREQ-RES}]$, except that the additional terminating pair is (f, n) and the extended order $<:$, closed under transitivity (superscript $^+$), includes (f, n) .

Rule $[\text{WSEND}]$ states that a termination without a response is well-sorted if $(f, \emptyset)$ is a terminating pair, f can terminate ($f \notin \pi_2(<:)$ means that f is not waiting for a response from any terminating role), and the continuation is well-sorted, with $(f, \emptyset)$ removed from the terminating pairs. Similarly, $[\text{WSEND-RES}]$ checks the termination of request-responses, removing (f, n) from the terminating pairs and all (f, m) from $<:$.

For conditionals ($[\text{WSCOND}]$), we check that the guard's evaluator is defined. Moreover, since roles can terminate in branches, we partition the stateless roles between the sets of terminating T' and nonterminating D ones. In forming D and T' , we impose them to be compatible (predicate comp) with $<:$. The seventh and eighth premises check that each branch is well-sorted by treating the elements of D as nonterminating (N' and $<:'$) and those in T' as terminating. Since we assume that the roles in T' terminate within the branches, we check the continuation by removing their pairs from $<:$ and extend the scope R to contain the stateless roles that appear in the branches.

Rule $[\text{WSCALL}]$ checks procedure calls. In the rule, the first premise retrieves the definition of the formal parameters of the procedure. The next three premises

$$\begin{array}{c}
\frac{[WS_{CLOSE}]}{\frac{T = \emptyset}{\langle \cdot, \cdot, T, \cdot, \cdot, 0 \rangle \checkmark}} \quad \frac{[WS_{LOC}]}{\frac{\text{def}(n) \quad \langle \cdot, F, N, T, \cdot, \cdot, C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, x @ n = e @ n; C \rangle \checkmark}} \quad \frac{[WS_{COM}]}{\frac{\text{def}(n) \quad q \in F \quad n \neq q \quad \langle \cdot, F, N, T, \cdot, \cdot, C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, e @ n \rightarrow x @ q; C \rangle \checkmark}} \\
\\
\frac{[WS_{SEL}]}{\frac{\text{def}(n) \quad q \in F \quad n \neq q \quad \langle \cdot, F, N, T, \cdot, \cdot, C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, n \rightarrow q[L]; C \rangle \checkmark}} \quad \frac{[WS_{FTC}]}{\frac{\text{def}(n) \quad q \in F \quad n \neq q \quad \langle \cdot, F, N, T, \cdot, \cdot, C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, n \leftrightarrow e @ q \triangleright x @ n; C \rangle \checkmark}} \\
\\
\frac{[WS_{REQ}]}{\frac{\text{def}(n) \quad f \notin R \quad \langle \cdot, F, N, T \cup \{(f, 0)\}, \cdot, R \cup \{f\}, C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, R, e @ n \rightarrow x @ f; C \rangle \checkmark}} \quad \frac{[WS_{REQ-RES}]}{\frac{\text{def}(n) \quad f \notin R \quad \langle \cdot, F, N, T \cup \{(f, n)\}, \cdot, \langle \cdot, \cup \{(f, n)\} \rangle^+, R \cup \{f\}, C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, R, e @ n \iff x @ f \triangleright y @ n; C \rangle \checkmark}} \\
\\
\frac{[WS_{END}]}{\frac{(f, 0) \in T \quad f \notin \pi_2(\cdot) \quad \langle \cdot, \cdot, T \setminus \{(f, 0)\}, \cdot, \cdot, C \rangle \checkmark}{\langle \cdot, \cdot, T, \cdot, \cdot, \text{end } f; C \rangle \checkmark}} \quad \frac{[WS_{END-RES}]}{\frac{(f, n) \in T \quad f \notin \pi_2(\cdot) \quad \langle \cdot, \cdot, T \setminus \{(f, n)\}, \cdot, \cdot, \pi_{\cdot, 1}^{-1}(f), C \rangle \checkmark}{\langle \cdot, \cdot, T, \cdot, \cdot, \text{end } e @ f \rightarrow n; C \rangle \checkmark}} \\
\\
\frac{[WS_{COND}]}{\frac{\text{def}(n) \quad D \subseteq T \quad T' = T \cup D \quad \text{comp}(D, T', \cdot) \quad \langle \cdot, \cdot, T', \cdot, \cdot, \pi_{\cdot, 1}^{-1}(\pi_1(D)) \rangle \quad N' = N \cup \pi_1(D) \quad \langle \cdot, F, N', T', \cdot, \cdot, R, C_1 \rangle \checkmark}{\langle \cdot, F, N', T', \cdot, \cdot, R, C_2 \rangle \checkmark \quad R' = (\text{pn}(C_1) \cup \text{pn}(C_2)) \cap \mathcal{H} \quad \langle \cdot, F, N, D, \cdot, \cdot, \pi_{\cdot, 1}^{-1}(\pi_1(T')) \rangle, R \cup R', C \rangle \checkmark}{\langle \cdot, F, N, T, \cdot, \cdot, R, \text{if } e @ n \text{ then } \{C_1\} \text{ else } \{C_2\}; C \rangle \checkmark}} \\
\\
\frac{[WS_{CALL}]}{\frac{T \subseteq T \quad \text{comp}(T \setminus T, T, \cdot) \quad \langle X(F_X | N_X | T_X) = C_X, \cdot, \cdot \rangle \in \mathcal{C} \quad F \subseteq F \quad N \subseteq N \cup F \cup \pi_1(T) \quad \langle \mathcal{C}, F, N, T \setminus T, \cdot, \cdot, C \rangle \checkmark}{\langle \mathcal{C}, F, N, T, \cdot, \cdot, X(F | N | T); C \rangle \checkmark}}
\end{array}$$

