Formalizing a Hoare Calculus for Choreographic Programming

Luís Cruz-Filipe
IT University Copenhagen, Denmark

Thomas Wulff Heissel
Department of Computer Science, University of Southern Denmark, Odense, Denmark

Abstract

Choreographic programming is a paradigm where developers write the global specification (called choreography) of a communicating system, and then a correct-by-construction distributed implementation is compiled automatically. Choreographies formalize the way many practitioners think about distributed protocols, and are a natural framework in which to prove properties of such protocols.

Previous work has introduced a Hoare calculus for reasoning about choreographies. In this article, we show how a formalization of that work in a theorem prover revealed several issues with the pen-and-paper development. We discuss the extent to which these issues can be fixed, and conclude with some considerations on the need for more formal verification of research results.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Automated reasoning

Keywords and phrases choreographic programming, theorem proving, Hoare calculus

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.15

Supplementary Material
Software (Formalization): `https://doi.org/10.5281/zenodo.20290519` [16]

Declaration about GenAI/LLM use Generative AI was not used in any form in the preparation of this article – neither to assist with the development of Rocq proofs, nor to aid in writing the article itself.

1 Introduction

Implementing communicating systems is a challenging task, as one needs to ensure that complementary communication actions (such as sending and receiving a message) that are executed by different components are correctly matched at runtime. Indeed, a significant fraction of bugs in concurrent software derive precisely from such actions being mismatched, making techniques for avoiding such issues extremely relevant in practice [28].

Choreographic programming [33] is a programming paradigm that addresses this challenge by combining high-level specifications of abstract protocols in an Alice-and-Bob notation with a compilation procedure that generates correct-by-construction distributed implementations [6, 9, 17, 20, 29, 31, 30]. Research on choreographic programming has included extending the expressivity of choreographic languages, implementing them, and formalizing general properties of compilation. A more recent line of research explores how to prove functional correctness properties about choreographies using ideas from Hoare calculi [8, 24].

The theory of choreographic programming is not without its challenges. The complexity of reasoning about formalisms where a single choreography instruction may be implemented by multiple instructions in the target language means that errors often occur. Even in a simple framework abstracting from computation, there are cases where proofs published in peer-reviewed articles have been found to be flawed, requiring corrections to their theories. While in most cases these adjustments amounted to correcting small details in proofs or deal with missing cases, some situations have required new proof strategies or even reformulating


© Luís Cruz-Filipe and Thomas Wulff Heissel;
licensed under Creative Commons License CC-BY 4.0

statements [11, 32, 35]. In exceptional situations, it has been discovered that key results actually did not hold [3, 4, 5, 19, 27]. This motivated the development of a library for reasoning about a core choreographic language [14], whth the aim of encouraging researchers to validate their results formally. This library has subsequently been extended with formalizations of additional results and extensions [11, 15].

This work

In this work, we follow the development of a Hoare logic for reasoning about choreographies from [8] and formalize it in the theorem prover Rocq, building upon a previously existing formalization of the underlying choreography language. The formalization reveals challenges with some of the results, and we show that some of them can be resolved with a slight tweaking of the original definitions. The main completeness result, however, turns out to be incorrect – and the feedback from the theorem prover can be used as guidance to build a counterexample. Nevertheless, we argue that our results and our development can still be used to simplify the development of proofs of properties of choreographic programs by automating part of the process.

Structure

We start by reviewing related work in Section 2. We then present the choreographic language and its formalization from [14] in Section 3, where we also briefly recall the main ideas behind Hoare logics. We present the syntax and semantics of the logic from [8] together with our formalization in Rocq in Sections 4 and 5. Section 6 includes a discussion of the results and the issues found in the formalization process, together with some simple illustrative examples. We draw some conclusions from our work in Section 7.

Our exposition assumes familiarity with interactive theorem proving. We use Rocq syntax throughout, but the work is intended to be accessible to non-Rocq experts.

2 Related Work

Hoare calculi for choreographies are a new area of research. To the authors' best knowledge, the only two preexisting attempts are the calculi from [8], on which the current contribution is based, and the choreography language from [24], which is intertwined with the Hoare logic.

Hoare logics for imperative programs have been formalized before in different theorem provers, such as Agda [25], Coq/Rocq [2, 26]. The Iris project [22] provides a toolbox for working with separation logic, which generalizes Hoare calculi, in Rocq.

Our work is based on the formalisation of Core Choreographies, originally introduced in [12, 13]. We use the version described in [15], which expands the reference presentation [14]. The current contribution extended this formalization with a semantics for head reductions (see Section 6), but otherwise has not been changed – substantiating the authors' claim that it has reached a sufficient level of maturity for being used as a library to reason about choreographies.

Kalas [34] is another choreographic programming language that can be compiled to CakeML, and has been formalized. Multiparty session types are more abstract versions of choreographies, which specify protocols abstracting from the actual details of the values being computed and communicated. Zooid [7] is a Coq library for checking whether message-passing code respects a given multiparty session type, while multiparty GV [23] is a similar contribution for a functional language. Subject reduction for multiparty session types has also been recently formalized in Coq [36].

3 Background

This work is based on an earlier formalization of CC in Coq [14], which has been updated to Rocq version 9.0. We briefly summarize it in this section in a simplified version, omitting some details that are immaterial for our work – in particular, the fact that communications have annotations (meant to be used in future compilation to real-world programming languages) and the fact that recursive procedure calls can include "silent" processes [15].

We write choreographies using a simplified Rocq notation for the purpose of readability: choreography and process terms are written by overloading the dot symbol (not allowed by the Rocq notation mechanism), while inductive definitions and inference rules are presented using the usual mathematical notation.

3.1 Core Choreographies

We start by summarizing the language of Core Choreographies (CC) together with its Rocq formalization [13].

Syntax. The syntax of CC is given by the following grammar.

```
C ::= η; C | If p.b Then C1 Else C2 | Call X | RT_Call X ps C | End
η ::= p.x ⟵ e | p.e ⟶ q.x | p ⟶ q[l]
```

A choreography `C` can be: a prefix η; `C`, executing the action η and continuing as `C`; a conditional `If p.b Then C1 Else C2`, where process `p` evaluates the Boolean expression `b` and continues as either `C1` or `C2`; a procedure call `Call X` invoking the procedure with name `X`; a runtime term `RT_Call X ps C`;[1] or the terminated choreography `End`. There are three kinds of prefixes: local assignments `p.x ⟵ e`, where process `p` evaluates expression `e` locally and stores the result in its variable `x`; value communications `p.e ⟶ q.x`, where the result of the evaluation is sent to process `q`, which stores it in *its* local variable `x`; or a label selection `p ⟶ q[l]`, where `p` sends one of the labels `left` or `right` to `q`.

