Alma Mater Studiorum — Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXVIII

Settore Concorsuale di Afferenza: 01/B1
Settore Scientifico Disciplinare: INF/01

Real-World Choreographies

Presentata da: Saverio Giallorenzo

Coordinatore Dottorato:
Paolo Ciaccia

Relatore:
Maurizio Gabbrielli

Correlatore:
Fabrizio Montesi

Esame finale anno 2016

Abstract

Since the early days of the Internet, distributed software applications have become one of the leading forces behind the development and economic growth of our society. Nonetheless, the practice of programming distributed systems is one of the most error-prone. Developers strive to correctly implement separate components that, put together, enact an agreed protocol. If one component fails to follow such protocol, it could lead to system blocks or misbehaviours. Ensuring that all components correctly follow the intended protocol is very difficult due to the inherent non-determinism of distributed programs.

This led practitioners and theoretical researchers to explore new tools to assist the development of distributed systems. Choreographies are one of these tools. They have been introduced to describe from a global viewpoint the exchange of messages among the components of a distributed system. Moreover, since they describe atomic communications (not split into I/Os), they are free from deadlocks and race conditions by design. Recent theoretical results proved that it is possible to define proper Endpoint Projection (EPP) functions to compile choreographic specifications into their single components. Since EPPs are behavioural preserving, projected systems also enjoy freedom from deadlocks and races by construction. Some of these results have been implemented, however much work has to be done to make choreographies a suitable tool for real-world programming.

Aim of this PhD is to formalise non-trivial features of distributed systems with choreographies and to translate our theoretical results into the practice of implemented systems. To this purpose, we provide two main contributions.

The first contribution tackles one of the most challenging features of distributed development: *programming correct and consistent runtime updates of distributed systems*. There is no affirmed technology for structuring runtime updates of distributed applications. Moreover, the non-determinism of distributed systems easily leads to partial applications of updates and to inconsistent systems. Our solution is a theoretical model of dynamic choreographies, called DIOC. DIOC provides a clear definition of which components and behaviours can be updated. We prove that systems compiled from a DIOC definition are correct and consistent after any update. Finally, we refine our theoretical model with constructs for a finer control over updates. On this refinement, we develop a framework for programming adaptable distributed systems, called AIOCJ.

The second contribution covers one of the main issues of implementing theoretical results on choreographies: *formalising the compilation from choreographies to executable programs*. There is a sensible departure between choreographic frameworks like Chor (the first on this paradigm) and AIOCJ and their theoretical models: their theories abstract communications with synchronisation on names (à la CCS/π-calculus) yet they compile to Jolie programs, an executable language that uses *correlation* — a renown technology of Service-Oriented Computing — for message routing. This discontinuity breaks the chain of proven correctness from choreographies to implemented systems. Our solution is a theory of Applied Choreographies (AC) that models correlation-based message passing, useful to pinpoint the key theoretical problems and formalise the guiding principles that developers should follow to obtain correct implementations. Finally, we prove our approach by defining a correct compiler from AC to the calculus behind the Jolie language.

Acknowledgements

If I have seen further, it is by standing on the shoulders of giants.

Sir Isaac Newton

A PhD is a formidable endeavour, not only academically speaking but also at the personal level. It gave to me the opportunity to face my limits, to experiment failure, and, in many senses, to grow up. Thankfully, I was very lucky to have the best people around me. These are my acknowledgements to them all.

I thank my parents, Patrizia and Vito, for supporting all my (mostly eccentric) projects and for giving me the freedom to be wrong. I owe to them my creativity and love for Science. They taught me patience and perseverance, principles that "saved" me countless times and that I like to think constitute some of my best qualities.

In these years, I also contracted an infinite debt of gratitude with my girlfriend Daniela. She has been my most passionate supporter and best friend. Times and times she encouraged me while I lingered in self-doubt and comforted me when I complained about broken formalisations, buggy programs, and rejected papers. I owe to her all the sacrifices she made for me: the days and nights passed alone while I was working for a deadline, the missed vacations, and the long trips to Denmark. I deeply thank her for all the love she gave to me and for accepting a simple "Ti amo" ("I love you") in exchange for all of that.

I dedicate this work to Daniela, Ma, and Pa, for their unbreakable belief in me.

Thanks also to Roberta and Andrea for having accepted this chaotic guy in their family and to Giorgia for all the joyful games and talks about dinosaurs (!).

In my PhD I had the good fortune to have many teachers, whom I thank.

Thanks to my two co-advisors: Maurizio Gabbrielli and Fabrizio Montesi.

I thank Maurizio for having believed in me and that a poor "Scienziato di Internet" (graduate in IT and Management) could pull off some basic research. I owe to Maurizio many thanks: for having set up a nice research team at the beginning of my PhD, for the freedom he has always given to me, for having pushed me beyond my limits, and for cheering me up during my saddest moments.

I owe a lot also to Fabrizio. He is the crazy guy who coded the most part of the Jolie language, which brought me in 2012 to work with him in Copenhagen (again, thanks to Maurizio) while he was working on his PhD on "Choreographic Programming" — whose this thesis can be considered a continuation. Fabrizio dedicated a lot of his time to teach me academic rigour and to seek clarity in the definition of problems before venturing in their resolution. I really enjoyed all the time we spent together trying to bring elegance and balance to our formalisations, Jolie, and choreographies[1].

Thanks to Ivan Lanese for the many times he proved me (and, above all, calmly explained why I was) wrong. I hope I have been able to steal a little of Ivan's mathematical genius and cleverness during our restless sessions on our theoretical model and its proofs.

[1]and, by extension, to the Force.

Thanks to Jacopo — O Captain, My Captain! — Mauro. I had the good fortune to have a big brother like him in the Underground Laboratory in Bologna. Thanks for having given to me stability when I desperately needed some, for all the advices and chats, and for begin a very good friend to me.

Thanks to Claudio Guidi for having introduced me into the academic world, “Galeotto fu il linguaggio (Jolie) e chi lo scrisse[2]” and for all the inspiring conversations we had on languages, abstractions, and “visions”.

Thanks to Marco Carbone, my (now) academic grandfather who (back then) hosted me for my first stay-abroad in Copenhagen. It was a pleasure to know him as a person and I thank him for contributing in making all this possible.

Thanks to Davide Sangiorgi for all the inestimable support he gave to me before and during my PhD, both academically and personally. I am also really grateful to Davide for having let me teach Jolie in his course on Operative Systems (yay!).

Thanks to Gianluigi Zavattaro for the interesting discussions and alternative visions on choreographies and for having given to me the opportunity to present Jolie at the Bertinoro International Spring School, that was really great!

I also want to thank my reviewers, Ilaria Castellani and Thomas Hildebrandt, who not only took the duty of refereeing this thesis but provided encouraging and stimulating comments to improve its contents.

I thank all the members of the INRIA Focus team in Bologna for all the inspiring talks. Thanks to the people in the research group of Programming, Logic and, Semantics at the IT University of Copenhagen and in the department of Mathematics and Computer Science (IMADA) at the Southern Denmark University. I also thank Maja, Borna, Marco, and Luís for the joyful time in Odense.

Finally I owe a great deal of gratitude to all my fellow colleagues and PhD students for having shared with me the joys, the grumbles, the aperitivi, and the awesome discussions in and out of the (Hardcore) Underground Laboratory.

Many thanks to Valeria Vignudelli for the endless hours spent to teach me proof techniques, for all the laughs, and the cheerful pats on the shoulder that I needed.

Thanks to Jean-Marie Madiot for sharing with me many serious (and definitely not serious) talks, many pat-pats on the shoulder, and for that “You don’t know Hacker News?!”.

Thanks to Francesco Gavazzo, for the desperate mission to explain to me Category Theory and for all the fruitful chats on the philosophy of dark knights and the super men.

Thanks also to Mila, Paolo, Gustavo, Giulio, Roberto, Abel and Yiyi (¡gracias por el ron!), Stefano, Vincenzo, Giacomo, (the other) Vincenzo, Federico, and Luca for all the nice time spent together.

Thank you all, my giants.

[2]Accurséd was the language (Jolie) and those who wrote it.

List of Publications

- Dalla Preda, Mila, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. **Dynamic Choreographies: From Theory to Practice.** Submitted.

- Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Fabrizio Montesi, Jacopo Mauro. *Self-configuring Microservices.* In Theory and Practice of Formal Methods, pp. 194–210. Springer International Publishing, 2016.

- Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. **Applied Choreographies.** Submitted.

- Dalla Preda, Mila, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. **Dynamic Choreographies — Safe Runtime Update of Distributed Applications.** In Coordination Models and Languages, pp. 67–82. Springer International Publishing, 2015.

- Dalla Preda, Mila, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. *Developing correct, distributed, adaptive software.* Science of Computer Programming 97, pp. 1–46. Elsevier, 2015.

- Dalla Preda, Mila, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. **AIOCJ: A choreographic framework for safe adaptive distributed applications.** In Software Language Engineering, pp. 161-170. Springer International Publishing, 2014.

- Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. *Service-Oriented Architectures: from design to production exploiting workflow patterns.* In Distributed Computing and Artificial Intelligence, 11th International Conference, pp. 131–139. Springer International Publishing, 2014.

- Claudio Guidi, Saverio Giallorenzo, and Maurizio Gabbrielli. *Towards a Composition-based APIaaS Layer*, In CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science, pp. 425–432. SciTePress, 2014.

Items with titles in **bold** are part of this dissertation.

Contents

List of Figures

CHAPTER 1

Introduction to the dissertation

Here we present the scientific context of our research and the problems that we want to address in this dissertation. The purpose of this chapter is to provide an overview of the structure of this dissertation. We give a brief introduction of the concepts that are extensively presented in the next chapters.

1.1 Problem Description

Since the early days of the Internet [1], distributed software applications have become one of the leading forces behind the development and economic growth of our society. A distributed system is a software application whose behaviour emerges from the interaction of several components (programs) that run in parallel and rely on message passing to communicate and coordinate their actions [2]. News systems, messaging, governance, healthcare, and transportation are just some of the contexts recently revolutionised by distributed applications.

Nonetheless, the practice of programming distributed systems is one of the most error-prone. Developers strive to correctly implement separate components that, put together, enact an agreed global behaviour (protocol). If one or more of said components fail to follow their common protocol, the distributed system can block or misbehave [3, 4]. Unfortunately, ensuring that all components correctly *play* their *part* in the interaction — i.e., that they follow their intended protocol — is very difficult due to the inherent non-determinism of several distributed programs running in parallel.

1.2 Aim and Thesis Statement

Here, we consider "safe" a distributed system in which its components correctly enact the intended global behaviour of the system, i.e., a safe distributed system is one free from defective behaviours due to faulty communications between its components. On this definition, we state that the aim of this dissertation is

to advance the scientific knowledge on programming safe distributed systems.

Having proper language abstractions to express and analyse concurrent and distributed software has proven to be one of the most effective techniques to harness their complexity and to provide guarantees of their safety [5, 6]. Hence, we pursue our aim investigating the linguistic abstractions that can support the development of safe distributed applications.

Choreographies [7] are one of such linguistic abstractions. Introduced for the definition of protocols of interactions [8, 9], they describe the communications between the components of a distributed system from a global point of view. Distinguishing feature of choreographic languages is that they describe communications as atomic entities, i.e., not split into inputs and outputs. Since it is impossible to write mismatched I/Os, distributed systems described in a choreographic language are free from deadlocks and race conditions by design. However, a choreographic description of a distributed system is not directly executable. Recent theoretical results proved [10, 11, 12, 13, 14] that, given a proper choreographic description of a distributed system, it is possible to correctly compile it into its single components, which are also deadlock- and race-free by construction. This makes choreographies a very interesting linguistic abstraction for the development of real-world safe distributed systems. Although promising, the present choreography theories and their implementation prototypes still cover a very limited set of properties of distributed systems. Making choreographies a suitable tool for real-world programming [15] still requires a lot of work. Extensions of the theoretical models are needed to capture desirable requirements of distributed applications, like safety-preserving integrations with existing executable languages (e.g., the possibility to write Java or Scala code directly in the choreography), exception handling [16], and runtime updates [17]. Finally, also the correct implementation of the theoretical models is extremely important to the aim of bringing into the real-world the desirable properties of choreographies. Indeed, providing a formal treatment of such compilation is of paramount importance to guarantee the preservation of the properties of correctness and safety of choreographies down to compiled executable programs.

The thesis of this dissertation affirms that choreographies are a suitable linguistic abstraction that can be successfully extended to support the development of real-world safe distributed systems.

Thesis Statement

Choreographies can be used to program real-world safe distributed systems.

1.3 Contributions

We prove our thesis presenting two main contributions.

The first contribution tackles one of the most challenging features of distributed programming: guaranteeing the correct update of the behaviour of distributed systems at runtime. Indeed, beside the lack of proper abstractions to structure distributed updates, the non-determinism of distributed systems easily lead to partial applications of updates, leaving systems in an inconsistent state. Our solution is a theory of Dynamic Choreographies, called DIOCs. DIOCs can update at runtime in a coherent way and provide a clear definition of which components and behaviours can be changed. We prove that systems compiled from a DIOC definition are correct and consistent after any update. Our theory of dynamic choreographies is deliberately general and can be refined to model specific policies of updates. We extend the DIOC language with constructs for providing a fine control on the updates and we implement AIOCJ, a framework for the development of correct, adaptable distributed systems.

The second contribution covers one of the main issues of implementing theoretical results on choreographies: providing a formal compilation from a choreographic language to an executable language. Our solution is to develop a theory of Applied Choreographies (AC) that models correlation-based message passing. We pinpoint the key theoretical problems and formalise the guiding principles that developers should follow to obtain correct implementations. Finally, we prove our approach by defining a correct compiler from AC to the foundational calculus of the executable language Jolie.

1.4 Structure of the Dissertation

The reminder of this dissertation is structured in three main parts in which we introduce the reasons and issues that motivate our investigations and report our results. All presented results have been produced during the course of the PhD studies and are here presented in an extended form.

- **Part I — Introduction to the Dissertation and Background**
  It comprises this Chapter about the aim, thesis, and structure of this dissertation. In the next Chapter of this Part we present the features and issues of concurrent, distributed programming, the scientific context of our research, and the state of the art, on which we build our results. This introduction is intended to provide the reader with the basic knowledge to interpret the results presented in the next Parts.

- **Part II — Adaptable Choreographies**
  We tackle the problem of programming distributed applications that update their behaviour at runtime, adopting some new, external behaviours that were unforeseen at the time of their development. The Part is divided intro three Chapters.

  In the first Chapter we provide a general overview on runtime software update and adaptation. In particular we focus on the renown technologies and abstractions introduced to enable and structure runtime software updates.

  In the second Chapter we show how choreographies can be modelled into a language, called DIOC, that provides suitable abstractions for programming distributed systems that update at runtime. We prove that DIOCs can be compiled (projected) to sets of processes that correctly enact the indented behaviour of their specification. We also prove that correctness is preserved after any step of update.

  DIOCs provide a general support for software update and do not specify a particular policy for the application of updates (indeed such choice is orthogonal wrt the guarantees of correctness). In the third Chapter of this Part we present a refinement of our theoretical model of DIOCs aimed at modelling adaptation of distributed systems. We report how we implemented such model into a framework, called AIOCJ, for programming safe and adaptable distributed programs.

- **Part III — Applied Choreographies**
  We address the issue of formalising the implementation of choreography models presenting our choreography theory of Applied Choreographies (AC). In particular, we model message passing based on correlation in the semantics of AC. We provide AC with a typing discipline that ensures the correct use of the low-level mechanism of message correlation, thus avoiding communication errors. We also define a two-step compilation from AC to a low-level Correlation Calculus, which is the basis of the executable language Jolie. Finally, we prove an operational correspondence theorem, ensuring that compiled programs behave as the original choreography.

- **Part IV — Conclusion**
  We summarise the contributions of this dissertation and relate them to similar existing work. We also discuss some interesting directions of future investigation.

- **Part V — Appendices**
  - Chapter A — Adaptable Choreographies: Proofs.
  - Chapter B — Adaptable Choreographies: Test Code.
  - Chapter C — Adaptable Choreographies: Models of Adaptation.
  - Chapter D — Applied Choreographies: Additional Material.
  - Chapter E — Applied Choreographies: Proofs.

Part I

Introduction and Background

CHAPTER 2

Introduction to Choreographies

Clara pacta, boni amici.

