イベントドリブンな分散プログラム向けのコレオグラフィックプログラミング言語の検討

加藤 大翔　山崎 徹郎　千葉 滋

本論文では JavaScript の promise 相当の機構を備えた，並行コレオグラフィックプログラミング言語 Renga を提案する．Renga は，プロセスのイベントの発生によって解決される promise を導入する．プログラマはこの promise を用いて，イベントドリブンな分散プログラムをコレオグラフィとして記述できる．Renga では，promise の解決時に起動される継続コレオグラフィを一つずつ逐次に実行するため，並行に発生した非同期イベントが引き起こす並行コレオグラフィ間のインターリーブに起因するレースコンディションが発生しない．また，それを避けるために，ループごとに通信を行う busy-loop やコレオグラフィ外部の同期機構を用いた低レイヤーの同期処理を記述する必要もない．

1　はじめに

コレオグラフィック・プログラミング [3] は，分散システムにおける通信を統一的に記述するためのプログラミングパラダイムである．プログラマは，システムを構成する各プロセスが個別に実行する局所的なプログラムではなく，システム全体の通信を記述した大域的なプログラムであるコレオグラフィを一つ書けば良い．コレオグラフィはコンパイラによって各プロセスのプログラムに機械的に変換されるため，プログラマは局所的なプログラムを大域的な通信に対応づけながら実装する難しさから解放される．また，大域的にプログラムを記述することで，通信における送信と受信を分けて記述する必要がなくなり，結果としてデッドロックフリーが保証される．

一方で，現実の分散システムで広く用いられているイベント・ドリブンなプログラムを，既存のコレオグラフィック・プログラミングで記述することは難しい．これは，イベントが並行かつ非同期的に発生し得るためである．素朴な解決策は，イベントディスパッチループの中で，順に全てのプロセスを訪問し，イベントの発生の有無を確認するポーリングを実装することである．しかしこの方法では，イベントが一つも発生していないループにおいても通信が発生するため，通信量が膨大となるという課題があった．

本研究では，JavaScript 風の promise をプリミティブオブジェクトとして備えた，コレオグラフィック・プログラミング言語である Renga を提案する．Renga の promise は，then と resolve の二つのメソッドを持つ．プログラマは，あるプロセスで発生するイベントに対応する promise を作成し，then メソッドでイベント発生時に呼び出したいハンドラ関数を登録することができる．その後イベントが発生すると，各プロセスのイベント機構が対応する promise の resolve メソッドを内部的に呼び出す．これにより登録されたハンドラ関数が全プロセスで同時に実行される．その結果，ポーリングを書くことなく，イベントが発生した時のみ通信が生じるコレオグラフィを記述できる．

Renga では promise が複数ある場合，各 promise の resolve メソッドの呼び出しは，全てのプロセスにおいて同じ順序で逐次化される．逐次化を行わなければ，複数プロセスでほぼ同時にイベントが発生した場合に，プロセスごとにハンドラ関数の実行順序が入れ

A Study on Choreographic Programming Languages for Event-Driven Distributed Programs.

Hiroto Kato, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院 情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

替わり，競合状態や，通信の対応が崩れることによるデッドロックが生じ得るからである．Renga における逐次化は，逐次化プロセスと呼ぶ特別なプロセスが持つ単一の大域キューを用いて達成する．また，対応する複数のハンドラ関数は，resolve メソッドが呼び出された順序と同じ順序で，一つずつ実行される．この結果，同時に複数のイベントが発生したとしても，コレオグラフィ上で競合状態やデッドロックが発生しないことが引き続き保証される．プログラマがこれを避けるために，コレオグラフィ外にある低レイヤーの同期機構を用いる必要もない．

2 背景

本章では，我々が開発しているコレオグラフィック・プログラミング言語 Renga を用いて，コレオグラフィック・プログラミングの基本概念を説明した上で，既存のコレオグラフィック・プログラミングではイベント・ドリブンなプログラムを実用的に記述することが難しいという限界を明らかにする．ただし，本章で扱う Renga は，第 3 章で導入する promise の拡張を含まない範囲に限られる．