The formalization aims at being as general as possible, and for this reason the underlying types of (Boolean) expressions, variables, values and procedure names are left as parameters. The only exception is the type of labels, which is hardcoded in order to simplify the development and allow for decidability results (see [14] for a discussion of this choice); this is also standard practice in other works on choreographies.

As a consequence, the type of choreographies is dependent on a signature (a record type with fields for all these datatypes), which we omit from the terms for simplicity. All types are assumed to be decidable (instances of a type `DecType`). The signature also includes two evaluation functions (from expressions to values and from Boolean expressions to Booleans) with a well-formedness requirement (they need to agree on extensionally equal states).

Choreographies are formalised in Rocq as the inductive type `Choreography`. For conciseness, we used fixed names for variables ranging over the different Rocq types in the formalization, as summarized in Table 1.

To execute a choreography, one must have information about the procedures that can be involved – specifically, their definition and the processes involved in them. This information is provided by a mapping from procedure names to pairs of process names and choreographies, also called *procedure definitions*.

[1] Runtime terms are a technical need for defining the semantics of choreographies [13]. The details are immaterial for the present development.

Table 1 Summary of types in the original Roqc formalisation [13, 12].

Type	Variable	Description
`Choreography`	`C`	Choreographies
`Pid`	`p`, `q`, `r`, `s`	Process names (identifiers)
`list Pid`	`ps`	List of process names
`Var`	`x`, `y`, `z`	Variable names
`Val`	`v`	Values
`Expr`	`e`	Expressions (evaluate to values)
`BExpr`	`b`	Boolean expressions (evaluate to Booleans)
`Label`	`l`	Labels (`left` and `right`)
`RecVar`	`X`	Procedure names (or recursive variables)
`DefSet`	`D`	Sets of procedure definitions in CC
`State`	`s`	Maps from variables to values
`Configuration`	`c`	Choreographic programs equipped with states
`TransitionLabel`	`t`	Transition labels
`list TransitionLabel`	`tl`	Lists of transition labels
`Program`	`P`	Choreographies/networks with procedure definitions

```
Definition DefSet := RecVar → (list Pid) * Choreography.
```

This can be defined as a (total) function, where unused procedures are mapped to ([p], End) for some process p. (For technical reasons, the set of process names cannot be empty.)

A *choreographic program* is a pair containing a set of procedure definitions and the choreography being executed (also called "main" or "running" choreography).

```
Definition Program := DefSet * Choreography.
```

There are a number of well-formedness conditions about choreographies that reflect their intended meaning, e.g., no process is allowed to communicate with itself. These assumptions are immaterial for our work, and we do not include them here.

▶ **Example 1.** We present a very simple choreography to find the first zero of a function f:nat → nat, which is the formalization in Rocq of one of the running examples from [8]. This choreography uses a language of expressions ITexpr with three constructors: zero and succ, representing natural numbers in unary notation, and val, where val x denotes the value stored at variable x in the process evaluating it. The evaluation function formalizes these intuititions. Here, ITvar (the type for procedure names) and ITnat are aliases for the decidable type of natural numbers, introduced for readability.

```
Fixpoint it_eval (e : ITexpr) (s : ITvar → ITnat) : ITnat :=
  match e with
  | zero ⇒ 0
  | succ e ⇒ it_eval e s + 1
  | val x ⇒ s x
  end.
```

In this signature, Boolean expressions have the form f_is_zero_ e, which evaluates to true iff f evaluates to 0 on the evaluation of e.

Procedures names are also encoded as natural numbers. We define a procedure map as follows, where the interesting procedure is the one called 0 while all the other ones are defined as End in the catch-all case.

```
Definition Defs : DefSet := fun ProcName ⇒
  match ProcName with
  | 0 ⇒ ([p :: q :: nil],
          p.(val x) ⟶ q.x;
          If q.f_is_zero (val x)
            Then (q ⟶ p [left]; End)
            Else (q ⟶ p [right]; p.x ⟵ succ (val x); Call ProcName))
  | _ ⇒ ([p :: nil], End)
  end.
```

Procedure is 0, which uses processes `p` and `q`. Here, `p` sends the current value stored in `x` to `q`, who checks whether `f` evaluates to 0 on that value. If this is the case, `q` notifies `p` that the choreography terminates. Otherwise, `q` notifies `p`, who increases the value of `x` and recurs.

In the main choreography, `p` initiates the value of `x` to 0 and calls procedure 0.

```
Example zero_chor := (p.x ⟵ zero; Call 0 nil).
```

Semantics. The semantics of CC is based on some standard, intuitive assumptions from process calculi: processes run concurrently, independently of each other, and own a local store that associates each variable to its value; communications are synchronous; and the network is reliable (messages are always delivered exactly once and in the right order).

▶ **Example 2.** Concurrent execution of processes allows for choreographies to exhibit concurrent behaviour. Consider the following example from [15].

```
o.order ⟶ p.x; o'.order' ⟶ p'.y; End
```

Here, two processes `o` and `o'` place independent orders to two different providers `p` and `p'`. Since these two communications occur between disjoint pairs of processes with no knowledge of each other's actions, they should be executable in any order. This possibility is known as *out-of-order execution* for choreographies.

The semantics of CC is given in [13] as a labelled transition system on configurations, consisting of a program and a (memory) state. A state is a family of maps from variable names to values, indexed by processes (each map represents the memory of one process).

```
Definition State := Pid → Var → Value
```

We write `s [==] s'` to denote that `s` and `s'` are extensionally equal, and `s⟦p,x ⇒ v⟧` for the state obtained from updating `s` with the mapping `p,x ↦ v`.

Figure 1 includes some representative transition rules for choreographic configurations. `(D,C,s) —[t]⟶ (D,C',s')`, denotes a transition between two configurations whose observable action is described by the transition label `t`.

Rule `CC_Com` mimics a value communication from process `p` to process `q`: expression `e` is first evaluated by `p` to a value `v` (first premise, which uses the auxiliary function `eval`), then the value of variable `x` at `q` is updated accordingly and the communication term is consumed. The transition label `TL_Com p v q` captures the visible communication of `v` from `p` to `q`.