Predicates

$$\text{comp}(T, T', \cdot) \iff \{ (f, g) \mid f \in \pi_1(T) \wedge g \in \pi_1(T') \} \cap \cdot = \emptyset$$

$$\frac{\forall i, j, p_i \neq n_j \quad \forall i, j, n_i \neq f_j \quad \forall i, f_i \neq s_i \quad \forall i, s_i = 0 \iff s_i^x = 0 \quad \forall i, j, p_i^x = s_j^x \Rightarrow p_i = s_j \quad \forall i, j, n_i^x = s_j^x \Rightarrow n_i = s_j}{\frac{\forall i, j, s_i^x = s_j^x \Rightarrow s_i = s_j \quad \forall i, j, s_i^x = f_j^x \Rightarrow s_i = f_j \quad \forall i, j, s_i = f_j \Rightarrow f_i^x <_X f_j^x \quad \forall i, j, f_i <_X f_j \Rightarrow f_i^x <_X f_j^x}{\text{cons}(\langle p_1, \dots, p_n \rangle, \langle p_1^x, \dots, p_n^x \rangle, \langle n_1, \dots, n_m \rangle, \langle n_1^x, \dots, n_m^x \rangle, \langle (f_1, s_1), \dots, (f_l, s_l) \rangle, \langle (f_1^x, s_1^x), \dots, (f_l^x, s_l^x) \rangle, \cdot, \cdot) \checkmark}}$$

Fig. 6. Inference system for the well-sortedness.

check that each parameter list appears in the corresponding sets of defined roles. In particular, T must contain a set of terminating pairs that is compatible with $\cdot$ (similarly to $[WS_{COND}]$). Briefly, predicate cons checks that the number and position of actual and formal parameters are correct, that all parameters are distinct (except when one of the involved parameters is the second elements of terminating pairs), that when two formal parameters are the same the corresponding actual parameters are the same, that $\cdot$ contains the pairs corresponding to the actual parameters, and that the declared termination order $\cdot$ is compatible with $\cdot$ ⁷. The last premise checks that the continuation is well-sorted without the terminating pairs in T – notably, we do not touch R because, when calling a procedure, all roles created within the procedure’s body are replaced with fresh roles, which cannot interfere with the rest of the program.