2.1 コレオグラフィック・プログラミング

コレオグラフィック・プログラミングにおいては，プログラマはシステム間の通信を大域的に記述するプログラムであるコレオグラフィを記述するだけでよい．コレオグラフィはコンパイラにより各プロセスのプログラムへ機械的に変換される．この変換過程を endpoint projection（EPP）と呼び，このコレオグラフィ C から得られるプロセス p のプログラムを，C の p への射影と呼ぶ．各プロセスがそれぞれ射影後のプログラムを実行することで，あたかも全てのプロセスが同じタイミングでコレオグラフィ内の同一プログラム片を実行しているように動作する．

図 1 の例を用いて，コレオグラフィック・プログラミングの主要な概念とそれらを Renga を用いて記述する方法について説明する．図 1 は，alice が入力した日付を bob が受け取り，bob の持つ空き日程に含まれるかどうかを知らせる処理の例であり，左にコレオグラフィ，中央および右にそのコレオグラフィの alice への射影と bob への射影を示している．三つのプログラムは対応する処理が同一行番号に配置されるように並べられている．

コレオグラフィに現れる全ての値や一部の組み込み関数は，それがどのプロセスに属するかという情報を伴う．これをロケーションと呼び，Renga では`@`に続けてプロセス名を書くことで注釈する．例えば左 1 行目の `slots@bob` はプロセス bob に属する変数であり，ロケーションが bob であることが注釈されている．射影により，`slots@bob` は `slots` として右の bob の射影のみに現れ，中央の alice への射影の 1 行目は空になることが確認できる．また，左 3 行目の `input<@alice>()` はロケーションが alice の組み込み関数である．

射影後のプログラムに書かれた変数や処理は，全てそのプロセスのみに関連するものであるため，ロケーションは射影によって失われる情報である．実際に，射影後の bob のプログラムでは`@bob` の注釈が除去されていることが確認できる．

Renga は，静的型検査に似たロケーションについての静的検査と，限定的なロケーション推論を備えている．そのため，左 5 行目の `slots` などで見られるように，変数を記述する度にロケーションの注釈を与える必要がない．また，左 3 行目の `date` 変数の宣言の例では，組み込み関数 `input<@alice>()` によりロケーションを推論できるため，ロケーションの注釈を省略できる．

プロセス間の通信はプリミティブ`<-`を用いて記述する．これは Go のチャネルに影響を受けた記法である．左 4 行目 `var d@bob <- date` はロケーション alice の `date` を bob に送信し，bob の新しい変数 `d` に値を束縛することを意味する．この文は alice へは中央 4 行目の `send(bob, date)`，bob へは右 4 行目の `var d = recv(alice)` のように射影される．送信者が存在しない通信の受信待ちは，分散システムにおいて通信によるデッドロックの原因となる．送信と受信が一体となったプリミティブを用いる限り，そのような受信待ちは起きない．

システム全体で同期的な振る舞いを実現するために，コレオグラフィック・プログラミングの条件分岐

```
1 var slots@bob = {"Sep3", "Oct20"}
2
3 var date = input<@alice>()
4 var d@bob <- date
5 if (d in slots) {
6     print("ok"@alice)
7 } else {
8     print("busy"@alice)
9 }
```

```
1
2
3 var date = input()
4 send(bob, date)
5 if (recv(bob)) {
6     print("ok")
7 } else {
8     print("busy")
9 }
```

```
1 var slots = {"Sep3", "Oct20"}
2
3
4 var d = recv(alice)
5 if (d in slots) {
6     send(alice, true)
7 } else {
8     send(alice, false)
9 }
```

図 1　左：コレオグラフィ 中央：alice への射影 右：bob への射影

では，条件の評価結果により振る舞いが変わる全てのプロセスは，必ず同じ方向に分岐しなければならない．そのためには，全ての関連プロセスが同じ方向に分岐するために十分な知識を持つ必要がある．この要件を knowledge of choice（KoC）と呼ぶ．この要件を満たすために，ある単一プロセスが条件分岐の条件式を評価したときは，その他の関連プロセスに評価結果が暗黙的に通知される．図 1 の左 5 行目の条件式 `d in slots` は，`d` と `slots` がロケーション bob の変数であることから，bob 上で評価される．その条件式の評価結果によって，図 1 の左 6，8 行目で見られるように alice が `print("ok"@alice)` を実行するか，`print("busy"@alice)` を実行するかが変化する．よって KoC を満たすために，図 1 の右 6，8 行目で bob が `send(alice, true)` や `send(alice, false)` によって評価結果を alice に通知し，図 1 の中央 5 行目で alice が `recv(bob)` によって評価結果を受け取る．この通知は`<-`を用いたプロセス間通信とは異なり，左のコレオグラフィには直接現れないことに注意されたい．