Rule `CC_Sel` is similar, but the state remains unchanged after the transition. Selections play a role in implementations, which is immaterial for this work [10]. The transition label `TL_Sel p q l` registers the communication of label `l` from `p` to `q`.

$$\frac{\texttt{v} := \texttt{eval e s p} \quad \texttt{s'} \ [==]\ \texttt{s}[\![\texttt{q,x} \Rightarrow \texttt{v}]\!]}{(\texttt{D}, \texttt{p.e} \longrightarrow \texttt{q.x; C,s}) \ \text{—}[\texttt{TL\_Com p v q}]\!\!\longrightarrow (\texttt{D,C,s'})} \ \texttt{CC\_Com}$$

$$\frac{\texttt{s} \ [==]\ \texttt{s'}}{(\texttt{D,p} \longrightarrow \texttt{q[l]; C,s}) \ \text{—}[\texttt{TL\_Sel p q l}]\!\!\longrightarrow (\texttt{D,C,s'})} \ \texttt{CC\_Sel}$$

$$\frac{\texttt{beval b s p} = \texttt{true} \quad \texttt{s} \ [==]\ \texttt{s'}}{(\texttt{D}, \texttt{If p.b Then C1 Else C2,s}) \ \text{—}[\texttt{TL\_Tau p}]\!\!\longrightarrow (\texttt{D,C1,s'})} \ \texttt{CC\_Then}$$

$$\frac{\texttt{disjoint\_eta\_rl}\ \eta\ \texttt{t} \quad (\texttt{D,C,s}) \ \text{—}[\texttt{t}]\!\!\longrightarrow (\texttt{D,C',s'})}{(\texttt{D},\eta\texttt{; C,s}) \ \text{—}[\texttt{t}]\!\!\longrightarrow (\texttt{D},\eta\texttt{; C',s'})} \ \texttt{CC\_Delay\_Eta}$$

$$\frac{\texttt{disjoint\_p\_rl p t} \quad (\texttt{D,C1,s}) \ \text{—}[\texttt{t}]\!\!\longrightarrow (\texttt{D,C1',s'}) \quad (\texttt{D,C2,s}) \ \text{—}[\texttt{t}]\!\!\longrightarrow (\texttt{D,C2',s'})}{(\texttt{D, If p.b Then C1 Else C2,s}) \ \text{—}[\texttt{t}]\!\!\longrightarrow (\texttt{D,If p.b Then C1'Else C2',s'})} \ \texttt{CC\_Delay\_Cond}$$

Figure 1 Semantics of choreographic configurations (selected rules).

Rule `CC_Then` addresses the case where process `p` evaluates the guard `b` of a conditional to `true` (using the auxiliary function `beval`), which makes the choreography reduce to its then-branch. The transition label `TL_Tau p` represents an internal action at `p`, as standard in process calculi.

Out-of-order execution of communications is managed by rule `CC_Delay_Eta`, following the intuition given in Example 2. A choreography η; `C` is allowed to perform an action in the continuation `C` as long as that action does not involve any processes in η (premise `disjoint_eta_rl` η `t`). The communication η is left unchanged in the result of the transition. A similar reasoning for conditionals is embodied in rule `CC_Delay_Cond`.

We write —[tl]⟶* for the reflexive and transitive closure of the transition relation, where `tl` is a list of transition labels.

▶ **Example 3.** We show how the semantics captures the intuition from Example 2. If `D` and `s` are such that `order` evaluates to `v` at `o` and `order'` evaluates to `v'` at `o'`, according to `eval`, then both

$$(\texttt{D}, \texttt{o.order} \longrightarrow \texttt{p.x; o'.order'} \longrightarrow \texttt{p'.y; End,s}) \ \text{—}[\texttt{TL\_Com o v p;TL\_Com o' v' p'}]\!\!\longrightarrow^* (\texttt{D,End,s'})$$

and

$$(\texttt{D}, \texttt{o.order} \longrightarrow \texttt{p.x; o'.order'} \longrightarrow \texttt{p'.y; End,s}) \ \text{—}[\texttt{TL\_Com o' v' p';TL\_Com o v s}]\!\!\longrightarrow^* (\texttt{D,End,s'})$$

are possible executions of this choreography, where $\texttt{s'} \ [==]\ \texttt{s}[\![\texttt{s1,x} \Rightarrow \texttt{v}]\!][\![\texttt{s2,x} \Rightarrow \texttt{v'}]\!]$.

Choreographies can be automatically compiled to distributed implementations, which are guaranteed to implement the corresponding protocol [10, 14]. This part of the theory is not relevant to the current work, and we do not describe it.

3.2 Hoare calculi

Hoare logic [1, 21] is a well-established framework for reasoning about programs in an axiomatic way. Hoare logics are based on triples $\{\varphi\}P\{\psi\}$, with intuitive semantics "if executing program P from a state satisfying φ terminates, then the end state satisfies ψ". Formulas φ and ψ are called the *precondition* and *postcondition* of the triple, respectively. These logics are typically two-layered hybrid logics, with independent languages for the state formulas (the φ and ψ in the triples) and for judgements about programs (the triples themselves).

The first proposal of a Hoare logic for a choreographic language [24] is unusual in that both the language of programs and the logic are mutually defined. This happens because the authors' purpose is to restrict the choreographic language to programs with specific well-formedness properties that are included in its syntax.

Our formalization follows [8], which takes a more typical approach – albeit for a recursive, rather than imperative, language. The lower layer of the language defines formulas that describe properties of states, in terms of expressions relating values of variables stored at processes. The syntax of expressions is derived from the choreography's own language of expressions, with a small change to allow for combining values from different processes (see Section 4). Judgements are then built from these formulas in the standard way, and proven using rules that reflect the semantics of choreographies (Section 5). Using these rules, we can for example show that choreography `zero_chor` from Example 1 satisfies the judgement {{`TRUE`}}`zero_chor` {{`PHI`}}, where `PHI` states that the value stored by process `p` in variable `x` is 0.

4 The State Logic

In this and the next sections, we present the Hoare calculus from [8] together with its formalization in Rocq. In this way, we avoid repetition and spare the reader the need for dealing with two notations, as we introduce everything with the notation from the formalization.

Since the underlying choreography calculus leaves the concrete types for e.g. expressions and values unspecified, the Hoare calculus is also parametric on these.

There are a few design options that need to be considered. The language for state formulas in [8] is defined as

$$\varphi, \psi ::= (\mathcal{E} = \mathcal{X}) \mid \delta \mid \varphi \wedge \varphi \mid \neg\varphi$$

where $\mathcal{X}$ is a logical variable, $\mathcal{E}$ is an expression in a language derived from that of choreographic expressions, and δ is a formula in a decidable theory whose terms include logical variables. Intuitively, state formulas can bind logical variables to the values of choreographic expressions, and these variables can in turn be used in formulas within a decidable theory.

Expressions. The authors of [8] state:

> our state logic is parameterised on a set of expressions that is freely generated from the same signature, but using localised variables `p.x`.

The idea is that the type of choreographic expressions is typically defined inductively, with variables as atomic terms, and the same inductive definition should be applied to define the type of logical expressions. Evaluation is implicitly defined for logical expressions in the obvious way. This very intuitive definition is however not amenable to formalization.