⁷ Note that $\forall i, j, s_i = f_j \Rightarrow f_i^x <_X f_j^x$ in the definition of cons allows us to omit the condition $s_i <_X f_j \Rightarrow f_i^x <_X f_j^x$. Indeed, if a call is well-sorted, and we have that $s_i <_X f_j$, then there must be some k such that $f_k = s_i$. As a consequence, we have $f_i^x <_X f_k^x$. Since $s_i = f_k$, we also have $f_k <_X f_j$, and we get that $f_k^x <_X f_j^x$. Since we assume $\mathcal{C}$ well-sorted, we have that $\cdot$ is a transitive, strict partial order, which gives us $f_i^x <_X f_j^x$.

3.2 Well-sortedness of Programs and Properties

Given the definition of well-sortedness, we can proceed to define well-sortedness of FaaSChalCore programs and their properties.

To define well-sorted initial choreographic configurations (Definition 5), we first define well-sortedness of a choreography (Definition 3) and of a set of procedure definitions (Definition 4).

Definition 3 (Well-sorted Choreographies). *Let $\mathcal{C}$ be a set of procedure definitions, $N \subseteq \mathbb{R}$, $T \subseteq (\mathfrak{A} \setminus M) \times (\mathbb{R} \cup \{\emptyset\})$, and $R \supseteq N \cup \pi_1(T)$, we say that $C \in \mathbf{FaaSChalCore}$ is well-sorted in the context of $\mathcal{C}, N, T$, and R if $\exists F \subseteq R$ such that $\langle \mathcal{C}, F, N, T, < := (T \setminus \pi_{T,2}^{-1}(\emptyset))^+, R, C \rangle \checkmark$ is derivable, and the requirements in Fig. 5 hold.*

Definition 4 (Well-sorted Procedures). *Let $\mathcal{C}$ be a set of procedure definitions. $\mathcal{C}$ is well-sorted if, $\forall (\text{def } X(F_X | N_X | T_X) : (O_X)\{C_X\}, < :_X) \in \mathcal{C}$ these statements hold:*

- $\langle \mathcal{C}, F_X, N_X, T_X, < :_X, N_X \cup (pn(T_X) \setminus R), C_X \rangle \checkmark$ is derivable;
- $(\forall i \neq j. p_i^X \neq p_j^X) \wedge (\forall i \neq j. n_i^X \neq n_j^X) \wedge (\forall i \neq j. f_i^X \neq f_j^X)$;
- $(\forall i, j. p_i^X \neq n_j^X) \wedge (\forall i, j. n_i^X \neq f_j^X)$;
- $< :_X$ is a strict partial order;
- $|pn(F_X \cup N_X \cup T_X)| \geq 1$, i.e., there is at least one formal parameter;
- $pn(F_X \cup N_X \cup T_X) \subseteq pn(C_X)$, i.e., the procedure body mentions all formal parameters.

The first requirement checks the body of X in the context of its formal parameters, with the declared termination order $< :_X$, which can be more restrictive than the one resulting from T_X . Note that this context does not satisfy the invariants in Fig. 5; in particular, it violates the second invariant when a terminating parameter responds to a formal parameter that is not in $F_X \cup N_X \cup \pi_1(T_X)$. However, this violation is not an issue because, when calling the procedure within a well-sorted choreography, roles defined in the calling context replace the formal parameters (cf. [15]).

Finally, we lift well-sortedness to initial choreographic configurations (Definition 5), which lets us state the main result of our analysis (Theorem 1): every well-sorted initial configuration is dangling-function-free (Definition 2).

Definition 5 (Well-sorted Initial Choreographic Configurations). *Given the initial choreographic configuration $\langle C, \Sigma, \Psi, \mathcal{C} \rangle$, we say that it is well-sorted if $\mathcal{C}$ is well-sorted and C is well-sorted in the context of $\mathcal{C}$, $N = \emptyset$, $T = \emptyset$, and $R = \emptyset$.*

Theorem 1. *A well-sorted initial configuration $\langle C, \Sigma, \Psi, \mathcal{C} \rangle$ is dangling-function-free.*

Proof outline The proof, fully detailed in a companion technical report [15], extends well-sortedness to runtime configurations, where N and T track the stateless roles alive in Σ and the responses pending in Ψ . The proof's core hinges on demonstrating that every transition preserves well-sortedness, updating T with the role it creates or terminates – for conditionals and procedure calls, we rely on the fact that composing a branch, or an expanded procedure body, with its continuation preserves well-sortedness. Since an initial well-sorted configuration has $N = T = \emptyset$, every configuration it reaches is well-sorted with $N = \emptyset$. Hence, there are no leak functions, since $\mathbf{0}$ is well-sorted only when T is empty, and no orphan functions, since the invariants in Fig. 5 ensure that every pending response is directed to a stateful or alive role.