Latin proverb

2.1 Distributed Programming, in brief

A distributed system is a software application whose behaviour emerges from the interaction of several components (programs) that run in parallel and rely on message passing to communicate and coordinate their actions [2]. Although simple, this definition covers all kinds of distributed systems, ranging from networks of computers to communicating threads of the same program. The practice of writing distributed applications is called *distributed programming*.

Distributed programming born with the introduction of the first wide [18] and local [19] area networks of computers between the late 1970s and the early 1980s. Since that time, distributed programs evolved along the networks that supported their communications. During the 1990s the Internet [1] widely spread to constitute an heterogeneous collection of networks — a network of networks — including wired and wireless sub-networks connected by different technologies (e.g., Ethernet, Bluetooth, WiFi, etc.). Lately, Pervasive Computing [20] and Cloud Computing [21] are major threads of research, ranging from room-scale to global-scale networks of interacting component.

Distributed software constitutes the backbone of our society: it enables people to carry out tasks otherwise unmanageable, automating and ensuring the correct execution of complex social systems. Healthcare, transportation, commerce, and manufacturing are some of the industries that are currently experiencing strong benefits from the integration of software.

The reasons for writing distributed programs can be either compulsory or deliberate. In some cases the nature of the application requires to be distributed, e.g., when production and consumption of data happens in different places. In other cases, distributed programming fits a particular choice of the designer of the system, e.g., a distributed system that uses replicated and (geographically) distributed components to ensure reliability.

More in general, distributed programs benefit from two main features:

- *modularity* as message passing imposes no bonds on the components of a system. Any resource that behaves according to the specification of a component of the system can be employed as such. If said resource fails or becomes busy, the system might replace it with a new, compliant one by simply redirecting its messages.

- *scalability*, both in functions and size. Once deployed, a distributed program can become itself a component employed by another program, enhancing its functionalities with minimal effort. In the same way, since components are modular, if the workload on a component can be shared, it is possible to add a new instance of the same component in the system and separate the workload between the two.

Although distributed programming contributed to shape the present, the practice of correctly developing and debugging distributed systems remains one of the most challenging.

2.1.1 Problems of Distributed Programming

Distributed systems are developed to implement some protocol of interaction.

Let us take as an example the sequence of messages exchanged between an ATM terminal, the bank accounting system, and the issuer of the credit card to approve a withdrawal. When a client requires a withdrawal, the ATM forwards to the bank the *card ID* and the inserted *PIN*. Next the bank asks the card issuer to validate the request. Finally, the card issuer approves or denies the withdrawal and the bank forwards the outcome to the ATM. We depict in Figure 2.1 the sequence chart diagram of the example.

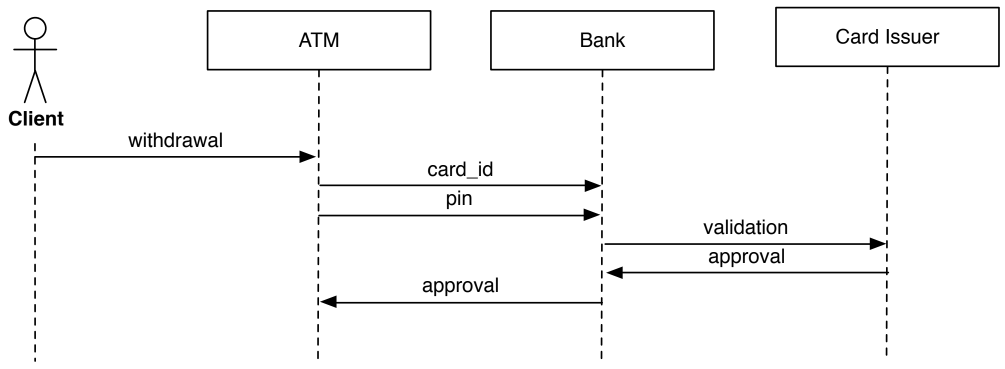


Figure 2.1: Sequence chart of the ATM withdrawal example.

Since the early days of distributed computing, developers introduced and used several tools to precisely describe the order of interactions between the components of a system, like message sequence charts [22] and sequence diagrams [23]. Baseline of all these tools is to avoid or at least minimise the ambiguity of the description of the sequence of messages in the system. However, developers struggle to make sure that the final interplay between the components of a distributed system correctly implement the global description of that system. The problem here is that they have to infer the logic of each component from a global protocol where interactions are considered as a whole and not broken down to sequences of send and receive actions.

Failing to correctly implement the global protocol of interaction of a system is the main source of system freezes and misbehaviours. Such errors are classified in literature as *deadlocks* [3] and *race conditions* [4].

Deadlocks. Deadlocks are one of the main causes of blocks of distributed systems. A system is in a *deadlock* state when one or more components hang, waiting for another component to release a resource (e.g., a message). The components cannot proceed with their computation and the whole system blocks.

Deadlocks usually occur when one of the components of a distributed application does not implement the global protocol of the system.

As an example, assume that the developer of the ATM mistakenly reversed the first two interactions with the bank. As a result, the system ends up in a deadlock state because *i*) the ATM hangs waiting for the bank to receive the message carrying the *PIN* whilst *ii*) the bank waits indefinitely for the message carrying the *card ID*.

Race Conditions. Race conditions are even subtler errors than deadlocks. When a system has a race condition, the runtime behaviour of an application depends on the timing or the sequencing of communications among its components. Due to their non-deterministic nature, race conditions are one of the most difficult bugs to spot and may potentially lead to worst scenarios than the block of the system; among these are data inconsistency and information leakage.

On the example in Figure 2.1, let us assume that the ATM sends the two messages for the *card ID* and *PIN* in parallel to the bank. Contrarily, the implementation of the bank is sequential: it waits to first receive the *card ID* and then the *PIN*. In this case we have two scenarios, depending on the order in which the bank receives the messages sent from the ATM. If the bank receives the *card ID* first then the interaction proceeds as intended. Otherwise the system ends up in a deadlock state where the bank waits to receive a message carrying the *card ID* whilst the first consumable message is the one carrying the *PIN*.

To avoid these faulty executions, programmers must make sure that all components of a distributed system correctly follow its designed protocol, however such strong guarantee is very difficult to assess.

On one hand, the static analysis of the non-deterministic interaction of many programs, executing in parallel, easily leads to the computation of exponentially many traces of execution, which makes such analysis computationally impractical.

On the other hand, non-determinism makes also the practice of testing and debugging the system unreliable. Plus, corner cases may be hard to predict and reproduce in all their possible combinations.

In the following sections we provide a brief introduction to the theoretical and practical solutions that have recently been proposed and adopted to ease the development of *safe* distributed systems. We introduce Service Oriented-Computing (SOC) and how SOC can be implemented following two opposite approaches, namely the Orchestration or the Choreographic approach. Finally, we present some seminal theoretical works that paved the way for the employment of choreographies as language abstraction for the development of safe distributed systems.

2.2 The Service-Oriented Approach

The World Wide Web [24] (WWW or Web for short) is an established technology that, although invented to provide human-readable content to users, has gone acquiring importance also for distributed programming.

Technologies like Uniform Resource Locators (URLs) [25], HyperText Transfer Protocol [26] (HTTP), and the eXtensible Markup Language [27] (XML) provided a common platform to make distributed programs communicate. In the context of the Web, distributed systems are composed by *Web services* that offer a specific interface of interaction. The World Wide Web Consortium (W3C) defines a Web Service [28] as:

> [. . .] a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [29]). Other systems interact with the Web service in a manner prescribed by its description using SOAP [30] messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Paraphrasing the above definition, Web services pose no restriction on the internal technologies that implement their functionalities. Provided that a service describes its interface of interaction in a machine-processable format (WSDL) and interacts through SOAP messages, it will be able to interoperate with other Web services by receiving and sending requests through messages.

Notably, the computing paradigm described by Web services is not bound to Web standards but can be further abstracted to the so called *Service-Oriented* [31] paradigm.

Service orientation is platform-agnostic, it abstracts from the underlying technologies that enable components to interact in a network, and dictates that any resource (i.e., any program) of the network shall self-describe its functionalities through a standard interface. Services offer their functionalities following the client-server principle. In particular service orientation hinges on two principles:

- *Neutrality*: clients must be able to require functions of services through open, standardised, and wide-adopted technologies. Therefore all the layers that support the invocation of a request should comply with widely accepted standards.

- *Loose Coupling*: clients and services must agree only on the terms of the invocation of a function. Their internal implementations remain unknown to each other and the interaction does not rely on any knowledge of the internal structures that enact the execution of that functionality.

The service-oriented paradigm applies to any context that *i*) provides a common channel of communication for programs which *ii*) comply with some shared specification regarding protocols of communication and serialisation of data. Other than the Web stack, some renown technologies that apply the service-oriented approach are *D-Bus* [32] and *DCOM* [33], which are respectively an open-source and a proprietary message bus that provide a context for one-to-one message passing for applications of the same system. Enterprise Service Bus [34] (ESB) technologies offer an analogous support for communication between distributed programs. In particular ESBs usually offer built-in transparent protocol conversion (e.g., HTTP, SOAP, DCOM, SAP RFC [35], etc.).

2.2.1 Breaking down complexity

The main novelty introduced by service orientation regards a new architectural perspective for organising distributed systems. Since the construction blocks of SOC are services, a service-oriented distributed system results from the composition of interacting services that form a Service-Oriented Architecture (SOA). The purpose of developing a distributed system as an SOA is to break down its complexity into simple services and build up the system in a hierarchical way.

This has several benefits, among which:

- *Parallel and compartmentalised development.* Since services communicate only through their standardised interfaces, an SOA can be quickly build up by different teams of developers working in parallel on a distinct subset of such services.
- *Reuse.* Usually projects share a common set of functionalities like logging, monitoring or security. Since services are compositional, it is easy to include an existing service that provides one of the required functionalities into a new project. Beside fostering re-usability of code, it also naturally lends itself to a market of specialised third-party services.
- *Simpler debugging.* Services that accomplish one simple task are easier to test and check for compliance (wrt specifics) that more complex ones.
- *Systems of systems.* SOAs can be built on top of other SOAs. The guiding principle remains that of ease of testability. Given a set of "simple" services, a first level of "simple" (to test) SOAs can accomplish more elaborate tasks as composition of such services. Iteratively, the most complex behaviours of a distributed system result from high-level SOAs that compose other architectures.

These general ideas gave birth to two opposite ways of composing services [7]:

Orchestration where the logic and order of the interactions in an SOA are defined and controlled by a single party, called the *orchestrator*.

Choreography which defines the sequence of messages each participant exchanges. Each party implements its own part and the intended interaction emerges from their interplay.

In the next section we present a brief introduction to the concept of orchestrated services and discuss the relationship between the former and choreography.

2.3 Orchestration

Nowadays, orchestration is the most adopted solution for service composition and the main reasons behind its wide adoption are *i*) that exposing and composing legacy programs (already developed and owned by businesses) requires little to no adjustments and *ii*) orchestrators become themselves new services that other orchestrators can use in their compositions.

Whilst the industrial standard for orchestration is WS-BPEL [36] (BPEL for short) all orchestration languages share a common base of functionalities to compose services at message level:

- *asynchronous* or *synchronous* delivery and reception of messages;
- sequential composition — an interaction can occur only after the one that precedes it;
- parallel composition — two or more interactions can occur in parallel;
- guarded composition — only one of a set of interactions can occur while the others are discarded.

As an example, we report in Figure 2.2 the orchestrated implementation of the protocol presented in Figure 2.1.

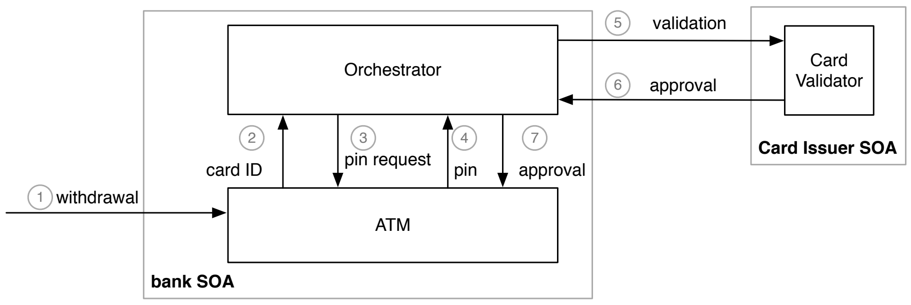


Figure 2.2: Example of orchestration.

In the example, since the bank is the owner of the SOA to which also the ATM belongs, we do not have a "bank" service but rather we introduce the Orchestrator service that implements the withdrawal protocol.

Following the orchestrative approach, the Orchestrator coordinates all interactions among the services in the systems. When the ATM receives a withdrawal request ①, it sends the *card ID* to the orchestrator. Then ② the Orchestrator asks the ATM for the *PIN*. After having received the *PIN* ③, the Orchestrator asks to validate the request to the Card Validator service (offered and owned by the card issuer). Finally, the Orchestrator receives the outcome of the validation (approval) ⑥ and ⑦ forwards it to the ATM.

In orchestration the logic of a distributed system is mostly implemented and enclosed in the orchestrator and therefore it becomes easier to check and test compliance between protocols and implementations.

However, orchestration poses a strong limitation in the development of distributed systems because it requires a central entity that manages the whole interaction whilst in many distributed contexts that assumption might not hold. For example, OpenID [37] is a widely adopted protocol that provides authentication in a decentralised way. In OpenID a service (e.g., a web site) delegates a relying party to authenticate users. The benefits are twofold: services do not have to manage users and users avoid to give their credentials to possibly untrusted services. In these cases, it is not possible to orchestrate the interaction and the only available solution is to develop each component separately.

2.4 Choreography

Unlike orchestration, choreography has no central point of coordination and describes SOAs in terms of interactions between the participant services. In a choreography all services are peers and the logic of the composition is spread among them. Choreographic composition is an unexplored topic wrt orchestration. In fact, the paradigm has been mostly used for protocol specification [8, 9].

W3C proposed a candidate recommendation, called WS-CDL [8]. In WS-CDL the developer specifies the protocol of observable interactions between services from a global point of view. As already remarked, the main reason behind the need for a choreographic composition is that the orchestration paradigm requires a full control on outputs and inputs toward and from the invoked services. On the contrary, there exist various systems that are composed by several services that communicate to each other but cannot be coordinated from a centralised position.

However, WS-CDL is not a language for programming composition of services, it is rather a formal language for specifying protocols, i.e., the correct sequences of interactions in the composition. Quoting the definition of the purpose of WS-CDL given by the W3C:

> The WS-CDL specification is aimed at being able to precisely describe collaborations between any type of participant regardless of the supporting platform or programming model used by the implementation of the hosting environment.
>
> Using the WS-CDL specification, a contract containing a "global" definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a global viewpoint, the common and complementary observable behaviour of all the participants involved. Each participant can then use the global definition to build and test solutions that conform to it. The global specification is in turn realized by combination of the resulting local systems, on the basis of appropriate infrastructure support.

Following to the above declaration of purposes, a choreography describes the interactions between several participants but it does not provide an implementation of such interaction. It is up to each participant to correctly implement its individual part in the choreography.

Beside the interpretation of choreographies given by the W3C, several works investigated the possibility to use choreographies as abstractions to implement distributed systems.

For example, OASIS proposed a different perspective to choreographic composition with BPEL4Chor [38]. Whilst WS-CDL sees choreographies as a top-down tool that can be used to specify protocols, OASIS considers choreographies from bottom-up, as an abstraction to define interoperability.

In particular, BPEL4Chor introduces a layer over BPEL where the developer defines only the activities of sending and reception of messages. Then, given an abstract specification in BPEL4Chor, the developer can implement the components that interact in the choreography as BPEL services. Once a BPEL4Chor choreography has been "grounded" into several BPEL orchestrators (one for each participant) the choreography is enacted by the parallel execution of the implemented orchestrators.

The two approaches have interests at odds (analysed in detail in [12]):

- top-down choreographies, à la WS-CDL, benefit from the maximum degree of abstraction, focussing only on the protocol of interaction. Communications are atomic and therefore they inherently exclude deadlocks and races (as seen in § 2.1.1). However, these kind of interaction-oriented choreographies are meant as design tools and cannot be used to implement distributed systems.

- contrarily, bottom-up choreographies written in BPEL4Chor are implementable, allowing developers to follow a common schema of interaction. However, since these process-oriented choreographies maintain such close relation to the processes that implement the distributed system, they model communications as sequences of send and receive actions, possibly leading to deadlocks and races.

To exemplify the difference between the interaction-oriented and the process-oriented approach, we report in Figure 2.3 how the two approaches would model the scenario presented in Figure 2.1.