Instead, we look at how the two types are related to each other. The Hoare logic defines $L(\mathsf{p}, e)$ (localization of e at p) and $\mathcal{E}[\mathsf{q}.x := \mathsf{p}.e]$ (localized substitution of $\mathsf{q}.x$ by $\mathsf{p}.e$) in $\mathcal{E}$ by straightforward replacement:

- $L(\mathsf{p}, e)$ replaces each variable x in e by $\mathsf{p}.x$;
- $\mathcal{E}[\mathsf{q}.x := \mathsf{p}.e]$ replaces each occurrence of $\mathsf{q}.x$ in $\mathcal{E}$ with $L(\mathsf{p}, e)$.

The key properties of these functions are particular cases of Lemma 1 and Corollary 1 in [8]: for any state Σ,

(i) e evaluates to v at $\Sigma(p)$ iff $L(\mathsf{p}, e)$ evaluates to v at Σ;

(ii) if e evaluates to v at $\Sigma(p)$, then evaluating $\mathcal{E}$ at $\Sigma[\langle \mathsf{q}, x\rangle \mapsto v]$ is equivalent to evaluating $\mathcal{E}[\mathsf{q}.x := \mathsf{p}.e]$ at Σ.

We make a further simplification. Given that there are two types of expressions in choreographies, it would be natural to use two distinct types of variables in the state logic. However, since Boolean expressions always evaluate to a Boolean value, we skip the second type and instead add a formula constructor allowing comparison directly to a Boolean (see below). This simplifies the formalization, eliminating the need for either duplicate results or use of sumtypes in several places. Additionally, it allows us to solve one specific issue with one lemma, as we discuss in Section 6.2.

```
Record HoareSignature := {
  lexpr  : DecType;
  blexpr : DecType;

  lev     : lexpr → (Pid → Var → Value) → Value;
  lev_wd  : (∀ p x, f p x = f' p x) → ∀ e, lev e f = lev e f';
  blev    : blexpr → (Pid → Var → Value) → Bool;
  blev_wd : (∀ p x, f p x = f' p x) → ∀ e, blev e f = blev e f';

  lvar   : DecType; (* Logical variables*)
}.
```

Functions `lev` and `blev` evaluate logical (Boolean) expressions, much as the evaluation functions for choreographic expressions in the original formalization [14]. For conciseness, we omit leading universal quantifiers in types/formulas.

We assume a fixed signature `HSig` (which parameterizes the remainder of the development), and use `LExpr`, `BLexpr`, `eval_lexpr`, `eval_blexpr` and `LVar` for the different datatypes in `HSig`.

Localization and substitution are record types, each consisting again of two functions (one for each type of expressions) required to satisfy the properties stated above. For conciseness we only include the fields related to expressions.

```
Record Localization := {
  localize_expr           : Pid → Expr → LExpr;
  localize_expr_state_eqv : eval_lexpr (localize_expr p e) s = eval_on_state Ev e s p;
  (...)
}.

Record Substitution := {
  subst_lexpr     : LExpr → Pid → Var → LExpr → LExpr;
  subst_lexpr_eqv : eval_on_state Ev e s p = v →
    eval_lexpr E (s⟦q, x ⇒ v⟧) =
      eval_lexpr (subst_lexpr E q x ((loc.(localize_expr) p e))) s;
  (...)
}.
```

We again parameterize the development on fixed instances `loc` and `sub` of these types.

▶ **Example 4.** Continuing with the example of our choreography to find zeros (Example 1), we define datatypes for logical expressions and logical Boolean expressions using constructors `lzero`, `lsucc`, `lval` and `lf_is_zero`. Evaluation mimics evaluation in the choreography language. Localization and substitution are then straightforward to define, and the required lemmas simple to prove.

The decidable theory. The second ingredient is formally defining the notion of a decidable logical theory with formulas ranging over logical variables.

Our intuition is to view formulas as functions from variables to `Prop`. We formalize this as a record containing a type of formulas, a function mapping each formula to a list of its free variables, and a function evaluating a formula given an assignment (function of

type `Assignment := LVar → ChorVal`). Evaluation of a formula can only depend on the values assigned to the variables appearing in the formula. Furthermore, we can always decide whether evaluated formulas are `True`.

```
Record DecTheory := {
  decform            : DecType;
  vars               : decform → list LVar;
  eval_decform       : decform → Assignment → Prop;
  eval_decform_dec   : {eval_decform d r} + {~eval_decform d r};
  eval_decform_wd    : let l := vars d in
    d = d' → map r l = map r' l → eval_decform d r = eval_decform d' r';
}.
```

The decidable theory can not refer to Boolean expressions directly. Although this is a restriction with respect to [8] (where logical variables can range over both types of expressions), it is a natural one: the purpose of the state logic is to talk about the values stored at processes. Additionally, the state logically can include properties of these expressions.

▶ **Example 5.** In most of our examples, the decidable theory contains only one type of formula, stating that a logical variable equals a concrete value.

State formulas. With these ingredients in place, we can define the type of state formulas. The first two constructors correspond to the informal type $\mathcal{E} = \mathcal{X}$, where we distinguish between the two types of formulas/expressions.

```
Inductive StateForm :=
| sf_eq_v    : LExpr → LVar → StateForm
| sf_eq_b    : BLExpr → Bool → StateForm
| sf_decform : DecForm → StateForm
| sf_conj    : StateForm → StateForm → StateForm
| sf_neg     : StateForm → StateForm.
```

We define suggestive notations for state formulas such as `! P` and `P [∧] Q`, as well as for disjunctions, implications and equivalences (defined as abbreviations). Equality between an expression and a logical variable is denoted `E [=E] X`, while equality between a Boolean expression and a Boolean is written `E [=B] B`. We additionally define the abbreviations `E1 [= X =] E2` for `((E1 [=E] X) [∧] (E2 [=E] X))` and `E1 [< X >] E2` for `((E1 [=E] X) [↔] (E2 [=E] X))`. All these notations are in line with the definitions from [8].