4 Related Work

FaaSChalCore sits at the intersection of choreographic programming and formal models of serverless computing. Our main inspiration is in conceptual work on choreographic programming, which, in turn, finds its original application area in Service-Oriented Computing, with applications to web- [26] and micro-services [7, 9, 10, 16]. Since FaaSChalCore is, at heart, a formal model of service composition, we first position it w.r.t. the main formal traditions of the web-service literature and then discuss the choreographic and serverless proposals closest to ours.

Service composition has been formalised with process algebra – both to encode existing languages [13] and to design dedicated calculi [18] – with automata – as conversation protocols among finite-state peers [4], studying their realisability, or as interfaces constraining the environment [1] – and with Petri nets – from workflow models of business processes to interaction nets lifted to choreographies [11]. While conformance theories relate the global and the local view a posteriori [6], choreographic programming derives endpoints, so conformance holds by construction. The peculiarity of serverless not covered by the considered models is the presence of ephemeral participants created on demand. The unbounded number of these participants makes FaaSChalCore programs infinite-state over a dynamic topology, ruling out the exhaustive analyses that finite-state automata and nets make decidable – which motivates our syntax-directed, compositional inference system for checking lifecycle invariants.

Within the choreographic line of research, FaaSChal [23] is the closest work to ours, from which we drew inspiration to develop our choreographic calculus. In FaaSChal’s proposal, the authors introduce a choreographic language tailored to FaaS (actually, FaaSChal has both stateful and stateless roles, but the authors do not directly mention targeting hybrid systems). Briefly, the two proposals share the same high-level goal of applying CP to make the interaction of (hybrid) FaaS applications explicit and derive correct-by-construction local behaviour from a global specification. The difference between the two contributions is that FaaSChal’s authors present a conceptual proposal with example-driven exploration of the design space, whereas ours is a core calculus with a formal operational semantics and formal proofs of the correctness of a static analysis that rules out wrong FaaSChalCore programs – creation, termination, response obligations, and termination ordering are explicit in FaaSChalCore. Our model also resembles the more abstract one of Procedural Choreographies [8] (PC), which includes full procedural abstraction, unbounded process creation, and name mobility. PC, like FaaSChalCore, aims at correctness-by-construction and modularity; however, our calculus introduces constructs and a static analysis for explicit role termination, response obligations, and termination order, which have no direct counterpart in PC.

Widening our scope on the literature, we find work on calculi for serverless, spearheaded by Giallorenzo et al. [14] and Jangda et al. [19], who present models of FaaS serverless computing, capturing the lifecycle of function instances, making explicit the relationships between functions, triggers, and the asynchronous execution model of FaaS platforms. More recent work tackled stateful serverless [5], presenting a high-level semantics and proving correspondence with concrete execution models that account for compute-storage separation and record-replay. In a complementary way, Kallas et al. [21] worked on the correct execution of microservice applications on serverless plat-

forms, introducing a framework that makes the interaction between microservice structure and serverless execution explicit and demonstrated how to preserve correctness in that setting. Compared with these efforts, FaaSChalCore is not primarily a model of the provider or runtime substrate but a choreographic serverless execution model – where the focus is on component interactions rather than single-function actions – equipped with a static checking discipline to rule out wrong (choreographic) programs.