In the example we use the “Alice and Bob” notation [39] for interaction-oriented choreographies, where $\texttt{A} \rightarrow \texttt{B} : action$ represents a communication between two endpoints (components) of the system, specifically $\texttt{A}$ sending a message through an $action$ to $\texttt{B}$, and ; is the sequential composition of interactions. For process-oriented choreographies we use the notations $\texttt{from A} : action$ and $\texttt{to A} : action$ to respectively represent the receiving from and the sending to the endpoint $\texttt{A}$ through $action$, ; still means sequential composition.

$$\begin{array}{l}
\texttt{Client} \rightarrow \texttt{ATM} : withdrawal;\\
\texttt{ATM} \rightarrow \texttt{Bank} : card\_id;\\
\texttt{ATM} \rightarrow \texttt{Bank} : pin;\\
\texttt{Bank} \rightarrow \texttt{Card Issuer} : validation;\\
\texttt{Card Issuer} \rightarrow \texttt{Bank} : approval;\\
\texttt{Bank} \rightarrow \texttt{ATM} : approval
\end{array}$$

$\texttt{ATM}$ **process**	$\texttt{Bank}$ **process**	$\texttt{Card Issuer}$ **process**
$\texttt{from Client} : withdrawal;$	$\texttt{from ATM} : card\_id;$	$\texttt{from Card Issuer} : validation;$
$\texttt{to Bank} : card\_id;$	$\texttt{from ATM} : pin;$	$\texttt{to Bank} : approval$
$\texttt{to Bank} : pin;$	$\texttt{to Card Issuer} : validation;$	
$\texttt{from Bank} : approval$	$\texttt{from Card Issuer} : approval;$	
	$\texttt{to ATM} : approval$	

Figure 2.3: Upper part: example of interaction-oriented choreography. Lower part: from left to right, the process-oriented choreographies of the ATM, the Bank, and the Card Issuer.

2.5 Programming with Choreographies

Recent theoretical investigations explored how the interaction- and process-oriented aspects of choreographies could be merged into one language, namely one that enjoys the minimality and safety (deadlock- and race-freedom) of top-down choreographies but able to express process-level interactions, needed to implement distributed systems.

Seminal work in such endeavour are [10, 40, 11].

In [10] Qiu et al. give a first theoretical account of the fundamental issues between interaction-oriented choreographies and implementations. In that work, the authors introduce the concept of *projection* to map the global behaviour of a distributed system, described as a choreography, into a set of processes that implement it.

In [40], Bravetti and Zavattaro build on the concept of *service contracts* [41] — i.e., descriptions of the behaviour of services in terms of their inputs and outputs — and define a projection that, given an interaction-oriented choreography, generates a set of contracts, one for each of its participants. Then, it is possible to implement the original choreography by executing those services whose contracts match the generated ones.

Finally, in [11] Carbone at al. formalise a core WS-CDL language and relate interaction-oriented choreographies and implementations by means of an Endpoint Projection (EPP) function. The EPP is a mapping from a choreography specification to a set of processes which, run in parallel, enact the behaviour described by the choreography. The target of the EPP is an applied π-calculus [5] and not an actual executable language. However in [11] the authors prove that interaction-oriented choreographies can be made expressive enough to define the implementation of safe distributed systems. Essentially, the projected processes enact all and only the behaviours described in the choreography (protocol) and since choreographies cannot express deadlocks and races also the projected system is deadlock- and race-free.

The most notable results of [11] are that *i*) it paved the way for the concept of choreographic programming [42] — i.e., to use choreographies as abstraction to implement distributed systems —, *ii*) it inspired the introduction of *multiparty session types* [43] as evolution of *binary session types* [44], and *iii*) it provided a methodology for the development of distributed software, based on a correctness-by-construction approach, which we can depict as:

$$\begin{tikzcd}
\boxed{\text{Choreography}} \arrow[r, "EPP"] & \boxed{\text{Endpoint Projection}} \\
\textit{(Correct by design)} & \textit{(Correct by construction)}
\end{tikzcd}$$

Subsequent theoretical works [12, 13, 14] followed a similar approach, extending the choreography model to support multiparty sessions, channel mobility, and modularity.

On the other side of the spectrum, some early works [45, 46, 47] explored how choreographies could be used to support the implementation of distributed programs, however none of these proposals uses choreographies as a programming abstraction and rather employs them to check endpoint programs.

Chor [48] is the first work that brought the theoretical results on choreographies into the world of implementation languages. Based on the theoretical framework presented in [13], Chor supports the definition of global descriptions (protocols), the programming of compliant choreographies, and the safe projection of said choreographies into a collection of distributable and executable orchestrators.

Chor gives a tangible proof that choreographic programming is a suitable paradigm for the implementation of real-world distributed systems because: *i*) it lets developers focus on the description of the interactions between the components in the system and *ii*) it generates deadlock- and race-free distributed applications that are guaranteed to follow the designed protocol.

Nonetheless, Chor is just the first attempt at bringing into the real world the promising theoretical results on choreographies. The language lacks some standard features of programming languages like *modularity*[1], *dynamic updates* or *error handling*. Beside the expressiveness of the language, the major limitation of Chor is that its EPP is not formally proven. The theoretical treatment on which Chor is based uses as target language a process calculus that models communications through synchronisations on names (as in CCS [49] and the π-calculus [5]). Contrarily, the EPP equipped with Chor targets Jolie [50], which is an orchestration language that uses correlation (à la BPEL) to route messages. Hence, to ensure that programs projected from Chor correctly implement their originating choreography, it is necessary to formalise and prove an Endpoint Projection procedure for the targeted executable language.

2.6 Contributions

In Chapters 4–5 we give a positive answer to the question:

Can we model safe and dynamic updates of distributed systems with choreographies?

In Chapter 4 we present our theoretical model of Dynamic Choreographies and our results of correctness of projection and runtime update. In Chapter 5 we refine our theoretical model of Dynamic Choreographies to provide a fine control over updates. As a result we develop a programming framework for developing *adaptive* [51] (cf. Chapter 3) distributed systems. Like Chor, AIOCJ proves that choreographic programming can be used to deal with different and real-world application domains, preserving the desirable properties of correctness and safety of interaction-oriented choreographies.

[1]The possibility to compose different choreographies — or better said, different systems projected from different choreographies — to implement applications in a compositional way.

In Chapter 6 we progress our venture by tackling one of the most important problems of bringing choreographies to the real world. As discussed in § 2.5, Chor — but also AIOCJ — is based on a theoretical model that abstracts the mechanics of communication with synchronisation on names. Although such abstraction allows for an elegant theoretical treatment, it sensibly departs from the reality of distributed systems, where some underlying technology must be set and used to support the routing of messages between processes.

As a result, the implementations of Chor and AIOCJ loosely follow the EPP defined in their foundational models. This breaks the chain of guarantee of correctness from choreographies to the implemented systems. For instance, to establish the communication channels between the processes in the system, the programs projected by Chor and AIOCJ make some preliminary communications, unexpected by their theoretical models. Whether those communications are deadlock or race-free and that they do not interfere with the behaviour defined at the choreography level is not investigated nor theoretically proven. Moreover, these additional interactions weaken one of the main advantages of choreographic programming: the clear specification of the communications carried out during execution. The leading research question that we pose in Chapter 6 is therefore:

Can we formalise the implementation of communications in choreographies?

Interestingly, this question easily overcomes the boundaries of choreographies. The problem of bringing the results on name-based theoretical models for distributed systems into the real world of implemented languages has been faced in various works, following different custom practices [52, 53, 54]. Therefore, solving the problem of formalising the implementation of communications in choreographies would be a useful reference to formalise the implementation of name-based languages in general.

Our answer, presented in Chapter 6, is to develop a theoretical model for Applied Choreographies (AC). ACs support the most advanced features of recent choreography models like modularity [14], asynchrony, parallelism, and the dynamic creation of multiparty sessions and processes, yet it models message-passing based on *correlation*, one of the main technologies used in Service-Oriented Computing (BPEL, Java/JMS/ C#/.NET, etc.) to support message routing.

On the definition of AC, we pinpoint the key theoretical problems and formalise the guiding principles that developers should follow to obtain correct implementations. Finally, to prove our approach, we define a correct compiler from AC choreographies to the theoretical model of the Jolie language.

Part II

Adaptable Choreographies

CHAPTER 3

Runtime Software Update and Adaptation

In Chapter 2 we discussed how distributed software has become a commodity used on a daily basis like phone lines, electricity, and water.

As our society becomes more and more dependent on software, it also demands for solutions that quickly and continuously update and adapt to the changes in objectives and the surrounding environment. Software systems that respond to change by self-adapting while running — and providing services — reflect the vision of *autonomic* computing [55]. Along with this requirement of flexibility, software must remain reliable and predictable, in particular when adapting business- or safety-critical applications.

This requirements gave birth to an important branch of research interested in providing solutions like design patterns, language abstractions, and frameworks for the reliable development of systems that are able to adapt, preserving some desirable properties of correctness and coherence after the adaptation.

In the next sections we present some of the renown technologies that enable runtime software update and to structure adaptation, focussing on the constructs introduced by these technologies to:

- harness the inherent complexity of updating different parts of a software system, from a software engineering viewpoint;
- guarantee coherence between all the updated components and the correctness of the system after any update.

3.1 Linguistic Constructs for Adaptation

The practice of adapting software at runtime spans various levels of abstraction, ranging over parameters, methods, aspects, components, applications, architectures, systems, and data-centres [56].

In [57] the authors clarify that there exist two main approaches to software adaptation:

> This research has been addressing both the theoretical foundations of adaptation and, more pragmatically, how adaptation techniques can be applied to solve the problems at hand. [...] To achieve these goals, solutions to engineer self-adaptive behaviours are often sought at the software architecture level, including middleware and component-based design. Accordingly, architectural approaches to dynamic adaptation have been extensively studied by researchers [58, 59, 60]. As a complementary approach, researchers also adopted specialized programming paradigms to implement adaptive systems, such as metaprogramming and Aspect-Oriented Programming.

Essentially, the first approach is an architectural one and structures adaptation at the level of software components. The other approach is at language-level, which aims at providing language constructs for structuring which parts of a programs shall change at runtime. In their analysis, the authors of [57] remark that:

> Supporting behavioural variations at the language level is appealing with respect to dealing with adaptation at a higher architectural level. First, language-level approaches provide means to adapt at a very fine-grain detail. Second, many language concepts (e.g., polymorphism and late binding) are well known to programmers and can be easily specialized to support adaptation [61].

Hence, language-level solutions provide a finer control over the adaptation actions wrt architectural ones. In particular, constructs for adaptation embedded in programming languages provide a fine-grained granularity on adaptation whilst architectures for software adaptation usually rely on a component-based design that allows adaptation only at a coarser level.

Moreover, adaptation at language level brings several benefits:

- research on constructs for adaptation at language level tends to elicit general concepts that can be integrated, mixed, and evolved within new languages;
- languages take advantage of various techniques like syntactic and type checkers, compilers, etc. that enforce correctness of software;
- adaptation at architectural level introduces strong dependencies on the frameworks that enable adaptation.

Focussing on the practice of programming adaptive systems, we highlight two aspects that a language has to address for programming adaptation.

The first is an *enabling* aspect, i.e., the possibility of a language to control and change its own programming at runtime. The second is a *structuring* aspect, i.e., the abstractions offered by the language to program updates in a structured way.

In the following sections we briefly overview some of the main linguistic constructs that enable software updates and provide structure to their application.

3.1.1 Reflection and Metaprogramming

In [62], Pattie defines computation reflection as:

> [...] the behaviour exhibited by a reflective system, where a reflective system is a computational system which is about itself in a causally connected way.

In the context of the definition, a software system is said to be *causally connected* if there is a bond between its internal structures and the domain they represent. In particular, a reflective system is a computational system endowed with structures that represent aspects of itself. Having access to its self-representation, the system can respond to queries about its structural state and change it. Since a self-representing system is causally connected to the domain of the aspects it represents, *i*) the system has always an accurate representation of itself and *ii*) its computational status is always compliant with its representation, i.e., it can modify itself by means of its own computation.

Metaprogramming is the practice of using *computational reflection*, i.e., the use of a reflective interface on the program to change its behaviour. As exposed above, meta (domain) and base levels are causally connected, therefore changes at meta-level “reflect” in behavioural variations at base level.

Reflection is the most general and powerful mechanism to obtain runtime behavioural variation in programs. Examples of its usage span from higher-order languages like Lisp [63], which uses reflection to handle its own programming structures as *first-class* data types, to language APIs, which enable for runtime structure definitions (e.g., classes).

In general, the main two features of any metaprogramming facility are *inspection* and *modification*. Inspection supports queries on the state of execution of a program, whilst modification supports the change of the behaviour of a program, e.g., intercepting calls, modifying dispatching policies, augmenting data with new structures, and injecting new functions or modifying the body of existing ones. [64, 65, 66] are some established examples of usage of metaprogramming. In [64] Ledoux explains how it is possible to modify dynamically the behaviour of objects through dynamic classes. In [65] Dowling et al. employ reflection to enable a program to change the binding between its methods and their implementations. In [66] Xu et al. propose a method that exploits call interception and code injection for implementing fault tolerance.

All cited works prove that metaprogramming is the staple linguistic element that enables developers to pursue any kind of runtime adaptation on programs. However, due to its unstructured nature, the direct use of metaprogramming tends to generate code that is difficult to read and debug because programmers have to figure out which variations affected the behaviour of the program at runtime. More in general, metaprogramming does not guarantee any property on the state of the system after adaptation.

This led to the creation of a new set of language constructs *i*) to precisely define the boundaries of the effects of metaprogramming and to *ii*) to clearly specify the conditions that fire the application of each variation.

3.1.2 Aspect- and Context-Oriented Programming

Aspect-Oriented programming [67] (AOP) is one of the most renowned paradigms for managing runtime software update. The main reason behind the development of the aspect-oriented paradigm is to be able to separate the essential functionalities of a program from orthogonal modules like logging, persistence, synchronisation, and fault handling. The practice of uncoupling orthogonal modules from the main functionalities of a software is called *separation of concerns* and it is achieved by two specific constructs of AOP, namely *point-cut*s and *join-point*s.

A point-cut corresponds to a set of join-points. When the execution of the main code of a program reaches one of the join-points of a point-cut, the control is transferred from the main code to the code that implements a separate concern of that point-cut. The code corresponding to a join-point of a point-cut is called *advice*. The *aspect weaver* is the metaprogramming utility that "weaves" (merges) together the main code of an application, its point-cuts, and a set of advices to generate the final implementation of an aspect-oriented application.

Aspect-oriented programming has been successfully employed in many works on dynamic adaptation. Some examples are [68], [69], whilst [70] employed it to adapt distributed systems.

Although AOP provides a powerful yet easy abstraction to manage separation of concerns, its scope of application remains that of single programs and lacks mechanisms that ensure coherence among the adapted parts of a distributed system. Moreover, it is possible that various conditions coexist in the environment of execution and that several variations apply on the same system. Hence, beside making sure that adaptation is coherent over the system, it should be possible to express *conflicts* between variations. Adding such conflicting conditions in plain AOP quickly clutters the code, tangles the present point-cuts, and makes the insertion of new ones harder.

This motivations paved the way for the development of an evolution of the aspect-oriented paradigm for context-aware software: Context-Oriented programming [71] (COP).

The abstractions introduced by COP regard the modularisation of software updates to support the coherent activation of behavioural variations, based on *context*s [71], that Hirschfeld et al. define as:

> Any information which is computationally accessible may form part of the context upon which behavioural variations depend.

Given its recent introduction, one major direction of research on COP is involved in defining the essential features a context-oriented language should support. The context-oriented paradigm relies on *i*) proper abstractions for representing variations and *ii*) explicit mechanisms for triggering variations. Usually, COP extensions for mainstream languages [72], like ContextJ [71], provide abstractions for representing variations as first-class structures, which can be referenced and returned. The language provides ad-hoc mechanisms to dynamically activate and deactivate variations. When a variation is enabled, the set of behaviours it models activate and modify the runtime behaviour of the application. Vice versa, if the variation is deactivated the control returns to the main code.

Variations can also be activated simultaneously. In that case, the final behaviour of the application derives from the combination of the active variations. Since the applicability of variations is specified wrt both environmental conditions and the state of other variations, context-oriented languages ensure two properties: *i*) all the advices related to a variation are coherently active or inactive *ii*) conflicting advices cannot occur.