Renga では，反復は `loop`，関数宣言は `func` というキーワードを用いて記述する．関数は第 3 章で提案する promise の then メソッドの例外を除き，トップレベルで宣言され，Pirouette [1] と同様に値を関数引数として取る．Montesi による Recursive Choreographies [4] とは異なり，Renga の関数はプロセスを引数に取らない．

図 2 は，図 1 のコレオグラフィを関数とループを用いて書き直した例である．図 2 の 1 行目の関数 `schedule` は，ロケーションが bob の空き日程 `slots` を引数に取り，ロケーションが alice の値を返す．2 行目から 9 行目の `loop` では，alice が候補日を入力して bob に送り（3，4 行目），その日程が bob の空き日程に含まれていればその候補日を返して関数を抜ける（5，6 行目）．含まれていなければ，alice で `try another` と表示し（8 行目），再度 3 行目から繰り返す．

```
1  func schedule(slots @bob) @alice {
2      loop {
3          var date = input<@alice>()
4          var d @bob <- date
5          if d in slots {
6              return date
7          }
8          print("try another"@alice)
9      }
10 }
11
12 var slots@bob = {"Sep3", "Oct20"}
13 var date = schedule(slots)
14 print("scheduled: "@alice + date)
```

図 2　関数とループを用いて記述した例

2.2　イベント・ドリブンなコレオグラフィック・プログラミングの難しさ

イベント・ドリブンプログラミングは，現実の分散システムで広く用いられているプログラミング手法である．Web アプリケーションやリアクティブなロボット，IoT システムの実装はその代表例である．例えばチャットアプリケーションなどでは，利用者によるメッセージ入力の一つ一つがイベントとして扱われ，それに対応するイベントハンドラが実行される．コレオグラフィでイベント・ドリブンなプログラムを

記述できるかどうかは，コレオグラフィック・プログラミングを現実の分散システムへ適用する上での重要な問題である．

イベント・ドリブンプログラミングの大きな特徴は，イベントが非決定的に発生することである．また，そのイベントの発生順序もプログラム記述時に予測することができない．例として，alice と bob の二人が参加するチャットアプリケーションを考える．alice がメッセージを入力すると bob の画面に表示され，bob がメッセージを入力すると alice の画面に表示される．このとき，次に入力するのが alice と bob のいずれであるか，それがいつ発生するのか，そしてどのような順序で入力が続くのかは，いずれも事前に分からない．二人が交互に入力するとは限らず，alice が続けて二度入力した後に，bob が入力することもあれば，一方が一度も入力することなしに終了することもあり得る．

既存のコレオグラフィック・プログラミングを用いて，このようなイベント・ドリブンなプログラムを記述する素朴な方法は，ループ内で全てのプロセスを順に訪問し，イベントが発生していないかを確認するポーリングを実装する方法である．図 3 に，上記チャットアプリケーションをポーリングによって記述したコレオグラフィの例を示す．このコレオグラフィでは，2 行目で alice のイベントの発生の有無を確認し，発生していれば入力された内容を bob に送信して表示する（3 行目から 6 行目）．8 行目から 12 行目は bob について同様の処理である．ループごとにこれを繰り返す．

この方法には，イベントが発生していない場合であってもループごとに通信が生じるという問題がある．図 3 のコレオグラフィだけをみると，イベントが発生した場合のみに通信が行われるように見えるが，実際にはそうはならない．図 4 に，図 3 のコレオグラフィの alice への射影を示す．bob への射影は，alice への射影と対称であるため省略する．図 3 の 3 行目の条件は alice 上で評価される一方で，その分岐の本体である 4，5 行目の処理には bob が関係する．そこで，KoC を満たすために alice から条件式の評価結果が bob へ通知される．図 4 の 4 行目の `send(bob, true)` や 7 行目の `send(bob, false)` がこの通知に該当する．図 4 の 7 行目から分かるように，この通知はイベントが発生しなかった際にも必要である．7 行目を除去しイベントが発生した際のみ通知を行うように修正することはできない．仮にそのように修正し，alice から通知が届かなかった際に，bob が，alice がイベントを検出しなかったのか，それとも alice がまだこの地点に到達していないだけなのかを判断することができないためである．

```
1  loop {
2      var ea = poll<@alice>()
3      if (ea != nil) {
4          var msg@bob <- ea
5          print(msg)
6      }
7
8      var eb = poll<@bob>()
9      if (eb != nil) {
10         var msg@alice <- eb
11         print(msg)
12     }
13 }
```