The semantics of state formulas also follows the presentation in [8] directly. Satisfaction of a formula depends on a `Store` (the choreographic state, renamed to avoid ambiguity, to evaluate expressions) and on an `Assignment` (for the logical variables). We define the suggestive notation $[\![\mathtt{s},\ \rho\ \vdash \mathtt{P}]\!]$ for (`StateForm_satisfaction P s` ρ).

```
Fixpoint StateForm_Satisfaction (SF : StateForm) (s: Store) (ρ : Assignment) : Prop :=
  match SF with
  | sf_eq_v E X   ⇒ (eval_lexpr E s) = ρX
  | sf_eq_b E B   ⇒ (eval_blexpr E s) = B
  | sf_decform d ⇒ Theory.(eval_decform) d ρ
  | sf_conj P Q   ⇒ [[s, ρ ⊢ P]] ∧ [[s, ρ ⊢ Q]]
  | sf_neg P      ⇒ ~[[s, ρ ⊢ P]]
  end.
```

This relation is decidable and well-defined.

$$\frac{\{\{P\}\}\,C\,\{\{Q\}\}}{\{\{P\{\![q,x \leftarrow p,e]\!\}\}\}(p.e \longrightarrow q.x;\, C)\,\{\{Q\}\}}\ \texttt{H\_Com}$$

$$\frac{\begin{array}{c}\{\{P\ [\wedge]\ (\texttt{loc.(localize\_bexpr)}\ p\ b\ [=B]\ \texttt{true})\}\}C1\ \{\{Q\}\}\\ \{\{P\ [\wedge]\ (\texttt{loc.(localize\_bexpr)}\ p\ b\ [=B]\ \texttt{false})\}\}C2\ \{\{Q\}\}\end{array}}{\{\{P\}\}\,\texttt{If}\ p.b\ \texttt{Then}\ C1\ \texttt{Else}\ C2\ \{\{Q\}\}}\ \texttt{H\_Cond}$$

$$\frac{D\ X = (P,Q)}{\{\{P\}\}\texttt{Call}\ X\ ps\ \{\{Q\}\}}\ \texttt{H\_Call} \qquad \frac{}{\{\{P\}\}\texttt{End}\ \{\{P\}\}}\ \texttt{H\_Nil}$$

$$\frac{\texttt{valid}\ (P\ [\rightarrow]\ P') \qquad \{\{P'\}\}\ C\ \{\{Q'\}\} \qquad \texttt{valid}\ (Q'\ [\rightarrow]\ Q)}{\{\{P\}\}\,C\,\{\{Q\}\}}\ \texttt{H\_Weak}$$

Figure 2 Hoare calculus for choreographies (selected rules).

```
Lemma sfs_dec : {⟦s, ρ ⊢ P⟧} + {~⟦s, ρ ⊢ P⟧}.
Lemma sfs_wd : (∀ X, ρ X = ρ' X) → ⟦s, ρ ⊢ P⟧ ↔ ⟦s, ρ' ⊢ P⟧.
Lemma sfs_eqv_states : ⟦s, ρ ⊢ P⟧ → s [==] s' → ⟦s', ρ ⊢ P⟧.
```

Validity of formulas is defined as expected by a predicate `valid:StateForm → Prop`.

Substitution extends by a straightforward recursive definition to a function `sf_subst` on `StateForm`. We use the suggestive notation `P{⟦q,x ← p,e⟧}` for (`sf_subst P q x p e`). For this extension, we can prove the full versions of Lemma 1 and Corollary 1 from [8].

```
Lemma localize_expr_preserves_interpretation : ρX = v →
  eval_on_state Ev e s p = v ↔ ⟦s, ρ ⊢ ((loc.(localize_expr) p e) [=E] X)⟧.

Lemma sfs_subst : eval_on_state Ev e s p = v →
  ⟦s⟦q, x ⇒ v⟧, ρ ⊢ P⟧ ↔ ⟦s, ρ ⊢ (P{⟦q,x ← p,e⟧})⟧.
```

There is an additional Boolean counterpart to the first lemma, which we omit.

5 The Hoare Calculus

The next step is to formalize the rules for the Hoare calculus from [8]. To deal with procedures, some of the rules require the presence of a *procedure specification map*, formalized as

```
Definition ProcSpecMap := RecVar → (StateForm * StateForm).
```

This map specifies a precondition and postcondition for each procedure name, with the informal expectation that the corresponding judgment holds. We assume a specific instance `D` of this type.

The rules are defined as an inductive type `H_Rules`, with the accompanying notation `{{P}}C {{Q}}` for (`H_Rules P C Q`). For readibility, we present the constructors of this type in a standard mathematical notation (Figure 2).

These rules model the semantics of choreographies on states. We discuss the most interesting rules briefly. Rule `H_Com` follows the intuition behind the standard rule for assignment in other Hoare calculi by backpropagating the result of the assignment. Rule `H_Call` directly reads the effect of a procedure call from the procedure specification map. Rule `H_Weak` is the standard weakening rule.

Rule `H_Cond` differs from the one presented in [8]. Given the possibility of comparing directly to a Boolean value, the premises for this rule can be written without resorting to freshly generated Boolean variables. Initially, this rule was formalized exactly as in the original presentation – which turned out to be problematic in the proof of one lemma (see the discussion in Section 6.2). The current formulation also has the advantage of being simpler.

▶ **Example 6.** We consider again the choreography for finding zeros from Example 1. We can specify its functional correctness with the Hoare triple {{`TRUE`}}`zero_chor` {{`PHI`}}, where `TRUE` is a tautological formula (we use (`lval p x [=E] V`) [→] (`lval p x [=E] V`), which is an instance of `A [→] A`) and `PHI` is the formula (`lf_is_zero (lval p x) [=B] true`).

The procedure specification map for this example looks like

```
Definition Psm : ProcSpecMap := fun ProcName ⇒
  match ProcName with
  | 0 ⇒ (TRUE, PHI)
  | _ ⇒ (TRUE, TRUE)
  end.
```

that is, procedure 0 ensures `PHI` from any state, while the other procedures are not relevant.

Validity of a Hoare triple is defined following the usual intuition.

```
Definition h_valid_triple P C Q s s' r Ps ts :=
  ⟦s, r ⊢ P⟧ → (Ps, C, s) —[ts]⟶* (Ps, End, s') → ⟦s', r ⊢ Q⟧.
```

The final ingredient is the notion of *weakest liberal precondition* – intuitively, the weakest requirement on a precondition that, after executing the given choreography, ensures the postcondition.

```
Fixpoint wlp (psm: ProcSpecMap) (C:Choreography) (P:StateForm) : StateForm :=
  match C with
  | p.x ⟵ e;; C' ⇒ (wlp psm C' P){[p,x ← p,e]}
  | p.e ⟶ q.x;; C' ⇒ (wlp psm C' P){[q,x ← p,e]}
  | p ⟶ q [l];; C' ⇒ wlp psm C' P
  | Cond p b C1 C2 ⇒ let e := loc.(localize_bexpr) p b in
        (e [=B] true) [→] wlp psm C1 P [∧] (e [=B] false) [→] wlp psm C2 P
  | Call Y _       ⇒ fst (psm Y)
  | RT_Call Y _ C' ⇒ wlp psm C' P
  | End            ⇒ P
  end.
```

This definition differs slightly from the one in [8], again, in the case for the conditional – where we directly compare localized Boolean expressions to Booleans.