ContextErlang [73] is an interesting example of context-orientation brought in the (possibly) distributed world, by enabling the coherent application of variations on a per-thread basis. Variations can be both expressed internally and be included at runtime from an external source via message passing.

Although AOP and COP give a sensible contribution to ease the programming of runtime software updates, their scope remains that of single programs. Indeed, to the best of our knowledge, there are no formal results on the employment of such programming abstractions to preserve the coherency of updated distributed programs nor on the correctness of the programs after the application of updates.

3.1.3 Adaptation in Process-Aware Information Systems

In § 3.1, we underlined that in this work we do not consider architectural-level solutions wrt to language-level ones because the latter give a finer control over the adaptation actions.

Nonetheless, architectural-level approaches like that of *business process management* [74], and in particular that of Process-Aware Information Systems [75] (PAIS), investigated in depth how flexibility [76] — to support dynamic process adaptations — impacts on business processes and their realisation.

PAISs provide a high-level perspective over the whole composition of processes and, since the implementation is detached from the logic of composition, it is possible to plan for various degrees of flexibility by selecting which and when parts of the composition can change.

The community of BPM and PAIS did a formidable work in breaking down the nuances of such flexible systems into their attributes. In the context of this work, it is interesting to summarise such attributes and patterns, using them to clarify the possibilities offered by our language-level solution presented in Chapter 4.

In particular, we focus on the notion of *loosely specified processes* as given in [76] and defined as: *process models that are not fully pre-specified and that keep some parts unspecified at built-time by deferring decisions to runtime*. We report below the same breakdown of the attributes of such processes, drawn from [76]:

freedom given to the user to change the process. The lowest degree of freedom is *none*, which entails fixed processes that cannot change at runtime. Above it, we have the runtime *selection* of parts from a set of predefined alternatives. The highest level is the *modelling or composition* of parts of fragments of processes out of a set of activity templates;

planning approach used to take decisions when changing some parts. From most constrained to loosest: the *plan-driven* approach defines a detailed model of change up-front, the *iterative and continuous* approach relies on a coarse plan which is refined at runtime, and the *ad-hoc modelling* has no plan and resorts to the interaction with some user-manned or automatic system;

scope of change, i.e., which "regions" of the process model can be changed. The scope ranges from *predefined regions* to the *entire process*;

process perspective define which properties of a process can change. Changes can regard: the *behaviour* of a process, the *actors* executing a particular activity, the *data* used in the computation, the *functions* available in the process and their concrete implementations (called *operational decisions*), and finally the concrete *time* of start or end of activities;

making and support of decisions to concretise loosely specified processes at runtime. Decision systems are classified by the policy on which they base their decisions. These can be *goal-based*, which take into account the overall goals of the process, *rule-based*, which rely on a set of rules, *experience-based* which suggest solutions based on past experiences made in similar contexts (e.g., execution logs), *user decision* where decision making is left to the human expert;

degree of automation given to the system to compose instances of processes.

The division goes from full *automation*, *system-supported* interaction with the user (e.g., by providing context-specific templates which can be adopted), and *manual*, without system support.

Focussing in particular on the first two attributes of flexibility, namely the degree of *freedom* and the *planning approach*, the authors of [76] elicit some patterns of decision deferral, listed below in order of degree of flexibility:

Pattern	Description	Degree of freedom	Planning approach
Fully pre-specified processes	Processes that offer little flexibility. The model is fully pre-specified. They use a strict plan-driven approach separating modelling and execution entirely.	none	plan-driven
Late selection	Processes that allow placeholder activities (whose *scope* regards predefined regions). The placeholder activities are refined during runtime with concrete implementations from the available set of alternatives, which must be specified at build-time following a plan-driven approach.	selection	plan-driven
Late modelling and composition	Processes that allow also the automatic composition (not only selection) of activities. The *scope* can encompass regions or the whole process. The runtime refinement is still driven by a pre-specified plan.	modelling and composition	plan-driven
Iterative refinement	Processes that allow users to iteratively compose the content of a placeholder activity by selecting any process fragment from a given pool. The runtime refinement is not bound to a pre-specified plan. The *scope* spans the whole process or regions of it.	modelling and composition	iterative
Ad-hoc composition	Processes are composed in an ad-hoc manner, without any planning, with the *scope* spanning the whole process or regions of it.	modelling and composition	ad-hoc

Table 3.1: Decision deferral patterns.

As stated at the beginning of this section, in § 7.1.1 we use the fine-grained definitions of flexibility reported above to clarify the possibilities offered by our language-level choreography-based solution presented in Chapter 4.

3.2 Safe Update of Distributed Systems

Updating a single (non-distributed) program and making sure that its behaviour remains correct and coherent is a difficult task. However, also distributed systems and, by extension, the programs composing them, need to be updated and adapted. Guaranteeing that a distributed system remains correct and coherent after a step of adaptation is even more challenging.

Some interesting works on dynamic adaptation of distributed systems regard application of Aspect-Oriented programming to Service-Oriented Applications (introduced in § 2.2). For instance, in [77] Charfi et al. proved that adaptive service composition can be effectively addressed combining the constructs of AOP in BPEL [36]. However, the current technology of development is not well suited for writing and verifying adaptive distributed systems. This is due to the lack of high-level structuring abstractions needed for handling complex communication behaviours and context-aware adaptation.

Recently, several works introduced interesting abstractions for providing high-level specifications of the expected behaviour of a distributed systems. In particular we cite session types [11, 78, 43], choreography languages [12, 13], behavioural contracts [40], and ad-hoc scripting languages [79].

These concepts usually consider static techniques, which alone are not sufficient to model dynamically adaptable software. Indeed, the assumption on either the ability to type-check the component source code, or the availability of its complete behavioural interface, may not be realistic in presence of adaptation.

On the practice of adapting a distributed system, [80, 81, 61, 82, 83] are some interesting proposals of middlewares and architectures to enable runtime adaptation (an interesting survey can be found in [84]). Other works applied concepts of adaptive languages in the context of distributed systems like [85] with traits, [86] with role-based modelling, and [87] with aspect-oriented programming.

All the mentioned approaches provide tools for programming distributed adaptive systems, although they do not provide any guarantee on the expected behaviour of the system after the execution of some steps of adaptation. In the case of dynamically adaptive distributed programs, techniques like static analysis are impracticable due to the impossibility to know in advance the structure of the adapted system. For this reason, most of the approaches in the literature offer no guarantee on the behaviour of the adaptive system after adaptation.

In the next Chapter we report our choreographic solution to the problem of the safe, runtime (i.e., dynamic) update of distributed system.

CHAPTER 4

Dynamic Choreographies

Intelligence is the ability to adapt to change.

Stephen W. Hawking

4.1 Introduction

As illustrated in Chapter 2, programming safe distributed systems is a difficult activity, which can be made even more difficult if those systems shall update their behaviour at runtime. In Chapter 3 we presented the difficulties of such task within a single process and explained why updating a distributed system at runtime in a correct and coherent way is even more challenging. However, enabling distributed software to safely update at runtime has countless uses, spanning from dealing with emergencies, to coping with unexpected requirements or to improving and specialising applications to user preferences.

Here, we argue that choreographies (presented in § 2.4) are a suitable abstraction to express the dynamic update of distributed systems and to guarantee coherency and correctness of the system after each step of update.

In our treatment, we consider applications whose code can change at runtime by integrating other code provided by some external party. We propose a general mechanism for the structuring of updates that consists in delimiting inside the application blocks of code, called *scopes*, that may be dynamically replaced with new code, called an *update*. Remarkably, scopes and updates are independent and the details of the behaviour of updates do not need to be foreseen when writing the scopes. Indeed, updates may be written while the application is running.

Moreover, choreographies naturally lend themselves to the clear definition of the boundaries of the update and to enforce the coherent application of updates on all (and only) involved participants. To exemplify this concept, let us consider a simple interaction of a buyer that asks to a seller the price of some product.

```
1  priceReq : Buyer(prod) → Seller(order);
2  order_price@Seller = getPrice(order);
3  offer : Seller(order_price) → Buyer(prod_price)
```

The first line specifies that the Buyer sends along channel $priceReq$ the name of the desired product $prod$ to the Seller, which stores it in its local variable $order$. The Seller computes the price of the product calling the function $getPrice$ (Line 2) and, via the channel $offer$, it sends the price to the Buyer (Line 3), that stores it in a local variable $prod\_price$.

On the above example, let us suppose that the Seller would like to introduce new commercial offers in the future, e.g., to provide a discount on the computed prices. Since we want to be able to update the computation of prices, we enclose Lines 2–3 of the example within a scope, as shown below.

```
1  scope @Seller{
2     order_price@Seller = getPrice(order);
3     offer : Seller(order_price) → Buyer(prod_price)
4  }
```

In essence, a scope is a delimiter that defines which part of the application can be updated. Each scope identifies a participant as the *coordinator* of the update, responsible to decide whether to update and which (among possibly many) update to apply. In our example, the coordinator of the update is the Seller (Line 1).

Now that we have a scope, we can introduce runtime updates. Let us assume that the Seller issued a fidelity card to some returning customers and now the Seller wants to update the system to let Buyers insert their fidelity card to get a discount. The update in Figure 4.1 answers this business need.

```
1  cardReq : Seller(null) → Buyer(_);
2  card_id@Buyer = getInput();
3  cardRes : Buyer(card_id) → Seller(buyer_id);
4  if isValid(buyer_id)@Seller {
5     order_price@Seller = getPrice(order) * 0.9
6  } else {
7     order_price@Seller = getPrice(order)
8  };
9  offer : Seller(order_price) → Buyer(prod_price)
```

Figure 4.1: Fidelity Card Update.

At runtime, if the Seller (which is the coordinator of the update) selects the update in Figure 4.1, the code of the update replaces the one in the body of the scope. When this new code executes, the `Buyer` sends his/her $card\_id$ to the `Seller`. If the $card\_id$ is valid, the `Seller` issues a 10% discount on the price of the selected good, otherwise it reports the standard price to the `Buyer`.

We remark that separately writing the code for the `Buyer` and the `Seller` with some process-level language equipped with update primitives (cf. 3) would be extremely error prone. For instance, let us consider the case in which the `Buyer` is updated first and it starts the new interaction whilst the `Seller` cannot find the same update. The `Buyer` proceeds to send its card ID to the `Seller`. Contrarily, the `Seller` is executing the original interaction, it does not expect to receive the message carrying the card ID from the `Buyer` and the system is stuck (deadlocked): the `Buyer` waits for the `Seller` to receive its message carrying the card ID whilst the `Seller` waits for the `Buyer` to receive the message carrying the price of the product. In our setting, the available updates may change at any time, posing an additional challenge. Extra precautions are needed to ensure that all the participants agree on which code is used for a given update.

If both the original application and the updates are programmed using a choreographic language, these problems cannot arise. In fact, at the choreographic level, the update is applied atomically to all the involved participants.

However, providing the correct compilation from choreographic specifications to process-level distributed code is a challenging task. In particular, at the level of processes, the different participants have to coordinate their updates avoiding inconsistencies.

In this Chapter we propose our theory of dynamic choreographies. We present our choreographic language, called DIOC, for the programming of distributed applications supporting code update (§ 4.2). Next, we introduce a process-level language, called DPOC, based on standard send and receive primitives (§ 4.3) and we define our Endpoint Projection to compile DIOCs into DPOCs (§ 4.4). The EPP is proven correct (behaviour-preserving) (§ 4.7). We also prove that the property of correctness is preserved after any step of update.

In Chapter 5 we present one instantiation of our theory into a development framework for adaptive distributed applications called AIOCJ. In AIOCJ updates are embodied into adaptation rules, whose application is not purely non-deterministic (as in DIOCs), but depends on the state of the system and of its environment. AIOCJ comprises an Integrated Development Environment, a compiler from choreographies to executable programs, and a runtime environment to support their execution and update.

4.2 Dynamic Interaction-Oriented Choreographies

In this section we introduce the syntax of DIOCs, we illustrate the constructs of the DIOC language with a comprehensive example, and we finally present the semantics of DIOCs.

4.2.1 DIOC Syntax

DIOCs rely on a set of $Roles$, ranged over by $\mathtt{R}, \mathtt{S}, \ldots$, to identify the participants in the choreography. We call them roles to highlight that they have a specific duty in the choreography. Each role has its local state.

Roles exchange messages over public channels, also called *operations*, ranged over by o. We denote with $Expr$ the set of expressions, ranged over by e. We deliberately do not give a formal definition of expressions and of their typing, since our results do not depend on it. We only require that expressions include at least values, belonging to a set Val ranged over by v, and variables, belonging to a set Var ranged over by $x, y, \ldots$. We also assume a set of boolean expressions ranged over by b.

The syntax of *DIOC processes*, ranged over by $\mathcal{I}, \mathcal{I}', \ldots$, is defined as follows:

$$\begin{array}{rlll|lll}
\mathcal{I} ::= & o : \mathtt{R}(e) \rightarrow \mathtt{S}(x) & \text{(interaction)} & & \mathbf{0} & \text{(end)} \\
| & \mathcal{I};\mathcal{I}' & \text{(sequence)} & & \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\} & \text{(conditional)} \\
| & \mathcal{I}|\mathcal{I}' & \text{(parallel)} & & \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\} & \text{(while)} \\
| & x@\mathtt{R} = e & \text{(assignment)} & & \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\} & \text{(scope)} \\
| & \mathbf{1} & \text{(inaction)} & & &
\end{array}$$

Interaction $o : \mathtt{R}(e) \rightarrow \mathtt{S}(x)$ means that role $\mathtt{R}$ sends a message on operation o to role $\mathtt{S}$ (we require $\mathtt{R} \neq \mathtt{S}$). The sent value is obtained by evaluating expression e in the local state of $\mathtt{R}$ and it is then stored in the local variable x of $\mathtt{S}$. Processes $\mathcal{I};\mathcal{I}'$ and $\mathcal{I}|\mathcal{I}'$ denote sequential and parallel composition, respectively. Assignment $x@\mathtt{R} = e$ assigns the evaluation of expression e in the local state of $\mathtt{R}$ to its local variable x. The empty process $\mathbf{1}$ defines a DIOC that can only terminate. $\mathbf{0}$ represents a terminated DIOC. It is needed for the definition of the operational semantics and it is not intended to be used by the programmer. We call *initial* a DIOC process where $\mathbf{0}$ never occurs. The conditional $\mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}$ and the iteration $\mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}$ are guarded by the evaluation of the boolean expression b in the local state of $\mathtt{R}$. The construct $\mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}$ delimits a subterm $\mathcal{I}$ of the DIOC process that may be updated in the future. In $\mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}$, role $\mathtt{R}$ is the coordinator of the update: it decides whether to update or not and which update to apply.

A Running Example. We report in Figure 4.2 a running example of a DIOC process that extends the one presented in the Introduction: the example features a `Buyer` that orders a product from a `Seller`, and a `Bank` that supports the payment from the `Buyer` to the `Seller`.

At Lines 1–2 the `Buyer` initialises its local variables $price\_ok$ and $continue$. These variables control the while loop used by the `Buyer` to ask the price of some product to the `Seller`. The `Buyer` takes the name of the $prod$uct from the user with function $getInput$, which models interaction with the user (Line 4), and proceeds to send it to the `Seller` on operation $priceReq$ (Line 5). The `Seller` computes the price of the product calling the function $getPrice$ (Line 7) and, via operation $offer$, it sends the price to the `Buyer` (Line 8), that stores it in a local variable $prod\_price$. These last two operations are performed within a scope and therefore they can be updated at runtime to implement some new business policies (e.g., discounts). At Lines 10–12 the `Buyer` checks whether the user is willing to buy the product, and, if (s)he is not interested, whether (s)he wants to ask prices for other products. If the `Buyer` accepts the `Seller` offer, the `Seller` sends to the `Bank` the payment details (Line 16). Next, the `Buyer` authorises the payment via operation pay. We omit the details of the local execution of the payment at the `Bank`. Since the payment may be critical for security reasons, the related communications are enclosed in a scope (Lines 17–21), thus allowing the introduction of more refined procedures later on. After the scope successfully terminates, the application ends with the `Bank` acknowledging the payment to the `Seller` and the `Buyer` in parallel (Lines 23–25). If the payment is not successful, the failure is notified to the `Buyer` only. Note that at Lines 1–2 the annotation @`Buyer` means that the variables $price\_ok$ and $continue$ belong to the `Buyer`. Similarly, at Line 3, the annotation @`Buyer` means that the guard of the while loop is evaluated by the `Buyer`. The term @`Seller` at Line 6 is part of the scope construct and indicates the `Seller` as coordinator of the update.