5 Conclusion

We introduced FaaSChalCore, a choreographic calculus for programming and verifying hybrid microservice-serverless systems. FaaSChalCore captures the creation, coordination, and termination of stateless serverless functions alongside long-running stateful microservices within a single global specification, treating statefulness and statelessness as first-class traits of roles, and threading them through procedure parameters, workflow, and termination orders. We presented and proved correct a static-analysis discipline that guarantees that well-sorted FaaSChalCore programs are free of FaaS-specific errors that arise from mismatched creation, termination, and response obligations. Note that, since in FaaSChalCore procedures can recur, and each call allocates fresh stateless roles, FaaSChalCore programs have an infinite configuration space, making reachability analysis for dangling-function freedom (Definition 2) undecidable. For this reason, we define well-sortedness as a static analysis discipline that disregards specific configurations. The core of our analysis is the well-sortedness check defined in Fig. 6. Notice that it corresponds to a deterministic check to be executed on the abstract syntax tree of the programs, with precisely one possible rule for each syntactic construct. Let $|P|$ be the program’s size, the number of rules to be checked is $O(|P|)$. The additional preconditions to be checked by each rule require constant time (the most complex conditions are set membership checks) excluding the rules [WSCOND] and [WSCALL] . These last two rules perform set inclusion checks on sets of (pairs of) roles and checks of distinction between pairs of roles. Let $r \leq |P|$ be the number of roles occurring in the program, both checks above take $O(r^2)$ time. Hence, deciding well-sortedness takes $O(|P| \cdot r^2)$ time – cubic in $|P|$ in the worst case, linear for a fixed set of roles. Notably, the analysis scales with the size of the source, where procedures are checked once, in isolation, against their signatures, so r counts the roles in a single procedure’s scope – independently of the functions spawned at runtime.

We conclude with an example showcasing the differences w.r.t. existing CP proposals: we rewrite in FaaSChalCore the merge sort of Choral [17, Sec. 3.3], a state-of-the-art choreographic language, inviting the reader to refer to it to contextualise the comparison. Briefly, in the `main` procedure (on the left), the only role/parameter – the stateful process `p` – initialises a local variable (`arr`) with an array to be sorted and enters procedure `MergeSort` (on the right), i.e., `a` is replaced with `p`. Note that, while `p` is conversational in `main`, we pass it to `MergeSort` as an inquiring one, since (line 1) `MergeSort` accepts one formal inquiring nonterminating parameter (`a`), either stateful (like `p`) or stateless. In `MergeSort`, if the array is longer than 1 (line 2), `a` starts two serverless functions (lines 3–4), `f` and `g`, sending each half of the array. These recursively call `MergeSort` (as stateless nonterminating parameters), then terminate

```

1  def main(p||){
2    arr@p=get_arr()@p;
3    MergeSort(|p|);
4  }

1  def MergeSort(|a|){
2    if gt(size(arr), 1)@a then {
3      first_half(arr)@a <=> arr@f > arr1@a;
4      second_half(arr)@a <=> arr@g > arr2@a;
5      MergeSort(|f|); MergeSort(|g|);
6      end arr@f -> a; end arr@g -> a;
7      arr@a=merge(arr1, arr2)@a;
8    } }

```

Fig. 7. A highly-parallelised implementation of choreographic merge sort.

(end) and respond to **a** with the sorted arrays (line 6), which **a** merges. Beyond readability, the example showcases how FaaSChalCore supports highly parallelised implementations: the recursive calls for **f** and **g** appear in sequence but happen in parallel, out-of-order execution deriving from rule [DELAY] (cf. Fig. 4), which captures the natural parallelism of actions over disjoint sets of processes.

Another aspect is that, although the conditional involves roles other than the guard’s evaluator (**a**), we need no selections to convey knowledge of choice (e.g., Choral’s `.select` [17]) since **f** and **g** exist only when **a** takes the `then` branch.

The parallelism above will become fully explicit once we define, following standard choreographic techniques, an endpoint projection for FaaSChalCore – whereby **f** and **g** would become independent FaaS functions executing concurrently. We leave it to future work, along with directions such as implementing a prototype checker integrated with a FaaS platform and extensions of the discipline on e.g., performance [23] and security [3].

Acknowledgements This work has been partially supported by the French ANR project SmartCloud ANR-23-CE25-0012.