Finally, we say that a procedure specification map is *consistent* if all its judgements can be derived using the rules of the calculus; and *adequate* if each precondition implies the weakest liberal precondition of the postcondition. The formalization follows the spirit of the original formalization of the choreographic language [14]: it parameterizes these notions on a list of procedure names, and the constraints only apply to these. The motivation for this is that, in practice, choreographies only use a finite number of procedures,[2] and it is irrelevant how unused procedures behave. This will be apparent in some of the later results.

[2] Although one can define mathematically interesting choreographies without this property...

```
Definition psm_consistent (Ps:DefSet) (psm: ProcSpecMap) (Xs: list RecVar) :=
  ∀ X, In X Xs → {{fst (psm X)}} snd (Ps X) {{snd (psm X)}}.

Definition psm_adequate (defs: DefSet) (Q: StateForm) psm (Xs: list RecVar) : Prop :=
  ∀ X, In X Xs → valid ((fst (psm X)) [↔] wlp psm (snd (defs X)) Q) ∧ snd (psm X) = Q.
```

6 Results and Issues

We now discuss the results from [8] and the challenges observed while formalizing them.

6.1 Soundness

Soundness of the Hoare calculus states that, if {{P}}C {{Q}} is valid and execution of C from a state satisfying P terminates, then Q holds in the final state. This is proved in two steps. First, the result is shown to hold if the choreography is only allowed to perform *head reductions* (i.e. the rules for out-of-order execution are not used) – this is Lemma 2 in [8]. Then confluence is invoked to establish the general result (Theorem 1). The result requires consistency of the procedure specification map to cover the case of procedure calls.

While confluence was a part of the original development, head reductions had not been formalized previously – but they posed no challenges.

```
Theorem soundness : used_procedures (Ps, C) Xs → Program_WF (Ps, C) →
  psm_consistent Ps D Xs → {{P}} C {{Q}} → h_valid_triple P C Q s s' r Ps ts.
```

Predicate used_procedures checks that Xs contains all procedures that may be invoked during the execution of C, while Program_WF collects well-formedness requirements for the choregraphy. The proof of this result follows the inductive argument from [8] directly.

6.2 Completeness

The argument for completeness in [8] proceeds in several steps. First, it is proved that {{wlp D C Q}}C {{Q}} is derivable for every choreography C as long as the procedure specification map is adequate (Lemma 3). This implies that adequate procedure specification maps are consistent (Corollary 2) and that execution of C from a state satisfying wlp D C P, if it terminates, leads to a state satisfying P.

The next step is showing the converse: if execution of C terminates in a state satisfying P, then the initial state satisfies wlp D C P. This is first proved for head reductions (Lemma 4) and then generalized to any sequence of transitions using confluence (Corollary 4). These results are then combined to establish weak completeness (Theorem 2): if it is the case that executing C from a state satisfying P leads to a state satisfying Q whenever the execution is terminating, then the judgement {{P}}C {{Q}} is derivable.

The formalization shows two problems with this chain of arguments: a subtle detail in the proof of Lemma 3, which is solved by our change in treatment of the conditional (both in the calculus and in the computation of weakest liberal preconditions); and a reasoning flaw in the proof of Theorem 2 – which does not hold.

Proof of Lemma 3. In the original formulation [8], both rule H_cond and the case for conditionals in the computation of wlp required the introduction of a fresh variable to fix the value of the evaluation of the guard. The proof of the lemma proceeds inductively. In the case of a conditional, C is If p.b Then C1 Else C2. First, rule H_Cond is applied, generating

two similar subcases, which are then solved by weakening and induction hypothesis using the fact that `valid ((wlp D C P) [∧] (b' [=B] true)) [→] (wlp D C1 P)`, where we write `b'` for the localization of `b` at `p`.

With the original formulation, applying rule `H_Cond` required introducing a logical variable `X` that is fresh for `wlp D C P` and `P`. Unfolding the definitions, we now would need to show the validity of

```
(((b' [=B] Y [∧] Y [=B] true) [→] (wlp D C1 P)) [∧] (...)
    [∧] (b' [=B] X [∧] X [=B] true)) [→] (wlp D C1 P)
```

where `Y` is the fresh variable introduced by `wlp` and the omitted part is the corresponding subformula for `C2`. Now this implication does not hold in the states where `X` has value `false` – or indeed for any fresh variable. Using `Y` in the application of `H_cond` does not work either, since it does not fulfill the freshness requirement of the rule.

Instead, our definition of `H_Cond` allows the proof of Lemma 3 to go through.

```
Lemma wlp_correctness : used_procedures (Ps, C) Xs →
  psm_adequate Ps Q D Xs → {{wlp D C Q}} C {{Q}}.
```

(Note again the precondition `used_procedures (Ps, C) Xs`, which states that `Xs` contains all the procedures potentially called under execution of `C` – this ensures that the adequacy requirement is strong enough.) ◀

Intermediate results. The proof of the next results directly follows the reference article.

```
(* Corollary 2 *)
Lemma adequacy_implies_consistency :
  (∀ X, In X Xs → used_procedures (Ps, (snd (Ps X))) Xs) →
  psm_adequate Ps Q D Xs → psm_consistent Ps D Xs.

(* Corollary 3 *)
Lemma wlp_soundness : used_procedures (Ps, C) Xs → Program_WF (Ps, C) →
  psm_adequate Ps Q D Xs → h_valid_triple (wlp D C Q) C Q s s' r Ps ts.

(* Corollary 4 *)
Lemma wlp_completeness : used_procedures (Ps, C) Xs → Program_WF (Ps, C) →
  psm_adequate Ps Q D Xs →
  ⟦s', r ⊢ Q⟧ → (Ps, C, s) —[ts]⟶ * (Ps, End, s') → ⟦s, r ⊢ (wlp D C Q)⟧.
```

(We omit Lemma 4, which is nearly identical to the last result.)

Proof of Theorem 2. The next challenge is Theorem 2 (completeness), where the result does not hold. The statement claims that: if all terminating executions of `C` from states satisfying `P` end in states that satisfy `Q`, then {{P}}C {{Q}} is derivable. The argument is as follows: if executing `C` terminates in a state satisfying `Q`, then the start state satisfies `wlp D C Q` (Corollary 4). Since this must be the case for all states satisfying `P`, it follows that `valid (P [→] wlp D C Q)`, and Lemma 3 can be applied to build a derivation.