4.2.2 Annotated DIOCs and their Semantics

In the remainder of the paper, we define our results on an annotated version of the DIOC syntax. Annotations are numerical indexes $\mathbf{i} \in \mathbb{N}$ assigned to DIOC constructs. We only require indexes to be distinct. Any annotation that satisfies this requirement provides the same result. Indeed, programmers do not need to annotate DIOCs: the annotation with indexes is mechanisable and can be performed by the language compiler[1]. Indexes are used both in the proof of our results and in the projection to avoid interferences between different constructs. From now on we consider only well-annotated DIOCs, defined as follows.

[1]In fact, the AIOCJ compiler implements such a feature.

```
price_ok@Buyer = false;
continue@Buyer = true;
while( !price_ok and continue )@Buyer{
   prod@Buyer = getInput();
   priceReq : Buyer(prod) → Seller(order);
   scope @Seller{
      order_price@Seller = getPrice(order);
      offer : Seller(order_price) → Buyer(prod_price)
   };
   price_ok@Buyer = getInput();
   if( !price_ok )@Buyer{
       continue@Buyer = getInput()
   }
};
if(price_ok)@Buyer{
   payReq : Seller( payDesc( order_price ) ) → Bank(desc);
   scope@Bank{
      payment_ok@Bank = true;
      pay : Buyer( payAuth( prod_price ) ) → Bank(auth);
      // code for the payment
   };
   if(payment_ok)@Bank{
      confirm : Bank(null) → Seller(_)
      |
      confirm : Bank(null) → Buyer(_)
   }else{
      abort : Bank(null) → Buyer(_)
   }
}
```

Figure 4.2: DIOC process for Purchase Scenario.

Definition 1 (Well-annotated DIOC)**.** Annotated DIOC processes *are obtained by indexing every interaction, assignment, conditional, while loop, and scope in a DIOC process with a positive natural number* $\mathbf{i} \in \mathbb{N}$, *resulting in the following grammar:*

$$\begin{array}{rl} \mathcal{I} ::= & \mathbf{i}\colon o : \mathtt{R}(e) \rightarrow \mathtt{S}(x) \;\mid\; \mathcal{I};\mathcal{I}' \;\mid\; \mathcal{I}|\mathcal{I}' \;\mid\; \mathbf{i}\colon x@\mathtt{R} = e \;\mid\; \mathbf{1} \;\mid\; \mathbf{0} \\ & \mid\; \mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\} \;\mid\; \mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\} \;\mid\; \mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\} \end{array}$$

A DIOC process is well annotated *if all its indexes are distinct.*

DIOC processes do not execute in isolation: they are equipped with a *global state* Σ and a set of available updates $\mathbf{I}$, i.e., a set of DIOCs that may replace scopes. Set $\mathbf{I}$ may change at runtime. A global state Σ is a map that defines the value v of each variable x in a given role R, namely $\Sigma : \mathit{Roles} \times \mathit{Var} \rightarrow \mathit{Val}$. The local state of role R is denoted as $\Sigma_{\mathtt{R}} : \mathit{Var} \rightarrow \mathit{Val}$ and it verifies that $\forall x \in \mathit{Var} : \Sigma(\mathtt{R}, x) = \Sigma_{\mathtt{R}}(x)$. Expressions are always evaluated by a given role R: we denote the evaluation of expression e in local state $\Sigma_{\mathtt{R}}$ as $[\![e]\!]_{\Sigma_{\mathtt{R}}}$. We assume that $[\![e]\!]_{\Sigma_{\mathtt{R}}}$ is always defined (e.g., an error value is given as a result if evaluation is not possible) and that for each boolean expression b, $[\![b]\!]_{\Sigma_{\mathtt{R}}}$ is either `true` or `false`.

Definition 2 (DIOC systems)**.** *A DIOC system is a triple* $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ *denoting a DIOC process* $\mathcal{I}$ *equipped with a global state* Σ *and a set of updates* $\mathbf{I}$.

Definition 3 (DIOC systems semantics)**.** *The semantics of DIOC systems is defined as the smallest labelled transition system (LTS) closed under the rules in Figure 4.3, where symmetric rules for parallel composition have been omitted.*

The rules in Figure 4.3 describe the behaviour of a DIOC system by induction on the structure of its DIOC process. We use μ to range over labels. The possible values for μ are described below.

$$\begin{array}{rlll|ll} \mu ::= & o : \mathtt{R}(v) \rightarrow \mathtt{S}(x) & \text{(interaction)} & & \tau & \text{(silent)} \\ \mid & \mathcal{I} & \text{(update)} & & \texttt{no-up} & \text{(no update)} \\ \mid & \mathbf{I} & \text{(change updates)} & & \surd & \text{(termination)} \end{array}$$

Rule $\lfloor^{\text{DIOC}}|_{\text{INTERACTION}}\rceil$ executes a communication from R to S on operation o, where R sends to S the value v of an expression e. The communication reduces to an assignment that inherits the index $\mathbf{i}$ of the interaction. The assignment stores value v in variable x of role S. Rule $\lfloor^{\text{DIOC}}|_{\text{ASSIGN}}\rceil$ evaluates the expression e in the local state $\Sigma_{\mathtt{R}}$ and stores the resulting value v in the local variable x in role R ($[v/x, \mathtt{R}]$ represents the substitution). Rule $\lfloor^{\text{DIOC}}|_{\text{SEQUENCE}}\rceil$ executes a step in the first process of a sequential composition, while rule $\lfloor^{\text{DIOC}}|_{\text{SEQ-END}}\rceil$ acknowledges the termination of the first process, starting the second one. Rule $\lfloor^{\text{DIOC}}|_{\text{PARALLEL}}\rceil$

$$\frac{[\![e]\!]_{\Sigma_{\mathtt{R}}} = v}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon o : \mathtt{R}(e) \to \mathtt{S}(x)\rangle \xrightarrow{o:\mathtt{R}(v)\to\mathtt{S}(x)} \langle \Sigma, \mathbf{I}, \mathbf{i}\colon x@\mathtt{S} = v\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Interaction}}\rceil$$

$$\frac{[\![e]\!]_{\Sigma_{\mathtt{R}}} = v}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon x@R = e\rangle \xrightarrow{\tau} \langle \Sigma[v/x, \mathtt{R}], \mathbf{I}, \mathbf{1}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Assign}}\rceil$$

$$\frac{\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mu} \langle \Sigma, \mathbf{I}', \mathcal{I}'\rangle \quad \mu \neq \surd}{\langle \Sigma, \mathbf{I}, \mathcal{I}; \mathcal{J}\rangle \xrightarrow{\mu} \langle \Sigma, \mathbf{I}', \mathcal{I}'; \mathcal{J}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Sequence}}\rceil$$

$$\frac{\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\surd} \langle \Sigma, \mathbf{I}, \mathcal{I}'\rangle \quad \langle \Sigma, \mathbf{I}, \mathcal{J}\rangle \xrightarrow{\mu} \langle \Sigma, \mathbf{I}, \mathcal{J}'\rangle}{\langle \Sigma, \mathbf{I}, \mathcal{I}; \mathcal{J}\rangle \xrightarrow{\mu} \langle \Sigma, \mathbf{I}, \mathcal{J}'\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Seq-end}}\rceil$$

$$\frac{\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mu} \langle \Sigma, \mathbf{I}', \mathcal{I}'\rangle \quad \mu \neq \surd}{\langle \Sigma, \mathbf{I}, \mathcal{I} \parallel \mathcal{J}\rangle \xrightarrow{\mu} \langle \Sigma, \mathbf{I}', \mathcal{I}' \parallel \mathcal{J}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Parallel}}\rceil$$

$$\frac{\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\surd} \langle \Sigma, \mathbf{I}, \mathcal{I}'\rangle \quad \langle \Sigma, \mathbf{I}, \mathcal{J}\rangle \xrightarrow{\surd} \langle \Sigma, \mathbf{I}, \mathcal{J}'\rangle}{\langle \Sigma, \mathbf{I}, \mathcal{I} \parallel \mathcal{J}\rangle \xrightarrow{\surd} \langle \Sigma, \mathbf{I}, \mathcal{I}' \parallel \mathcal{J}'\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Par-end}}\rceil$$

$$\frac{[\![b]\!]_{\Sigma_{\mathtt{R}}} = \mathtt{true}}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}\rangle \xrightarrow{\tau} \langle \Sigma, \mathbf{I}, \mathcal{I}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{If-then}}\rceil$$

$$\frac{[\![b]\!]_{\Sigma_{\mathtt{R}}} = \mathtt{false}}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}\rangle \xrightarrow{\tau} \langle \Sigma, \mathbf{I}, \mathcal{I}'\rangle} \lfloor^{\text{DIOC}}|_{\textsc{If-else}}\rceil$$

$$\frac{[\![b]\!]_{\Sigma_{\mathtt{R}}} = \mathtt{true}}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}\rangle \xrightarrow{\tau} \langle \Sigma, \mathbf{I}, \mathcal{I}; \mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{While-unfold}}\rceil$$

$$\frac{[\![b]\!]_{\Sigma_{\mathtt{R}}} = \mathtt{false}}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}\rangle \xrightarrow{\tau} \langle \Sigma, \mathbf{I}, \mathbf{1}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{While-exit}}\rceil$$

$$\frac{\mathsf{roles}(\mathcal{I}') \subseteq \mathsf{roles}(\mathcal{I}) \quad \mathcal{I}' \in \mathbf{I} \quad \mathsf{connected}(\mathcal{I}') \quad \mathsf{freshIndexes}(\mathcal{I}')}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon \mathtt{scope}\ @r\ \{\mathcal{I}\}\rangle \xrightarrow{\mathcal{I}'} \langle \Sigma, \mathbf{I}, \mathcal{I}'\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Up}}\rceil$$

$$\frac{}{\langle \Sigma, \mathbf{I}, \mathbf{i}\colon \mathtt{scope}\ @r\ \{\mathcal{I}\}\rangle \xrightarrow{\mathsf{no\text{-}up}} \langle \Sigma, \mathbf{I}, \mathcal{I}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{NoUp}}\rceil$$

$$\frac{}{\langle \Sigma, \mathbf{I}, \mathbf{1}\rangle \xrightarrow{\surd} \langle \Sigma, \mathbf{I}, \mathbf{0}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{End}}\rceil \qquad \frac{}{\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mathbf{I}'} \langle \Sigma, \mathbf{I}', \mathcal{I}\rangle} \lfloor^{\text{DIOC}}|_{\textsc{Change-Updates}}\rceil$$

Figure 4.3: Annotated DIOC system semantics.

$$\begin{aligned}
&\mathsf{roles}(\mathbf{i}\colon o : \mathtt{R}(e) \to \mathtt{S}(x)) = \{\mathtt{R}, \mathtt{S}\}\\
&\mathsf{roles}(\mathbf{1}) = \mathsf{roles}(\mathbf{0}) = \emptyset\\
&\mathsf{roles}(\mathbf{i}\colon x@\mathtt{R} = e) = \{\mathtt{R}\}\\
&\mathsf{roles}(\mathcal{I};\mathcal{I}') = \mathsf{roles}(\mathcal{I}|\mathcal{I}') = \mathsf{roles}(\mathcal{I}) \cup \mathsf{roles}(\mathcal{I}')\\
&\mathsf{roles}(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) = \{\mathtt{R}\} \cup \mathsf{roles}(\mathcal{I}) \cup \mathsf{roles}(\mathcal{I}')\\
&\mathsf{roles}(\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}) = \mathsf{roles}(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}) = \{\mathtt{R}\} \cup \mathsf{roles}(\mathcal{I})
\end{aligned}$$

Figure 4.4: Auxiliary function roles.

allows a process in a parallel composition to compute, while rule $\lfloor^{\text{DIOC}}|_{\text{PAR-END}}\rceil$ synchronises the termination of two parallel processes. Rules $\lfloor^{\text{DIOC}}|_{\text{IF-THEN}}\rceil$ and $\lfloor^{\text{DIOC}}|_{\text{IF-ELSE}}\rceil$ evaluate the boolean guard of a conditional, selecting the "then" and the "else" branch, respectively. Rules $\lfloor^{\text{DIOC}}|_{\text{WHILE-UNFOLD}}\rceil$ and $\lfloor^{\text{DIOC}}|_{\text{WHILE-EXIT}}\rceil$ correspond respectively to the unfolding of a while loop when its condition is satisfied and to its termination otherwise. The rules $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ and $\lfloor^{\text{DIOC}}|_{\text{NOUP}}\rceil$ deal with updates: the former applies an update, while the latter allows the body of the scope to be executed without updating it. More precisely, Rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ models the application of the update $\mathcal{I}'$ to the scope $\mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}$ which, as a result, is replaced by the DIOC process $\mathcal{I}'$. In the conditions of the rule, we use the function roles and the predicates connected and freshIndexes. Function $\mathsf{roles}(\mathcal{I})$, defined in Figure 4.4, computes the roles of a DIOC process $\mathcal{I}$. The condition of the rule requires that the roles of the update are a subset of the roles of the body of the scope. Predicate $\mathsf{connected}(\mathcal{I}')$ holds if $\mathcal{I}'$ is connected. Connectedness is a well-formedness property of DIOCs and is detailed in § 4.6. Predicate $\mathsf{freshIndexes}(\mathcal{I}')$ holds if all indexes in $\mathcal{I}'$ are fresh with respect to all indexes already present in the target DIOC[2]. Rule $\lfloor^{\text{DIOC}}|_{\text{NOUP}}\rceil$, used when no update is applied, removes the scope boundaries and starts the execution of the body of the scope. Rule $\lfloor^{\text{DIOC}}|_{\text{END}}\rceil$ terminates the execution of an empty process. Rule $\lfloor^{\text{DIOC}}|_{\text{CHANGE-UPDATES}}\rceil$ allows the set **I** of available updates to change. This rule is always enabled and models the fact that the set of available updates is not controlled by the system, but by the external world: the set of updates can change at any time, the system cannot forbid or delay these changes, and the system is not notified when they happen.

Remark 1. *Whether to update a scope or not, and which update to apply if many are available, is completely non deterministic. We have adopted this view to maximise generality. However, for practical applications it is also possible to reduce*

[2]We do not give a formal definition of $\mathsf{freshIndexes}(\mathcal{I}')$ to keep the presentation simple. However, freshness of indexes can be formally ensured using restriction as in π-calculus [5].

the non-determinism using suitable policies to decide when and whether a given update applies. One of such policies is defined in AIOCJ (see § 5.1).

We can finally provide the definition of *DIOC traces* and *weak DIOC traces*, which we use to express our results of behavioural correspondence between DIOCs and DPOCs. Intuitively, in DIOC traces all the performed actions are observed, whilst in weak DIOC traces silent actions τ are not visible.

Definition 4 (DIOC traces). *A* (strong) trace *of a DIOC system* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{I}_1 \rangle$ *is a sequence (finite or infinite) of labels* $\mu_1, \mu_2, \ldots$ *such that there is a sequence of DIOC system transitions* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{I}_1 \rangle \xrightarrow{\mu_1} \langle \Sigma_2, \mathbf{I}_2, \mathcal{I}_2 \rangle \xrightarrow{\mu_2} \ldots$.

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

4.3 Dynamic Process-Oriented Choreographies

In this section we define the syntax and semantics of DPOCs, the target language of our projection from DIOCs. We remind that DIOCs are not directly executable since their basic primitives describe distributed interactions. On the contrary, mainstream languages like Java and C, used for implementation, describe distributed computations using local behaviours and communication/synchronisation primitives, such as message send and message receive. In order to describe implementations corresponding to DIOCs we introduce the DPOC language, a core language based on this kind of primitives, but tailored to program updatable systems. Indeed, differently from DIOC constructs, DPOC constructs are locally implementable in any mainstream language. In AIOCJ (see § 5.1) we implement the DPOC constructs in the Jolie [88] language.

4.3.1 DPOC syntax

DPOCs include *processes*, ranged over by $P, P', \ldots$, describing the behaviour of participants. $(P, \Gamma)_{\mathtt{R}}$ denotes a *DPOC role* named $\mathtt{R}$, executing process P in a local state Γ. *Networks*, ranged over by $\mathcal{N}$, $\mathcal{N}'$, $\ldots$, are parallel compositions of DPOC roles with different names. DPOC systems, ranged over by $\mathcal{S}$, are DPOC networks equipped with a set of updates $\mathbf{I}$, namely pairs $\langle \mathbf{I}, \mathcal{N} \rangle$.