図 3　ポーリングによるイベントディスパッチループ

```
1  loop {
2      var ea = poll()
3      if (ea != nil) {
4          send(bob, true)
5          send(bob, ea)
6      } else {
7          send(bob, false)
8      }
9
10     if (recv(bob)) {
11         var msg = recv(bob)
12         print(msg)
13     }
14 }
```

図 4　図 3 の alice への射影

ポーリングの間隔を広げることで通信量を削減する方策も考えられるが，これは通信量とハンドラ関数の応答性の間のトレードオフにおいて，選ぶ点を移動させているに過ぎず，根本的な問題を解決しているとは言えない．各プロセスを訪問する間隔を広く

取れば，単位時間あたりの通信量は減少するが，次にそのプロセスを訪問するまでの時間が延びるため，ハンドラ関数の応答性が低下する．

3 Promise を備えたコレオグラフィック・プログラミング言語 Renga の提案

我々はイベント・ドリブンなコレオグラフィを記述できるコレオグラフィック・プログラミング言語である Renga を提案する．Renga は JavaScript と Go に影響を受けた構文を持つ独自の言語であり，コンパイラによって JavaScript プログラムに変換され，JavaScript ランタイムを用いて実行されるように設計されている．本章では，Renga において promise がどのように設計されているかを述べた上で，その解決を全てのプロセスで逐次化する仕組みと，EPP により各プロセスの JavaScript ランタイム上での promise の実装方法を説明する．

3.1 Promise の設計

Renga は JavaScript 風の promise をプリミティブオブジェクトとして提供する．図 5 は，図 3 のイベントディスパッチループを，promise を用いて書き直したコレオグラフィを示す．以降では，この例に沿って，promise の作成，ハンドラ関数の登録，およびイベント発生に伴いどのようにハンドラ関数が呼び出されるのかを説明する．

promise は `new Promise(`*kind*`@`*p*`)` によって作成する．*p* はイベントが発生するプロセスであり，*kind* はそのプロセス *p* のイベント機構が検出するイベントの種別である．これらのイベント種別は全て組み込みで提供される．図 5 の 2 行目の `new Promise(input@alice)` は，alice における入力イベントに対応する promise を生成する．作成される promise のロケーションは *p* であり，*p* 以外のプロセスには射影されない．promise オブジェクトは，`then` と `resolve` の二つのメソッドを持つ．

then then は，作成した promise にハンドラ関数を登録するメソッドである．ハンドラ関数の本体はコレオグラフィであり，コレオグラフィの他の部分と同様に，通信や条件分岐を自由に記述できる．図 5 の 4 行目から 6 行目がこれにあたり，イベントが発生するプロセスが alice である（2 行目）にも関わらず，本体には bob に関連する処理である，bob への通信（4 行目）や bob における出力（5 行目）が含まれる．また，登録されたハンドラ関数は一度のみ実行される．そのためハンドラ関数を繰り返し実行したい場合は，ハンドラ関数の中で改めて promise を作成して，登録し直す必要がある．図 5 では，ハンドラ関数の末尾で自身を含む関数 `listen_alice` を再び呼び出すことで（6 行目）これを実現している．

```
1  func listen_alice() {
2      var promise = new Promise(input@alice)
3      promise.then(func (v @alice) {
4          var msg@bob <- v
5          print(msg)
6          listen_alice()
7      })
8  }
9
10 func listen_bob() {
11     var promise = new Promise(input@bob)
12     promise.then(func (v @bob) {
13         var msg@alice <- v
14         print(msg)
15         listen_bob()
16     })
17 }
18
19 listen_alice()
20 listen_bob()
```

図 5　promise を用いて書き直したコレオグラフィ例

resolve resolve は，対応するイベントが発生した際に呼ばれ，登録されたハンドラ関数を実行可能にするメソッドである．ハンドラ関数が実行可能になることを，promise が解決されると呼ぶ．解決を行うのはプログラマではなく，Renga のランタイム・システムである．各プロセスのイベント機構は，キーボード入力やタイマーイベントなどを検知すると，内部的に対応する promise の resolve メソッドを呼ぶ．プログラマがコレオグラフィに resolve の呼び出しを直接記述することはできない．promise が解決されると，実行すべきハンドラ関数の識別子が全てのプロセスに