References

- [1] Luca de Alfaro and Thomas A. Henzinger. “Interface automata”. In: *FSE 2001*. ACM, 2001, pp. 109–120.
- [2] Amazon Web Services. *Microservices with Lambda*. <https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/>. Acc. 07/2026.
- [3] Marco Bertoni et al. “Type-Based Enforcement of Non-Interference for Choreographic Programming”. In: *ICTCS 2026*. CEUR. CEUR-WS.org, 2026.
- [4] Tefvik Bultan et al. “Conversation specification: a new approach to design and analysis of e-service composition”. In: *WWW 2003*. ACM, 2003, pp. 403–410.
- [5] Sebastian Burckhardt et al. “Durable functions: semantics for stateful serverless”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–27.
- [6] Nadia Busi et al. “Choreography and Orchestration Conformance for System Design”. In: *COORDINATION 2006*. Vol. 4038. LNCS. Springer, 2006.
- [7] Marco Carbone and Fabrizio Montesi. “Deadlock-freedom-by-design: multiparty asynchronous global programming”. In: *POPL*. 2013, pp. 263–274.

- [8] Luís Cruz-Filipe and Fabrizio Montesi. “Procedural Choreographic Programming”. In: *FORTE 2017*. Vol. 10321. LNCS. Springer, 2017, pp. 92–107.
- [9] Mila Dalla Preda et al. “AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications”. In: *SLE 2014*. Vol. 8706. LNCS. Springer, 2014.
- [10] Mila Dalla Preda et al. “Dynamic Choreographies - Safe Runtime Updates of Distributed Applications”. In: *COORDINATION 2015*. Vol. 9037. LNCS. Springer, 2015, pp. 67–82.
- [11] Gero Decker and Mathias Weske. “Local Enforceability in Interaction Petri Nets”. In: *BPM 2007*. Vol. 4714. LNCS. Springer, 2007, pp. 305–319.
- [12] Nicola Dragoni et al. “Microservices: Yesterday, Today, and Tomorrow”. In: *PaUSE*. Springer, 2017, pp. 195–216.
- [13] Andrea Ferrara. “Web services: a process algebra approach”. In: *ICSOC 2004*. ACM, 2004, pp. 242–251.
- [14] Maurizio Gabbrielli et al. “No More, No Less - A Formal Model for Serverless Computing”. In: *COORDINATION 2019*. Vol. 11533. LNCS. Springer, 2019.
- [15] Saverio Giallorenzo et al. *A Choreographic Calculus for Programming Correct Microservice-Serverless Systems*. Tech. rep. 2026. DOI: 10.5281/zenodo.21415051.
- [16] Saverio Giallorenzo et al. “A model for correlation-based choreographic programming”. In: *PeerJ Comput. Sci.* 10 (2024), e1907.
- [17] Saverio Giallorenzo et al. “Choral: Object-oriented Choreographic Programming”. In: *ACM Trans. Program. Lang. Syst.* 46.1 (2024), 1:1–1:59.
- [18] Claudio Guidi et al. “ ω : A Calculus for Service Oriented Computing”. In: *ICSOC 2006*. Vol. 4294. LNCS. Springer, 2006, pp. 327–338.
- [19] Abhinav Jangda et al. “Formal foundations of serverless computing”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 149:1–149:26.
- [20] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. University of California, Feb. 2019.
- [21] Konstantinos Kallas et al. “Executing Microservice Applications on Serverless, Correctly”. In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 367–395.
- [22] Fabrizio Montesi. *Introduction to Choreographies*. Camb. Univ. Press, 2023.
- [23] Giuseppe De Palma et al. “Towards a Function-as-a-Service Choreographic Programming Language: Examples and Applications”. 2024.
- [24] David Recordon and Drummond Reed. “OpenID 2.0: a platform for user-centric identity management”. In: *WDIM 2006*. ACM, 2006, pp. 11–16.
- [25] Francesco Tusa et al. “Microservices and serverless functions - lifecycle, performance, and resource utilisation of edge based real-time IoT analytics”. In: *FGCS* 155 (2024), pp. 204–218.
- [26] W3C WS-CDL Working Group. *WS-CDL version 1.0*. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>. 2004.