DPOCs, like DIOCs, feature operations o. Here we call them *public operations* to mark the difference with respect to *private operations*, also called auxiliary operations, ranged over by o^*. We use $o^?$ to range over both public and private operations. Differently from communications on public operations, communications on private operations have no direct correspondent at the DIOC level. Indeed, we introduce private operations in DPOCs to implement the synchronisation mechanisms needed to realise the global constructs of DIOCs (conditionals, while loops, and scopes) at DPOC level.

Like DIOC constructs, also DPOC constructs are annotated using indexes. However, in DPOCs we use two kinds of indexes: *normal indexes* $\mathbf{i} \in \mathbb{N}$ and *auxiliary indexes* of the forms $\mathbf{i}_{\texttt{T}}$, $\mathbf{i}_{\texttt{F}}$, $\mathbf{i}_{\texttt{?}}$, and $\mathbf{i}_{\texttt{C}}$ where $\mathbf{i} \in \mathbb{N}$. Auxiliary indexes are introduced by the projection, described in § 4.4, and are described in detail there. We range over DPOC indexes with ι.

In DPOCs, normal indexes are also used to prefix the operations of sends and receives. Thus, a send and a receive can interact only if they are on the same operation and they are prefixed by the same normal index. This is needed to avoid interferences between different communications, in particular when one of them comes from an update. We will describe in greater detail this issue later on.

The syntax of DPOCs is the following.

$$\begin{array}{llll}
P ::= \iota\colon \mathbf{i}.o^? : x\ \texttt{from}\ \texttt{R} & \text{(receive)} & \mid \mathbf{1} & \text{(inaction)} \\
\quad \mid \iota\colon \mathbf{i}.o^? : e\ \texttt{to}\ \texttt{R} & \text{(send)} & \mid \mathbf{0} & \text{(end)} \\
\quad \mid \mathbf{i}\colon \mathbf{i}.o^* : X\ \texttt{to}\ \texttt{R} & \text{(send-update)} & \mid \mathbf{i}\colon \texttt{if}\ b\ \{P\}\ \texttt{else}\ \{P'\} & \text{(conditional)} \\
\quad \mid P; P' & \text{(sequence)} & \mid \mathbf{i}\colon \texttt{while}\ b\ \{P\} & \text{(while)} \\
\quad \mid P \mid P' & \text{(parallel)} & \mid \mathbf{i}\colon \texttt{scope}\ @\texttt{R}\ \{P\}\ \texttt{roles}\ \{S\} & \text{(scope-coord)} \\
\quad \mid \iota\colon x = e & \text{(assignment)} & \mid \mathbf{i}\colon \texttt{scope}\ @\texttt{R}\ \{P\} & \text{(scope)}
\end{array}$$

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

DPOC processes include receive action $\iota\colon \mathbf{i}.o^? : x\ \texttt{from}\ \texttt{R}$ on a specific operation $\mathbf{i}.o^?$ (either public or private) of a message from role R to be stored in variable x, send action $\iota\colon \mathbf{i}.o^? : e\ \texttt{to}\ \texttt{R}$ of the value of an expression e to be sent to role R, and higher-order send action $\mathbf{i}\colon \mathbf{i}.o^* : X\ \texttt{to}\ \texttt{R}$ of the higher-order argument X to be sent to role R. Here X may be either a DPOC process P, which is the new code for a scope in R, or a token no, notifying that no update is needed. $P; P'$ and $P|P'$ denote the sequential and parallel composition of P and P', respectively. Processes also feature assignment $\iota\colon x = e$ of the value of expression e to variable x, the process $\mathbf{1}$, that can only successfully terminate, and the terminated process $\mathbf{0}$. DPOC processes also include conditionals $\mathbf{i}\colon \texttt{if}\ b\ \{P\}\ \texttt{else}\ \{P'\}$ and loops $\mathbf{i}\colon \texttt{while}\ b\ \{P\}$. Finally, there are two constructs for scopes. Construct $\mathbf{i}\colon \texttt{scope}\ @\texttt{R}\ \{P\}\ \texttt{roles}\ \{S\}$ defines a scope with body P and set of participants S, and may occur only inside role R, which acts as coordinator of the update. The shorter version $\mathbf{i}\colon \texttt{scope}\ @\texttt{R}\ \{P\}$ is used instead when the role R is not the coordinator of the update. In fact, only the coordinator needs to know the set S of involved roles to be able to send to them their updates.

4.3.2 DPOC semantics

DPOC semantics is defined in two steps: we define the semantics of DPOC roles first, and then we define how roles interact giving rise to the semantics of DPOC systems.

Definition 5 (DPOC roles semantics)**.** *The semantics of DPOC roles is defined as the smallest LTS closed under the rules in Figure 4.5, where we report the rules dealing with computation, and Figure 4.6, in which we define the rules related to updates. Symmetric rules for parallel composition have been omitted.*

DPOC role semantics. We use δ to range over labels. Possible values for δ are as follows:

$$\begin{array}{llrlr}
\delta ::= & \mathbf{i}.o^?(x \leftarrow v)@\mathtt{S} : \mathtt{R} & \text{(receive)} \mid & \overline{\mathbf{i}.o^?}\langle v\rangle@\mathtt{S} : \mathtt{R} & \text{(send)} \\
\mid & \mathbf{i}.o^*(\leftarrow X)@\mathtt{S} : \mathtt{R} & \text{(receive-update)} \mid & \overline{\mathbf{i}.o^*}\langle X\rangle@\mathtt{S} : \mathtt{R} & \text{(send-update)} \\
\mid & \mathtt{no\text{-}up} & \text{(no-update)} \mid & \mathcal{I} & \text{(update)} \\
\mid & \surd & \text{(termination)} \mid & \tau & \text{(silent)}
\end{array}$$

The semantics is in the early style. Rule $\lfloor^{\text{DPOC}}|_{\text{Recv}}\rceil$ receives a value v from role $\mathtt{S}$ and assigns it to local variable x of $\mathtt{R}$. Similarly to Rule $\lfloor^{\text{DIOC}}|_{\text{Interact}}\rceil$ (see Figure 4.3), the reception reduces to an assignment that inherits the index $\mathbf{i}$ from the receive primitive.

Rules $\lfloor^{\text{DPOC}}|_{\text{Send}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{Send-Up}}\rceil$ execute send and higher-order send actions, respectively. Send actions evaluate expression e in the local state Γ. Rule $\lfloor^{\text{DPOC}}|_{\text{One}}\rceil$ terminates an empty process. Rule $\lfloor^{\text{DPOC}}|_{\text{Assign}}\rceil$ executes an assignment ($[v/x]$ represents the substitution of value v for variable x). Rule $\lfloor^{\text{DPOC}}|_{\text{Sequence}}\rceil$ executes a step in the first process of a sequential composition, while Rule $\lfloor^{\text{DPOC}}|_{\text{Seq-end}}\rceil$ acknowledges the termination of the first process, starting the second one. Rule $\lfloor^{\text{DPOC}}|_{\text{Parallel}}\rceil$ allows a process in a parallel composition to compute, while Rule $\lfloor^{\text{DPOC}}|_{\text{Par-end}}\rceil$ synchronises the termination of two of parallel processes. Rules $\lfloor^{\text{DPOC}}|_{\text{If-then}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{If-else}}\rceil$ select the "then" and the "else" branch in a conditional, respectively. Rules $\lfloor^{\text{DPOC}}|_{\text{While-unfold}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{While-exit}}\rceil$ model respectively the unfolding and the termination of a loop.

The rules reported in Figure 4.6 deal with code updates. Rules $\lfloor^{\text{DPOC}}|_{\text{Lead-Up}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{Lead-NoUp}}\rceil$ specify the behaviour of the coordinator of the update, respectively when an update is performed and when no update is performed. Rules $\lfloor^{\text{DPOC}}|_{\text{Up}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{NoUp}}\rceil$ are the corresponding rules for other roles. The rules exploit private operations sb^* and se^* to coordinate the beginning and the end of the update, respectively. Communications on sb^* also carry the new code if the

$$\frac{}{(\mathbf{1},\Gamma)_{\mathtt{R}} \xrightarrow{\surd} (\mathbf{0},\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{One}}\rceil \qquad \frac{[\![e]\!]_{\Gamma} = v}{(\iota\colon x = e,\Gamma)_{\mathtt{R}} \xrightarrow{\tau} (\mathbf{1},\Gamma[v/x])_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Assign}}\rceil$$

$$\frac{[\![e]\!]_{\Gamma} = v}{\big(\iota\colon \mathbf{i}.o^? : e \ \mathtt{to}\ \mathtt{S},\Gamma\big)_{\mathtt{R}} \xrightarrow{\overline{\mathbf{i}.o^?}\langle v\rangle@\mathtt{S}:\mathtt{R}} (\mathbf{1},\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Send}}\rceil$$

$$\frac{}{\big(\iota\colon \mathbf{i}.o^? : x \ \mathtt{from}\ \mathtt{S},\Gamma\big)_{\mathtt{R}} \xrightarrow{\mathbf{i}.o^?(x\leftarrow v)@\mathtt{S}:\mathtt{R}} (\iota\colon x = v,\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Recv}}\rceil$$

$$\frac{}{\big(\mathbf{i}\colon \mathbf{i}.o^? : X \ \mathtt{to}\ \mathtt{S},\Gamma\big)_{\mathtt{R}} \xrightarrow{\overline{\mathbf{i}.o^?}\langle X\rangle@\mathtt{S}:\mathtt{R}} (\mathbf{1},\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Send-Up}}\rceil$$

$$\frac{(P,\Gamma)_{\mathtt{R}} \xrightarrow{\delta} (P',\Gamma')_{\mathtt{R}} \quad \delta \neq \surd}{(P;Q,\Gamma)_{\mathtt{R}} \xrightarrow{\delta} (P';Q,\Gamma')_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Sequence}}\rceil$$

$$\frac{(P,\Gamma)_{\mathtt{R}} \xrightarrow{\surd} (P',\Gamma)_{\mathtt{R}} \quad (Q,\Gamma)_{\mathtt{R}} \xrightarrow{\delta} (Q',\Gamma')_{\mathtt{R}}}{(P;Q,\Gamma)_{\mathtt{R}} \xrightarrow{\delta} (Q',\Gamma')_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Seq-end}}\rceil$$

$$\frac{(P,\Gamma)_{\mathtt{R}} \xrightarrow{\delta} (P',\Gamma')_{\mathtt{R}} \quad \delta \neq \surd}{(P \mid Q,\Gamma)_{\mathtt{R}} \xrightarrow{\delta} (P' \mid Q,\Gamma')_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Parallel}}\rceil$$

$$\frac{(P,\Gamma)_{\mathtt{R}} \xrightarrow{\surd} (P',\Gamma)_{\mathtt{R}} \quad (Q,\Gamma)_{\mathtt{R}} \xrightarrow{\surd} (Q',\Gamma)_{\mathtt{R}}}{(P \mid Q,\Gamma)_{\mathtt{R}} \xrightarrow{\surd} (P' \mid Q',\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{Par-end}}\rceil$$

$$\frac{[\![b]\!]_{\Gamma} = \mathtt{true}}{(\mathbf{i}\colon \mathtt{if}\ b\ \{P\}\ \mathtt{else}\ \{P'\},\Gamma)_{\mathtt{R}} \xrightarrow{\tau} (P,\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{If-Then}}\rceil$$

$$\frac{[\![b]\!]_{\Gamma} = \mathtt{false}}{(\mathbf{i}\colon \mathtt{if}\ b\ \{P\}\ \mathtt{else}\ \{P'\},\Gamma)_{\mathtt{R}} \xrightarrow{\tau} (P',\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{If-Else}}\rceil$$

$$\frac{[\![b]\!]_{\Gamma} = \mathtt{true}}{(\mathbf{i}\colon \mathtt{while}\ b\ \{P\},\Gamma)_{\mathtt{R}} \xrightarrow{\tau} (P;\mathbf{i}\colon \mathtt{while}\ e\ \{P\},\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{While-unfold}}\rceil$$

$$\frac{[\![b]\!]_{\Gamma} = \mathtt{false}}{(\mathbf{i}\colon \mathtt{while}\ b\ \{P\},\Gamma)_{\mathtt{R}} \xrightarrow{\tau} (\mathbf{1},\Gamma)_{\mathtt{R}}} \lfloor^{\text{DPOC}}|_{\textsc{While-exit}}\rceil$$

Figure 4.5: DPOC role semantics. Computation rules. (Update rules in Figure 4.6)

$$\frac{\mathsf{roles}(\mathcal{I}) \subseteq S \quad \mathsf{freshIndexes}(\mathcal{I}) \quad \mathsf{connected}(\mathcal{I})}{(\mathbf{i}\colon \texttt{scope}\ @R\ \{P\}\ \texttt{roles}\ \{S\}, \Gamma)_{\texttt{R}} \xrightarrow{\mathcal{I}} \left(\begin{array}{l} \prod_{\texttt{R}_j \in S\setminus\{\texttt{R}\}} \mathbf{i}\colon \mathbf{i}.sb^*_{\mathbf{i}} : \pi(\mathcal{I},\texttt{R}_j)\ \texttt{to}\ \texttt{R}_j; \\ \pi(\mathcal{I},\texttt{R}); \\ \prod_{\texttt{R}_j\in S\setminus\{r\}} \mathbf{i}\colon \mathbf{i}.se^*_{\mathbf{i}} : \_\ \texttt{from}\ \texttt{R}_j,\ \Gamma \end{array}\right)_{\texttt{R}}} \ \lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$$

$$\frac{}{\begin{array}{l}(\mathbf{i}\colon \texttt{scope}\ @R\ \{P\}\ \texttt{roles}\ \{S\}, \Gamma)_{\texttt{R}} \xrightarrow{\mathsf{no\text{-}up}} \\ \left(\prod_{\texttt{R}_j \in S\setminus\{\texttt{R}\}} \mathbf{i}\colon \mathbf{i}.sb^*_{\mathbf{i}} : \mathsf{no}\ \texttt{to}\ \texttt{R}_j; P;\ \prod_{\texttt{R}_j \in S\setminus\{\texttt{R}\}} \mathbf{i}\colon \mathbf{i}.se^*_{\mathbf{i}} : \_\ \texttt{from}\ \texttt{R}_j, \Gamma \right)_{\texttt{R}}\end{array}} \ \lfloor^{\text{DPOC}}|_{\text{LEAD-NOUP}}\rceil$$

$$\frac{}{(\mathbf{i}\colon \texttt{scope}\ @\texttt{S}\ \{P\}, \Gamma)_{\texttt{R}} \xrightarrow{\mathbf{i}.sb^*_{\mathbf{i}}(\leftarrow P')@\texttt{S}:\texttt{R}} (P'; \mathbf{i}\colon \mathbf{i}.se^*_{\mathbf{i}} : \mathsf{ok}\ \texttt{to}\ \texttt{S}, \Gamma)_{\texttt{R}}} \ \lfloor^{\text{DPOC}}|_{\text{UP}}\rceil$$

$$\frac{}{(\mathbf{i}\colon \texttt{scope}\ @\texttt{S}\ \{P\}, \Gamma)_{\texttt{R}} \xrightarrow{\mathbf{i}.sb^*_{\mathbf{i}}(\leftarrow \mathsf{no})@\texttt{S}:\texttt{R}} (P; \mathbf{i}\colon \mathbf{i}.se^*_{\mathbf{i}} : \mathsf{ok}\ \texttt{to}\ \texttt{S}, \Gamma)_{\texttt{R}}} \ \lfloor^{\text{DPOC}}|_{\text{NOUP}}\rceil$$

Figure 4.6: DPOC role semantics. Update rules. (Computation rules in Figure 4.5)

update is performed, and a token no otherwise. Communications on se^* carry no relevant data: they are used for synchronisation purposes only.

Rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$ models the fact that the coordinator R of the update applies an update $\mathcal{I}$. The premises of Rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$ are similar to those of Rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ (see Figure 4.3). Function roles is used to check that the roles in $\mathcal{I}$ are included in the roles of the scope. Freshness of indexes is checked by predicate freshIndexes, and well formedness of $\mathcal{I}$ by predicate connected (formally defined later on, in Definition 10 in § 4.6).