送られ，各プロセスにそのハンドラ関数の実行が指示される．このとき，イベント内容を表す値がハンドラ関数の引数に渡される．図 5 の 3 行目の引数 v は，alice の入力文字列を受け取る．この値はイベントの発生したプロセスに留まるため，引数のロケーションとイベントの発生するプロセスは一致している必要がある．

コレオグラフィの記述によっては，異なる promise がほぼ同時に解決され得る．例えば，図 5 の 2 行目で作成された alice の入力イベントを待機する promise と，11 行目で作成された bob の入力イベントを待機する promise がこれにあたる．このとき，各プロセスが独立に promise を解決し，ハンドラ関数の実行を指示すると，プロセスごとに実行されるハンドラ関数の順序が入れ替わり，通信の対応が崩れる．これはデッドロックを引き起こすことがある．

そこで Renga は，promise の解決を全プロセスで逐次化する．同時に複数のプロセスでイベントが発生した場合，それらの promise の解決は逐次化され，対応するハンドラ関数も同順で一つずつ実行される．これにより，イベント・ドリブンなコレオグラフィにおいても，引き続きデッドロックフリーが保たれる．

3.2 Promise の解決の逐次化

複数の異なるプロセスでイベントが発生した際の promise の解決を逐次化するために，Renga では中央集権的な方法を採用し，逐次化プロセスと呼ぶ特別なプロセスを用意する．逐次化プロセスは Renga のランタイム・システムの一部であり，単一の大域キューを持つ．あるプロセスのイベント機構がイベントを検出すると，そのプロセスは対応する promise の解決とハンドラ関数の実行の指示を直ちに行うのではなく，まずハンドラ関数の識別子を逐次化プロセスに送る．逐次化プロセスは受け取った識別子を到着順に大域キューに格納する．

逐次化プロセスは大域キューの先頭から順に識別子を一つずつ取り出し，promise を解決し，全てのプロセスに，対応するハンドラ関数の識別子を送り，実行を指示する．これにより promise の解決を逐次化する．逐次化プロセスは，直前に指示したハンドラ関数の実行が全てのプロセスで終了することを待たない．代わりに，各プロセスもまた自身の実行キューを持ち，配送された識別子をこのキューに入れる．プログラムの他の部分の実行が終了した後に，このキューの先頭から識別子を一つずつ取り出し，対応するハンドラ関数の射影を実行する．このとき，一つの実行が完了するまで，次の識別子を取り出すことはない．また識別子が指すハンドラ関数の中に，自身のプロセスに関連する処理が含まれない場合もある．この場合は何も実行せず，次に実行すべきハンドラ関数の識別子を実行キューから取り出す．

なお，逐次化プロセスが各プロセスに送信するのは promise に対応するハンドラ関数の識別子のみである．イベントの発生時に得られた値は，イベントが発生したプロセスに留まり，そのプロセスへのハンドラ関数の射影の引数としてのみ渡される．

3.3 Endpoint Projection（EPP）

Renga のプログラムは，Renga コンパイラが各プロセスへの射影を JavaScript プログラムとして生成し，それを JavaScript ランタイム上で実行することにより動作する．そのため，各プロセスのイベント機構も JavaScript 上に実装されている．本節では，このイベント機構が提供するメソッドを用いて，promise を含むコレオグラフィが実際にどのように射影されるかを示す．

図 6 に，図 5 のコレオグラフィの alice への射影を示す．1，2 行目では，通信を担う `Network` クラスと，イベント機構の本体である `Scheduler` クラスをインスタンス化している．これらはコレオグラフィには現れず，コンパイラが自動で挿入する．

promise の作成は，イベント機構の `newPromise` メソッドの呼び出しに射影される．図 5 の 2 行目の `new Promise(input@alice)` は，図 6 の 5 行目の `s.newPromise("input")` にあたる．一方で図 5 の 11 行目の `new Promise(input@bob)` は，ロケーションが bob の promise を作成するため，alice への射影である図 6 には現れない．