The issue is that this argument only holds if execution of `C` always terminates – otherwise there may be states satisfying `P` but not `wlp D C Q`. Indeed, it is quite easy to make a counterexample: define a single procedure `X` using one process `p`, where the only action in `X` is recursively calling itself. Then the choreography `X` never terminates, which means that any judgement {{P}}X {{Q}} is trivially true. However, only the instances that follow from rules `H_Call` and `H_Weak` are provable. We have formalized a concrete instance of this counterexample, showing that we can write a concrete procedure specification map that is adequate and a Hoare triple that is valid but not derivable. ◀

6.3 Decidability

The reference for this work also includes three results on decidability, claiming that (i) it is decidable whether {{P}}C {{Q}} is derivable, (ii) consistency of a procedure specification map is decidable, and (iii) adequacy of a procedure specification map is decidable. The proofs of these results all rely on Theorem 2, and are therefore compromised. There is a further issue, as the argumentation at one point mistakenly assumes decidability of the state logic (instead of only the decidable theory).

Indeed, it is straightforward to show that the Hoare logic is not decidable: the triple {{TRUE}}C {{FALSE}} (where we write TRUE and FALSE for some suitable tautology in the state logic and its negation) holds iff C does not terminate for any of its inputs, which is undecidable.

In order to analyse the validity of the arguments in the original exposition, we explored results on *relative decidability*, and showed that property (i) above suffices to derive the two other results – as well as decidability of the state logic.

```
Hypothesis hoare_dec : ∀ C P Q, {({{P}} C {{Q}})} + {~{{P}} C {{Q}}}.

Lemma valid_dec : ∀ P, {valid P} + {~valid P}.

(* Corollary 5 *)
Lemma psm_consistent_dec : {psm_consistent Ps psm Xs} + {~psm_consistent Ps psm Xs}.

(* Lemma 6 *)
Lemma psm_adequate_dec : {psm_adequate Ps Q psm Xs} + {~psm_adequate Ps Q psm Xs}.
```

One might argue that this is a purely academic exercise, given that we already established the falsehood of hypothesis hoare_dec. Nevertheless, there is some value in studying notions of relative decidability. Furthermore, these proofs rely on very general properties of the Hoare calculus – meaning that these implications should also hold for simpler choreography languages, where decidability may be achievable.

6.4 Examples and automation

To test the validity of our development we formalized some example choreographies. We were interested in assessing the usability of our definitions, as well as to see how much the construction of Hoare derivations could be automated.

For the latter part, we developed a very simple tactic that first applies weakening, and then recursively goes through the choreography and applies the rule consuming its first action. All these application steps leave the intermediate formulas instantiated. When the last action of the choreography is reached (End or a procedure call), the corresponding rule unifies the precondition with either the postcondition or the formula in the procedure specification map, which instantiates all existential variables and leaves a single goal for the user – proving that the original precondition implies the formula generated from the postcondition and the choreography.

This process corresponds to computing the weakest liberal precondition. For the main choreography, it could be simplified to invoking Lemma 3. However, this result requires the procedure specification map to be adequate. It may be useful to use a procedure specification map that is consistent, but not adequate (where some preconditions may be stronger than weakest liberal preconditions); in this situation, Lemma 3 can not be applied. This tactic can also be used to prove consistency of the procedure specification map directly.

Our first two examples use a simple choreography language where values are natural numbers, and expressions are either looking up the value of a variable (`val x`) or adding a constant to an expresion (`add e n`). The only Boolean expression can check whether an expression evaluates to an even number (`even e`). The decidable theory only contains formulas of the form `eqv n X`, which hold for assignments mapping `X` to `n`.

```
Example comm_chor (p q:Pid) (x y:Var) :=
  (p.(val x) ⟶ q.y; q.y ⟵ (add (val y) 1); q.(val y) ⟶ p.x; End).
```

In this example, `p` sends its value of `x` to `q`, who receives it on `y`, adds 1 to it, and returns the result to `p`, who updates its value in `x`. If $\texttt{p} \neq \texttt{q}$, then we can derive the judgement

```
{{(ladd (lval p x) 1 [=E] I) [∧] sf_decform Sig HSig Theory (eqv i I)}}
  comm_chor p q x y
{{(lval p x [=E] I) [∧] sf_decform Sig HSig Theory (eqv i I)}}
```

both directly and by applying our tactic. Not surprising, the latter proof is much shorter.

Our next example is a choreography containing a conditional

```
Example if_chor (p:Pid) (x:Var) :=
  If p.(even (val x)) Then End Else (p.x ⟵ add (val x) 1; End).
```

and we show that we can derive the judgement

```
{{lval p x [< X >] lval p x}} if_chor p x {{leven (lval p x) [=B] true}}
```

where the precondition is simply a tautology. Here, the automated tactic produces one trivial goal (for the `Then` case) and one goal that requires some arithmetic reasoning.

Our next example is the Diffie-Hellmann key exchange protocol [18], also included in [8]. This choreography again works with natural numbers as values, and there are two constructors for expressions: `val x` again reads the value stored in the variable `x`, while `powmod e1 e2` computes $(\texttt{e1}^{\texttt{e2}} \mod \texttt{p})$, where `p` is a large prime number. There are no Boolean expressions.

Using this language, we can write the choreography

```
Example dh_chor (Alice Bob:Pid) (rA rB g xA xB xBA xAB s: Var) :=
    Alice.xA ⟵ (val rA); Bob.xB ⟵ (val rB);
    Alice.(powmod (val g) (val xA)) ⟶ Bob.xAB;
    Bob.(powmod (val g) (val xB)) ⟶ Alice.xBA;
    Alice.s ⟵ (powmod (val xBA) (val xA));
    Bob.s ⟵ (powmod (val xAB) (val xB));
    End.
```

and show that the final values stored by both participants in their respective local variables `s` coincide as long as their initial values in `g` are the same.

```
{{lval Bob g [= G =] lval Alice g}}
  dh_chor Alice Bob rA rB g xA xB xBA xAB sec
{{lval Bob sec [< SEC >] lval Alice sec}}
```

Here again, our simple tactic immediately reduces the proof to an arithmetic argument.

Finally, we finish the analysis of Example 1 – the choreography for finding zeros. Since this example uses procedures, we also show that (i) only procedure 0 can ever be invoked and (ii) the procedure specification map is adequate for this procedure.

```
Lemma zero_chor_procs : used_procedures (Defs, zero_chor) (cons 0 nil).
Lemma zero_chor_adequate : psm_adequate Defs PHI Psm (cons 0 nil).
```

The latter proof requires essentially rewriting formulas in the state logic to formulas in Rocq's own logic, yielding simple goals that can be solved by the `intuition` tactic.