It is important that the decision on whether to update or not is taken by the unique coordinator R for two reasons. First, R ensures that all involved roles agree on whether to update or not. Second, since the set of updates may change at any time, the choice of the update inside **I** needs to be atomic, and this is guaranteed using a unique coordinator. The coordinator R also generates the processes to be executed by the roles in S using the process-projection function π (detailed in § 4.4). More precisely, $\pi(\mathcal{I}, \texttt{R}_i)$ generates the code for role $\texttt{R}_i$. The processes $\pi(\mathcal{I}, \texttt{R}_i)$ are sent via auxiliary higher-order communications to the roles that have to execute them. These communications also notify the other roles that they can start executing the new code. Here, and in the remainder of the paper, we define $\prod_{\texttt{R}_i \in S} P_i$ as the parallel composition of DPOC processes P_i for $R_i \in S$.

After the communication of the updated code to the other participants, R starts its own updated code $\pi(\mathcal{I}, \mathtt{R})$. Finally, auxiliary communications $se^*_{\mathtt{i}}$ are used to synchronise the end of the execution of the update (here _ denotes a fresh variable to store the synchronisation message ok). Summarising, during scope execution auxiliary communications ensure that the update is performed in a coordinated way, i.e., the roles agree on when the scope starts and terminates and on whether the update is performed or not.

Rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-NOUP}}\rceil$ defines the behaviour of the coordinator R when no update is applied. In this case, R sends a token no to the other involved roles, notifying them that no update is applied and that they can start executing their code. End of scope synchronisation follows that of Rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$.

Rules $\lfloor^{\text{DPOC}}|_{\text{UP}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{NOUP}}\rceil$ define the behaviour of the other roles involved in the scope. The scope waits for a message from the coordinator. If the content of the message is no, the body of the scope is executed. Otherwise, the content of the message is a process P' which is executed instead of the body of the scope.

DPOC system semantics.

Definition 6 (DPOC systems semantics)**.** *The semantics of DPOC systems is defined as the smallest LTS closed under the rules in Figure 4.7. Symmetric rules for parallel composition have been omitted.*

We use η to range over DPOC systems labels. Possible values of η are as follows:

$$\begin{array}{lr} \eta ::= o^? : \mathtt{R}(v) \rightarrow \mathtt{S}(x) & \text{(interaction)} \\ \quad\;\; \mid o^* : \mathtt{R}(X) \rightarrow \mathtt{S}() & \text{(interaction-update)} \\ \quad\;\; \mid \delta & \text{(role label)} \end{array}$$

Rules $\lfloor^{\text{DPOC}}|_{\text{LIFT}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{LIFT-UP}}\rceil$ lift role transitions to the system level. Rule $\lfloor^{\text{DPOC}}|_{\text{LIFT-UP}}\rceil$ also checks that the update $\mathcal{I}$ belongs to the set of currently available updates **I**. Rule $\lfloor^{\text{DPOC}}|_{\text{SYNCH}}\rceil$ synchronises a send with the corresponding receive, producing an interaction. Rule $\lfloor^{\text{DPOC}}|_{\text{SYNCH-UP}}\rceil$ is similar, but it deals with higher-order interactions. Note that Rules $\lfloor^{\text{DPOC}}|_{\text{SYNCH}}\rceil$ and $\lfloor^{\text{DPOC}}|_{\text{SYNCH-UP}}\rceil$ remove the prefixes from DPOC operations in transition labels. The labels of these transitions store the information on the occurred communication: label $o^? : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x)$ denotes an interaction on operation $o^?$ from role $\mathtt{R}_1$ to role $\mathtt{R}_2$ where the value v is sent by $\mathtt{R}_1$ and then stored by $\mathtt{R}_2$ in variable x. Label $o^* : \mathtt{R}_1(X) \rightarrow \mathtt{R}_2()$ denotes a similar interaction, but concerning a higher-order

$$\frac{\mathcal{N} \xrightarrow{\delta} \mathcal{N}' \quad \delta \neq \mathcal{I}}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\delta} \langle \mathbf{I}, \mathcal{N}' \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Lift}} \right\rceil \qquad \frac{\mathcal{N} \xrightarrow{\mathcal{I}} \mathcal{N}' \quad \mathcal{I} \in \mathbf{I}}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\mathcal{I}} \langle \mathbf{I}, \mathcal{N}' \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Lift-Up}} \right\rceil$$

$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\overline{\mathbf{i}.o^?}\langle v \rangle @\mathtt{S}:\mathtt{R}} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{\mathbf{i}.o^?(x \leftarrow v)@\mathtt{R}:\mathtt{S}} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{o^?:\mathtt{R}(v) \to \mathtt{S}(x)} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Synch}} \right\rceil$$

$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\overline{\mathbf{i}.o^*}\langle X \rangle @\mathtt{S}:\mathtt{R}} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{\mathbf{i}.o^*(\leftarrow X)@\mathtt{R}:\mathtt{S}} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{o^*:\mathtt{R}(X) \to \mathtt{S}()} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Synch-Up}} \right\rceil$$

$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \eta \neq \surd}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}'' \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Ext-Parallel}} \right\rceil$$

$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\surd} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{\surd} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{\surd} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Ext-Par-End}} \right\rceil$$

$$\frac{}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\mathbf{I}'} \langle \mathbf{I}', \mathcal{N} \rangle} \left\lfloor^{\text{DPOC}} \middle|_{\textsc{Change-Updates}} \right\rceil$$

Figure 4.7: DPOC system semantics.

value X, which can be either the code used in the update or a token no if no update is performed. No receiver variable is specified, since the received value becomes part of the code of the receiving process. Rule $\lfloor^{\text{DPOC}}|_{\text{EXT-PAR}}\rceil$ allows a network inside a parallel composition to compute. Rule $\lfloor^{\text{DPOC}}|_{\text{EXT-PAR-END}}\rceil$ synchronises the termination of parallel networks. Finally, Rule $\lfloor^{\text{DPOC}}|_{\text{CHANGE-UPDATES}}\rceil$ allows the set of updates to change arbitrarily.

We now define *DPOC traces* and *weak DPOC traces*, which we later use, along with DIOC traces and weak DIOC traces, to define our result of correctness.

Definition 7 (DPOC traces). *A* (strong) trace *of a DPOC system* $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$ *is a sequence (finite or infinite) of labels* $\eta_1, \eta_2, \ldots$ *with*

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

such that there is a sequence of transitions $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle \xrightarrow{\eta_1} \langle \mathbf{I}_2, \mathcal{N}_2 \rangle \xrightarrow{\eta_2} \ldots$.
A weak trace *of a DPOC system* $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$ *is a sequence of labels* $\eta_1, \eta_2, \ldots$ *obtained by removing all the labels corresponding to private communications, i.e., of the form* $o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x)$ *or* $o^* : \mathtt{R}_1(X) \to \mathtt{R}_2()$, *and the silent labels* τ, *from a trace of* $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$.

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

Remark 2. *Contrarily to DIOCs, DPOCs can deadlock. For instance,*

$$(\mathbf{i}\text{: } \mathbf{i}.o : x \;\mathtt{from}\; \mathtt{R}', \Gamma)_{\mathtt{R}}$$

is a deadlocked DPOC network where processes are not terminated and the only enabled actions are changes of the set of updates (i.e., transitions with label $\mathbf{I}$*), which are not actual system activities, but are taken by the environment.*

4.4 Projection Function

We now introduce the projection function proj. Given a DIOC specification, proj returns a network of DPOC programs that, by interacting, enact the behaviour defined by the originating DIOC.

We write the projection of a DIOC $\mathcal{I}$ as $\mathsf{proj}(\mathcal{I}, \Sigma)$, where Σ is a global state. Informally, the projection of a DIOC is a parallel composition of terms, one for each role of the DIOC. The body of these roles is computed by the *process-projection* function π (defined below). Given a DIOC and a role name R, the process-projection returns the process corresponding to the local behaviour of role $\mathtt{R}$. Since the roles executing the process-projections are composed in parallel, the projection of a DIOC program results into the DPOC network of the projected roles.

To give the formal definition of projection, we first define $\|_{i\in I}\ \mathcal{N}_i$ as the parallel composition of networks $\mathcal{N}_i$ for $i \in I$.

Definition 8 (Projection)**.** *The projection of a DIOC process $\mathcal{I}$ with global state Σ is the DPOC network defined by:*

$$\mathsf{proj}(\mathcal{I}, \Sigma) = \|_{\mathtt{S}\in\mathsf{roles}(\mathcal{I})}\ (\pi(\mathcal{I}, \mathtt{S}), \Sigma_{\mathtt{S}})_{\mathtt{S}}$$

The process-projection function that derives DPOC processes from DIOC processes is defined as follows.

Definition 9 (Process-projection)**.** *Given an annotated DIOC process $\mathcal{I}$ and a role $\mathtt{R}$ the projected DPOC process $\pi(\mathcal{I}, \mathtt{R})$ is defined as in Figure 4.8.*

With little abuse of notation, we write $\mathsf{roles}(\mathcal{I}, \mathcal{I}')$ for $\mathsf{roles}(\mathcal{I}) \cup \mathsf{roles}(\mathcal{I}')$. We assume that variables $x_{\mathbf{i}}$ are never used in the DIOC to be projected and we use them for auxiliary synchronisations.

The projection is homomorphic for sequential and parallel composition, $\mathbf{1}$ and $\mathbf{0}$. The projection of an assignment is the assignment on the role performing it and $\mathbf{1}$ on other roles. The projection of an interaction is a send on the sender role, a receive on the receiver, and $\mathbf{1}$ on any other role. The projection of a scope is a scope on all its participants. On its coordinator it also features a clause that records the roles of the involved participants. On the roles not involved in the scope the projection is $\mathbf{1}$. Projections of conditional and while loop are a bit more complex, since they need to coordinate a distributed computation. To this end they exploit communications on private operations. In particular, $cnd^*_{\mathbf{i}}$ coordinates the branching of conditionals, carrying information on whether the "then" or the "else" branch needs to be taken. Similarly, $wb^*_{\mathbf{i}}$ coordinates the beginning of a while loop, carrying information on whether to loop or to exit. Finally, $we^*_{\mathbf{i}}$ coordinates the end of the body of the while loop. This closing operation carries no relevant information and it is just used for synchronisation purposes. In order to execute a conditional $\mathbf{i}$: $\mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}$, the coordinator $\mathtt{R}$ of the conditional locally evaluates the guard and tells the other roles which branch to choose using auxiliary communications on $cnd^*_{\mathbf{i}}$. Finally, all the roles involved in the conditional execute their code corresponding to the chosen branch. Execution of a loop $\mathbf{i}$: $\mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}$ is similar, with two differences. First, end of loop synchronisation on operations $we^*_{\mathbf{i}}$ is used to agree on when an iteration is terminated, and a new one can be started. Second, communication of whether to loop or to

$$\boxed{\pi(\mathbf{1},\mathtt{S})} = \mathbf{1} \qquad \boxed{\pi(\mathbf{0},\mathtt{S})} = \mathbf{0}$$

$$\boxed{\pi(\mathcal{I};\mathcal{I}',\mathtt{S})} = \pi(\mathcal{I},\mathtt{S});\pi(\mathcal{I}',\mathtt{S}) \qquad \boxed{\pi(\mathcal{I}|\mathcal{I}',\mathtt{S})} = \pi(\mathcal{I},\mathtt{S})|\pi(\mathcal{I}',\mathtt{S})$$

$$\boxed{\pi(\mathbf{i}\colon x@\mathtt{R} = e,\mathtt{R})} = \mathbf{i}\colon x = e$$

$$\boxed{\pi(\mathbf{i}\colon x@\mathtt{R} = e,\mathtt{S})\ \mathsf{and}\ \mathtt{S} \neq \mathtt{R}} = \mathbf{1}$$

$$\boxed{\pi(\mathbf{i}\colon o : \mathtt{R}_1(e) \to \mathtt{R}_2(x),\mathtt{R}_1)} = \mathbf{i}\colon \mathbf{i}.o : e\ \mathtt{to}\ \mathtt{R}_2$$

$$\boxed{\pi(\mathbf{i}\colon o : \mathtt{R}_1(e) \to \mathtt{R}_2(x),\mathtt{R}_2)} = \mathbf{i}\colon \mathbf{i}.o : x\ \mathtt{from}\ \mathtt{R}_1$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon o : \mathtt{R}_1(e) \to \mathtt{R}_2(x),\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \notin \{\mathtt{R}_1,\mathtt{R}_2\}\end{array}} = \mathbf{1}$$

$$\boxed{\pi(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\},\mathtt{R})} = \left\{\begin{array}{l} \mathbf{i}\colon \mathtt{if}\ b \left\{\begin{array}{l} \left(\prod\limits_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\setminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{T}}\colon \mathbf{i}.\mathit{cnd}^*_{\mathbf{i}} : \mathtt{true\ to\ R}'\right); \\ \pi(\mathcal{I},\mathtt{R}) \end{array}\right\} \\ \quad \mathtt{else} \left\{\begin{array}{l} \left(\prod\limits_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\setminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{F}}\colon \mathbf{i}.\mathit{cnd}^*_{\mathbf{i}} : \mathtt{false\ to\ R}'\right); \\ \pi(\mathcal{I}',\mathtt{R}) \end{array}\right\} \end{array}\right.$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\},\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \in \mathsf{roles}(\mathcal{I},\mathcal{I}') \setminus \{\mathtt{R}\}\end{array}} = \mathbf{i}_?\colon \mathbf{i}.\mathit{cnd}^*_{\mathbf{i}} : x_{\mathbf{i}}\ \mathtt{from\ R};\ \mathbf{i}\colon \mathtt{if}\ x_{\mathbf{i}}\ \{\pi(\mathcal{I},\mathtt{S})\}\ \mathtt{else}\ \{\pi(\mathcal{I}',\mathtt{S})\}$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\},\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \notin \mathsf{roles}(\mathcal{I},\mathcal{I}') \cup \{\mathtt{R}\}\end{array}} = \mathbf{1}$$

$$\boxed{\pi(\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\},\mathtt{R})} = \left\{\begin{array}{l} \mathbf{i}\colon \mathtt{while}\ b\ \Big\{ \\ \quad \left(\prod\limits_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\setminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{T}}\colon \mathbf{i}.\mathit{wb}^*_{\mathbf{i}} : \mathtt{true\ to\ R}'\right); \quad \pi(\mathcal{I},\mathtt{R}); \\ \quad \prod\limits_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\setminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{C}}\colon \mathbf{i}.\mathit{we}^*_{\mathbf{i}} : \_\ \mathtt{from\ R}' \\ \Big\}; \quad \prod\limits_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\setminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{F}}\colon \mathbf{i}.\mathit{wb}^*_{\mathbf{i}} : \mathtt{false\ to\ R}' \end{array}\right.$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\},\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \in \mathsf{roles}(\mathcal{I}) \setminus \{\mathtt{R}\}\end{array}} = \left\{\begin{array}{l} \mathbf{i}_?\colon \mathbf{i}.\mathit{wb}^*_{\mathbf{i}} : x_{\mathbf{i}}\ \mathtt{from\ R}; \\ \quad \mathbf{i}\colon \mathtt{while}\ x_{\mathbf{i}} \left\{\begin{array}{l} \pi(\mathcal{I},\mathtt{S}); \\ \mathbf{i}_{\mathtt{C}}\colon \mathbf{i}.\mathit{we}^*_{\mathbf{i}} : \mathsf{ok}\ \mathtt{to\ R}; \\ \mathbf{i}_?\colon \mathbf{i}.\mathit{wb}^*_{\mathbf{i}} : x_{\mathbf{i}}\ \mathtt{from\ R} \end{array}\right\} \end{array}\right.$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\},\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \notin \mathsf{roles}(\mathcal{I}) \cup \{\mathtt{R}\}\end{array}} = \mathbf{1}$$

$$\boxed{\pi(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\},\mathtt{R})} = \mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\pi(\mathcal{I},\mathtt{R})\}\ \mathtt{roles}\ \{\mathsf{roles}(\mathcal{I})\}$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\},\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \in \mathsf{roles}(\mathcal{I}) \setminus \{\mathtt{R}\}\end{array}} = \mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\pi(\mathcal{I},\mathtt{S})\}$$

$$\boxed{\begin{array}{l}\pi(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\},\mathtt{S}) \\ \quad \mathsf{and}\ \mathtt{S} \notin \mathsf{roles}(\mathcal{I}) \cup \{\mathtt{R}\}\end{array}} = \mathbf{1}$$

Figure 4.8: process-projection function π.

exit is more tricky than communication on the branch to choose in a conditional. Indeed, there are two points in the projected code where the coordinator R sends the decision: the first is inside the body of the loop and it is used if the decision is to loop; the second is after the loop and it is used if the decision is to exit. Also, there are two points where these communications are received by the other roles: before their loop at the first iteration, inside the body of the loop during previous iteration in the others.