ハンドラ関数を登録する `then` は，コレオグラフィでは promise のメソッドであるのに対し，射影後は

イベント機構のメソッドとなる．実際に，図 5 の 3 行目の `promise.then` が図 6 の 6 行目の `s.then` に射影されることが確認できる．promise の `then` メソッドは，promise のロケーションに関係なく射影される．これは，promise のロケーションがプロセス p であるときであっても，ハンドラ関数の本体には p 以外の他のプロセスに関係する処理を含みうるからである．実際に，図 5 の 12 行目の `then` はロケーションが bob の promise に対する呼び出しであるが，その本体には，bob から alice への通信（13 行目），alice への出力（14 行目），および `listen_bob` の再帰呼び出し（15 行目）が含まれる．

全てのプロセスは同一のコレオグラフィから射影されるため，`then` を同じ順序で呼び出す．これにより，呼び出し順に連番を識別子として割り当てるだけで，同一ハンドラ関数に対して，全てのプロセスで共通の識別子が得られる．

図 5 の 19，20 行目のように関数の外に記述された部分は，図 6 の 20 行目から 23 行目のように，alice におけるイベント機構のエントリーポイントである `run` メソッドに渡される．これにより最初の実行単位として実行される．

4 関連研究

Lugovic らの Real-World Choreographic Programming [2] では，イベントが発生し得るプロセスごとに異なるコレオグラフィスレッドを割り当てて並列に実行する．これらのコレオグラフィ間の通信はコレオグラフィ内で直接サポートしない．プログラマはコレオグラフィ外の低レイヤーの同期機構やデータ共有機構を用いて，自身の手で管理する必要がある．これにより，プログラマの実装次第では，低レイヤーの同期機構や共有機能の使用方法によりデッドロックや競合状態が発生することがある．

Montesi による Choreographic Choice [4] では，非決定的なコレオグラフィの選択をサポートする意味論を与えている．イベント・ドリブンプログラミングにおいても，非決定的なイベント発生に基づき，ハンドラ関数を非決定的に実行する必要があるため，この点で似ている．一方で，この Choreographic Choice を用いてイベントディスパッチループを書く方法では，イベント・ドリブンプログラミングで多く見られるユースケースと照らし合わせると課題が残る．Choreographic Choice は発生するイベントの順序を全て列挙する必要があるが，この記述量は Renga を用いた場合と比べてはるかに大きく，多くのイベント・ドリブンなシステムで見られる，任意の順序でイベントが発生し得る要件を考えると，現実的ではない．

```
1  const n = new Network();
2  const s = new Scheduler(n);
3
4  function listen_alice() {
5    const promise = s.newPromise("input");
6    s.then(promise, async (v) => {
7      n.send("bob", v);
8      listen_alice();
9    });
10 }
11
12 function listen_bob() {
13   s.then(null, async () => {
14     const msg = await n.recv("bob");
15     print(msg);
16     listen_bob();
17   });
18 }
19
20 s.run(async () => {
21   listen_alice();
22   listen_bob();
23 });
```

図 6　図 5 の alice への射影

5 結論

本論文では，イベントの発生によって解決される promise を備えたコレオグラフィック・プログラミング言語である Renga を提案した．プログラマは Renga を用いることで，あるプロセスで発生するイベントに対応する promise を作成し，`then` メソッドによってハンドラ関数を登録できる．

また，promise の解決を逐次化プロセスの大域キューを用いて逐次化することで，複数プロセスでイベントがほぼ同時に発生した場合でも，全てのプロセスで同一の順序で解決され，一つ一つ実行される．これによ

り競合状態やデッドロックが発生せず，安全性が損なわれない．最後に，この promise が各プロセスのイベント機構のインターフェースを用いた EPP を通じて，どのように実装されるかを示した．

参 考 文 献

[1] Hirsch, A. K. and Garg, D.: Pirouette: higher-order typed functional choreographies, *Proc. ACM Program. Lang.*, Vol. 6, No. POPL(2022).

[2] Lugovic, L. and Montesi, F.: Real-World Choreographic Programming: An Experience Report, *CoRR*, Vol. abs/2303.03983(2023).

[3] Montesi, F.: *Choreographic Programming*, Ph.D. thesis, IT University of Copenhagen, 2013. `https://www.fabriziomontesi.com/files/choreographic-programming.pdf`.

[4] Montesi, F.: *Introduction to Choreographies*, Cambridge University Press, 2023.