Finally, we show that the judgement stating functional correctness of the choreography is derivable.

```
Lemma zero_chor_correctness : {{TRUE}} zero_chor {{PHI}}.
```

Here we again observe that the proof is very short, and amounts to showing that the generated weakest liberal precondition always holds.

7 Discussion and conclusions

Our original goal with this work was to implement a toolbox for reasoning about choreographic programs, capitalizing on the development of a Hoare calculus for the language of Core Choreographies from [8] and the preexisting Rocq formalization of the latter language [14]. Working within a formalism that was sound, complete and decidable provided potential for developing strong automation tactics that could support reasoning tasks.

The discovery of the issues with Theorem 2 (and its consequences for the subsequent decidability results) created some challenges for this effort. Nevertheless, the tools developed in this endeavour support much of our original goal. The main challenge – even without these issues – is finding the right procedure specification map, much as the main challenge for developing proofs in standard Hoare calculi for imperative languages is finding the right loop invariants. Once such a map has been found, the tactics that we develop can be used to develop most of the necessary proofs by structurally going through the choreography – leaving the user with the task of proving logical implications between the handwritten preconditions and the ones implicitly computed by the prover.

Our examples also provide empirical evidence for the claim that the level of indirection created by duplicating the language of expressions in the state logic does not create significant additional work. In all cases, we followed the informal intuition given in [8] – reuse the inductive definition for choreography expressions with the necessary adjustments – and found out that the consistency lemmas left to prove can be solved nearly automatically. We are therefore content with the extent to which we achieved our goal.

On a different note, our results corroborate previous observations regarding the need for more formal verification in the area of process calculi and choreographic languages [3, 4, 5, 11, 15, 19, 27, 32, 35]. The issue with the choice of fresh variables in the original proof of Lemma 3 [8] was subtle, and the proof system was a key ingredient in understanding exactly why it was not possible to construct a derivation as described. The mistake in the proof of Theorem 2 illustrates a different risk with pen-and-paper proofs – that, when adapting proof strategies to a new context, one may overlook new cases that were not present before. Here again it is valuable to have a proof assistant at hand, as it immediately flags such new cases as not dealt with.

Our formalization is available from Zenodo [16]. It includes our development, as well as the required files from the previous formalization of Core Choreographies [15] – which were updated to Rocq 9.0 with minor refactoring.

References

1 Krzysztof R. Apt and Ernst-Rüdiger Olderog. Fifty years of hoare's logic. *Formal Aspects Comput.*, 31(6):751–807, 2019. doi:10.1007/S00165-019-00501-3.

2 Sylvain Boulmé. Tutorial on Hoare logic. URL: https://rocq-prover.org/p/coq-hoare-tut/8.11.1.

3 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: `https://lmcs.episciences.org/7238`.

4 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. `doi:10.1016/j.ic.2017.07.010`.

5 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. `doi:10.1016/j.tcs.2018.02.010`.

6 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. `doi:10.1145/2429069.2429101`.

7 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In Stephen N. Freund and Eran Yahav, editors, *Procs. PLDI*, pages 237–251. ACM, 2021. `doi:10.1145/3453483.3454041`.

8 Luís Cruz-Filipe, Eva Graversen, Fabrizio Montesi, and Marco Peressotti. Reasoning about choreographic programs. In Sung-Shik Jongmans and Antónia Lopes, editors, *COORDINATION 2023*, volume 13908 of *Lecture Notes in Computer Science*, pages 144–162. Springer, June 2023. `doi:10.1007/978-3-031-35361-1_8`.

9 Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In Ahmed Bouajjani and Alexandra Silva, editors, *Procs. FORTE*, volume 10321 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017. `doi:10.1007/978-3-319-60225-7_7`.

10 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. `doi:10.1016/j.tcs.2019.07.005`.

11 Luís Cruz-Filipe and Fabrizio Montesi. Now it compiles! certified automatic repair of uncompilable protocols. In Adam Naumowicz and René Thiemann, editors, *ITP 2023*, volume 268 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, July 2023. `doi:10.4230/LIPIcs.ITP.2023.11`.

12 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Procs. ICTAC*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. `doi:10.1007/978-3-030-85315-0_8`.

13 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a Turing-complete choreographic language in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *Procs. ITP*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.15`.

14 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *Journal of Automated Reasoning*, 67(2):21:1–21:34, May 2023. `doi:10.1007/s10817-023-09665-3`.

15 Luís Cruz-Filipe, Fabrizio Montesi, and Robert R. Rasmussen. Keep me out of the loop: a more flexible choreographic projection. In Ruzica Piskac and Andrei Voronkov, editors, *LPAR 2023*, volume 94 of *EPiC Series in Computing*, pages 144–163. EasyChair, June 2023. `doi:10.29007/WBW3`.

16 Luís Cruz-Filipe and Thomas Wulff Heissel. Formalizing a Hoare calculus for choreographic programming, May 2026. `doi:10.5281/zenodo.20290519`.

17 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. `doi:10.23638/LMCS-13(2:1)2017`.

18 Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976. `doi:10.1109/TIT.1976.1055638`.

19 Alain Finkel and Étienne Lozes. Synchronizability of communicating finite state machines is not decidable. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *Procs. ICALP*, volume 80 of *LIPIcs*, pages 122:1–122:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ICALP.2017.122`.

20 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.

21 C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

22 The Iris project. URL: https://iris-project.org/.

23 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

24 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies – computing preconditions in choreographic programming. In Ilya Sergey, editor, *Procs. ESOP*, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.

25 Donnacha Oisín Kidney, Zhixuan Yang, and Nicolas Wu. Algebraic effects meet hoare logic in cubical agda. *Proc. ACM Program. Lang.*, 8(POPL):1663–1695, 2024. doi:10.1145/3632898.

26 Aleksandar Kupusinac and Dusan Malbask. Formalization of the general hoare logic laws. *TEM Journal*, 1(3):145–150, 2012.

27 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Procs. FOSSACS*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.

28 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *Procs. ASPLOS*, pages 517–530. ACM, 2016. doi:10.1145/2872362.2872374.

29 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.

30 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Procs. SAC*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

31 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Elvira Albert and Ivan Lanese, editors, *Procs. FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016. doi:10.1007/978-3-319-39570-8_13.

32 Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *Procs. ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015. doi:10.1007/978-3-319-22102-1_19.

33 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. URL: http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

34 Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In June Andronick and Leonardo de Moura, editors, *Procs. ITP*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.27.

35 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

36 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. Multiparty asynchronous session types: A mechanised proof of subject reduction. In Jonathan Aldrich and Alexandra Silva, editors, *Procs. ECOOP*, volume 333 of *LIPIcs*, pages 31:1–31:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ECOOP.2025.31.