One has to keep attention since, by splitting an interaction into a send and a receive primitive, primitives corresponding to different interactions, but on the same operation, may interfere.

Example 1. *We illustrate the issue of interferences using the two DPOC processes below, identified by their respective roles,* $\mathtt{R}_1$ *(right) and* $\mathtt{R}_2$ *(left), assuming that operations are not prefixed by indexes. We describe only* $\mathtt{R}_1$ *as* $\mathtt{R}_2$ *is its dual. At Line 1,* $\mathtt{R}_1$ *sends a message to* $\mathtt{R}_2$ *on operation* o*. In parallel with the send,* $\mathtt{R}_1$ *had a scope (Lines 3–5) that performed an update. The new code (Line 4) contains a send on operation* o *for role* $\mathtt{R}_2$*. Since the two sends and the two receives share the same channel (*o*) and run in parallel, they can interfere with each other.*

```
process R_1
1.  1: $o : e_1$ to R_2
2.  |
3.  // update auxiliary code
4.  2: $o : e_2$ to R_2
5.  // update auxiliary code
```

```
process R_2
1.  1: $o : x_1$ from R_2
2.  |
3.  // update auxiliary code
4.  2: $o : x_2$ from R_2
5.  // update auxiliary code
```

Note that this interference cannot be statically avoided since updates come from outside and one cannot know in advance which operations they use.

For this reason, in § 4.3 we introduced prefixes for DPOC operations.

A similar problem may occur also for auxiliary communications. In particular, imagine to have two parallel conditionals executed by the same role. We need to avoid that, e.g., the decision to take the "else" branch on the first conditional is wrongly taken by some role as a decision concerning the second conditional. To avoid this problem, we prefix private operations using the index $\mathbf{i}$ of the conditional. In this way, communications involving distinct conditionals cannot interact. Note that communications concerning the same conditional (or while loop) may share the same operation name and prefix. However, since all auxiliary communications are from the coordinator of the construct to the other roles involved in it, or vice versa, interferences are avoided.

We now describe how to generate indexes for statements in the projection. As a general rule, all the DPOC constructs obtained projecting a DIOC construct with index $\mathbf{i}$ have index $\mathbf{i}$. The only exception are the indexes of the auxiliary communications of the projection of conditionals and while loops.

Provided $\mathbf{i}$ is the index of the conditional: *i*) in the projection of the coordinator we index the auxiliary communications for selecting the "then" branch with index $\mathbf{i}_{\mathtt{T}}$, the ones for selecting the "else" branch with index $\mathbf{i}_{\mathtt{F}}$; *ii*) in the projection of the other roles involved in the conditional we assign the index $\mathbf{i}_{?}$ to the auxiliary receive communications. To communicate the evaluation of the guard of a while loop we use the same indexing scheme ($\mathbf{i}_{\mathtt{T}}$, $\mathbf{i}_{\mathtt{F}}$, and $\mathbf{i}_{?}$) used in the projection of conditional. Moreover, all the auxiliary communications for end of loop synchronisation are indexed with $\mathbf{i}_{\mathtt{C}}$.

4.5 Running Example: Projection and Execution

In this section we use our running example (see Figure 4.2) to illustrate the projection and execution of DIOC programs.

4.5.1 Projection

Given the code in Figure 4.2, we need to annotate it to be able to project it (we remind that in § 4.4 we defined our projection function on well-annotated DIOCs). Since we wrote one instruction per Line in Figure 4.2, we annotate every instruction using its line number as index. This results in a well-annotated DIOC.

From the annotated DIOC, the projection generates three DPOC processes for the `Seller`, the `Buyer`, and the `Bank`, respectively reported in Figures 4.9, 4.11, and 4.10. To improve readability, we omit some $\mathbf{1}$ processes. In the projection of the program, we also omit to write the prefix operations since it is always equal to the numeric part of the index of their correspondent construct. Finally, we write auxiliary communications in grey.

4.5.2 Runtime Execution

We now focus on an excerpt of the code to exemplify how updates are performed at runtime. We consider the code of the scope at Lines 6–9 of Figure 4.2. In this execution scenario we assume to introduce in the set of available updates the update presented in Figure 4.1, which enables the use of a fidelity card to provide a price discount. Below we consider both the DIOC and the DPOC level, dropping some $\mathbf{1}$s to improve readability.

```
$3_?$: $wb^*_3 : x_3$ from Buyer;
3: while ($x_3$){
  5: $priceReq : order$ from Buyer;
  6: scope @Seller{
    7: $order\_price = getPrice(order)$;
    8: $offer : order\_price$ to Buyer
    } roles { Seller, Buyer };
  $3_C$: $we^*_3 : ok$ to Buyer;
  $3_?$: $wb^*_3 : x_3$ from Buyer
};
$15_?$: $cnd^*_{15} : x_{15}$ from Buyer;
15: if ($x_{15}$){
  16: $payReq : payDesc(order\_price)$ to Bank;
  $22_?$: $cnd^*_{22} : x_{22}$ from Bank;
  22: if ($x_{22}$){
    23: $confirm$ : _ from Bank}
}
```

Figure 4.9: Seller DPOC Process.

```
$15_?$: $cnd^*_{15} : x_{15}$ from Buyer;
15: if ($x_{15}$){
  16: $payReq : desc$ from Seller;
  17: scope @Bank{
    18: $payment\_ok$ = true;
    19: $pay : auth$ from Buyer;
    // code for the payment
  } roles { Buyer, Bank };
  22: if ($payment\_ok$){
    {$22_T$: $cnd^*_{22}$ : true to Seller
     | $22_T$: $cnd^*_{22}$ : true to Buyer};
    { 23: $confirm : null$ to Seller
     | 25: $confirm : null$ to Buyer
    }
  } else {
    {$22_F$: $cnd^*_{22}$ : false to Seller
     | $22_F$: $cnd^*_{22}$ : false to Buyer};
    27: $abort : null$ to Buyer
  }
}
```

Figure 4.10: Bank DPOC Process.

```
1: $price\_ok$ = false;
2: $continue$ = true;
3: while (!$price\_ok$ and $continue$){
  $3_T$: $wb^*_3$ : true to Seller;
  4: $prod = getInput()$;
  5: $priceReq : prod$ to Seller;
  6: scope @Seller{
    7: $offer : prod\_price$ from Seller
  };
  10: $price\_ok = getInput()$;
  11: if (!$price\_ok$){
    12: $continue = getInput()$
  };
  $3_C$: $we^*_3$ : _ from Seller
};
$3_F$: $wb^*_3$ : false to Seller;
15: if ($price\_ok$){
  {
    $15_T$: $cnd^*_{15}$ : true to Seller
    | $15_T$: $cnd^*_{15}$ : true to Bank};
  17: scope $payment$@Bank{
    19: $pay : payAuth(prod\_price)$ to Bank
    // code for the payment
  }
};
$22_?$: $cnd^*_{22} : x_{22}$ from Bank;
22: if ($x_{22}$){
  25: $confirm$ : _ from Bank
} else {
  27: $abort$ : _ from Bank}
}
```

Figure 4.11: Buyer DPOC Process.

Since we describe a runtime execution, we assume that the Buyer has just sent the name of the product (s)he is interested in to the Seller (Line 5 of Figure 4.2). The annotated DIOCs we execute is the following.

```
1  6: scope @Seller{
2    7: order_price@Seller = getPrice(order);
3    8: offer : Seller(order_price) → Buyer(prod_price)
4  }
```

At runtime we apply Rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ that substitutes the scope with the new code. The replacement is atomic. Below we assume that the instructions of the update are annotated with indexes corresponding to their line number plus 30.

```
1  31: cardReq : Seller(null) → Buyer(_);
2  32: card_id@Buyer = getInput();
3  33: card : Buyer(card_id) → Seller(buyer_id);
4  34: if isValid(buyer_id)@Seller{
5       35: order_price@Seller = getPrice(order) ∗ 0.9
6  } else {
7       37: order_price@Seller = getPrice(order)
8  };
9  39: offer : Seller(order_price) → Buyer(prod_price)
```

Let us now focus on the execution at DPOC level, where the application of updates is not atomic. The scope is distributed between two participants. The first step of the update protocol is performed by the Seller, since (s)he is the coordinator of the update. The DPOC description of the Seller before the update is:

```
6: scope @Seller{
   7: order_price = getPrice(order);
   8: offer : order_price to Buyer
} roles {Seller, Buyer}
```

When the scope construct is enabled, the Seller non-deterministically selects whether to update or not and, in case, which update to apply. Here, we assume that the update using the code in Figure 4.1 is selected. Below we report on the left the reductum of the projected code of the Seller after the application of Rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$. The Seller sends to the Buyer the code — denoted as P_{B} and reported below on the right — obtained projecting the update on role Buyer.

```
1   6: $sb^*_6 : P_B$ to Buyer;
2   31: $cardReq : null$ to Buyer;
3   33: $card : buyer\_id$ from Buyer;
4   34: if $isValid(buyer\_id)$ {
5     35: $order\_price =$
6                 $getPrice(order) * 0.9$
7   } else {
8     37: $order\_price = getPrice(order)$
9   };
10  39: $offer : order\_price$ to Buyer;
11  6: $se^*_6 : \_$ from Buyer;
```

```
$P_B$ := 31: $cardReq : null$ from Seller;
        32: $card\_id = getInput()$;
        33: $card : card\_id$ to Seller;
        39: $offer : prod\_price$ from Seller
```

Above, at Line 1 the Seller requires the Buyer to update, sending to him the new DPOC fragment to execute. Then, the Seller starts to execute its own updated DPOC. At the end of execution of the new DPOC code (Line 10) the Seller waits for the notification of termination of the DPOC fragment executed by the Buyer.

Let us now consider the process-projection of the Buyer, reported below

```
6: scope @Seller{
   8: $offer : order\_price$ from Seller
}
```

At runtime, the scope waits for the arrival of a message from the coordinator of the update. In our case, since we assumed that the update is applied, the Buyer receives using Rule $\lfloor^{\text{DPOC}}|_{\text{UP}}\rceil$ the DPOC fragment P_B sent by the coordinator. In the reductum, P_B replaces the scope, followed by the notification of termination to the Seller.

```
31: $cardReq : null$ from Seller;
32: $card\_id = getInput()$;
33: $card : card\_id$ to Seller;
39: $offer : prod\_price$ from Seller
6: $se^*_6$ : ok to Seller
```

Consider now what happens if no update is applied. At DIOC level the Seller applies Rule $\lfloor^{\text{DIOC}}|_{\text{NoUp}}\rceil$, which removes the scope and runs its body. At DPOC level, the update is not atomic. The code of the Seller is the following one.

```
1  6: $sb^*_6$ : no to Buyer;
2  7: $order\_price = getPrice(order)$;
3  8: $offer : order\_price$ to Buyer;
4  6: $se^*_6 : \_$ from Buyer;
```

Before executing the code inside the scope, the `Seller` has to notify the other roles that they can proceed with their execution (Line 1). Like in the case of update, the `Seller` also waits for the notification of the end of execution from the `Buyer` (Line 4).

Finally, we report the DPOC code of the `Buyer` after the reception of the message that no update is needed. Rule $\lfloor^{\text{DPOC}}|_{\text{NoUp}}\rceil$ removes the scope and adds the notification of termination (Line 6) to the coordinator at the end.

```
7: offer : prod_price from Seller;
6: se*_6 : ok to Seller;
```

4.6 Connected DIOCs

We now give a precise definition of the notion of connectedness that we mentioned in § 4.2.2 and § 4.3.2. In both DIOC and DPOC semantics we checked such a property of updates with predicate connected, respectively in Rules $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$ (Figure 4.3) and $\lfloor^{\text{DPOC}}|_{\text{UP-LEAD}}\rceil$ (Figure 4.6).

To give the intuition of why we need to restrict to connected updates, consider the scenario below of a DIOC (left side) and its projection (right side).

$$op1 : \mathtt{A}(a) \rightarrow \mathtt{B}(b); \quad op2 : \mathtt{C}(c) \rightarrow \mathtt{D}(d) \quad \xrightarrow{projection}$$

process `A` $op_1 : a$ `to B`	process `B` $op_1 : b$ `from A`
process `C` $op_2 : c$ `to D`	process `D` $op_2 : d$ `from C`

DIOCs can express interactions that, if projected as described in § 4.4, can behave differently with respect to the originating DIOC. Indeed, in our example we have a DIOC that composes in sequence two interactions: an interaction between `A` and `B` on operation op_1 followed by an interaction between `C` and `D` on operation op_2. The projection of the DIOC produces four processes (identified by their role): `A` and `C` send a message to `B` and `D`, respectively. Dually, `B` and `D` receive a message form `A` and `C`, respectively. In the example, at the level of processes we lose the global order among the interactions: each projected process runs its code locally and it is not aware of the global sequence of interactions. Indeed, both sends and both receives are enabled at the same time. Hence, the semantics of DPOC lets the two interactions interleave in any order. It can happen that the interaction between `C` and `D` occurs before the one between `A` and `B`, violating the order of interactions prescribed originating DIOC.

Restricting to connected DIOCs avoids this kind of behaviours. We formalise connectedness as an efficient (see Theorem 1) syntactic check. More precisely, we check for *connectedness (for sequence)*, because, intuitively, we make sure that the DPOC network obtained by projecting a sequence $\mathcal{I};\mathcal{I}'$ executes first the actions in $\mathcal{I}$ and then those in $\mathcal{I}'$, thus respecting the intended semantics of sequential composition and avoiding undesired behaviours. We highlight that our definition of connectedness does not hamper programmability and it naturally holds in most of real-world scenarios (the interested reader can find at the website [89] several of such scenarios).

Remark 3. *There exists a trade-off between efficiency and ease of programming with respect to the guarantee that all the roles are aware of the evolution of the global computation. This is a common element of choreographic approaches, which has been handled in different ways, e.g.,* i*) by restricting the set of well-formed choreographies to only those that do not produce non-deterministic projections [11];* ii*) by mimicking the non-deterministic behaviour of process-level networks at choreography level [90]; or* iii*) by enforcing the order of interaction with additional auxiliary messages between roles [91].*

Our choice of preserving the order of interactions defined at DIOC level follows the same philosophy of [11], whilst for scopes, conditionals, and while loops we enforce connectedness with auxiliary messages as done in [91]. We remind that we introduced auxiliary messages for coordination in the semantics of scopes at DPOC level (§ 4.3.2) and in the projection (§ 4.4). We choose to add such auxiliary message to avoid to impose strong constraints on the form of scopes, conditionals, and while loops, which in the end would pose strong limitations to the programmers of DIOCs. On the other hand, for interactions we choose to follow a stricter approach by preserving the order of interactions defined at DIOC level.

To formalise connectedness we introduce, in Figure 4.12, the auxiliary functions transI and transF that, given a DIOC process, compute sets of pairs representing senders and receivers of possible initial and final interactions in its execution. We represent one such pair as $\mathtt{R} \to \mathtt{S}$. Actions located at $\mathtt{R}$ are represented as $\mathtt{R} \to \mathtt{R}$. For instance, given an interaction $\mathbf{i}$: $o : \mathtt{R}(e) \to \mathtt{S}(x)$ both its transI and transF are $\{\mathtt{R} \to \mathtt{S}\}$. For conditional, $\mathsf{transI}(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) = \{\mathtt{R} \to \mathtt{R}\}$ since the first action executed is the evaluation of the guard by role $\mathtt{R}$. The set $\mathsf{transF}(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\})$ is normally $\mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}')$, since the execution terminates with an action from one of the branches. If instead the branches are both empty then transF is $\{\mathtt{R} \to \mathtt{R}\}$, representing guard evaluation.

Finally, we give the formal definition of connectedness.

$$
\begin{array}{ll}
\mathsf{transI}(\mathbf{i}\colon o : \mathtt{R}(e) \to \mathtt{S}(x)) & = \mathsf{transF}(\mathbf{i}\colon o : \mathtt{R}(e) \to \mathtt{S}(x)) = \{\mathtt{R} \to \mathtt{S}\} \\
\mathsf{transI}(\mathbf{i}\colon x@\mathtt{R} = e) & = \mathsf{transF}(\mathbf{i}\colon x@\mathtt{R} = e) = \{\mathtt{R} \to \mathtt{R}\} \\
\mathsf{transI}(\mathbf{1}) & = \mathsf{transI}(\mathbf{0}) = \mathsf{transF}(\mathbf{1}) = \mathsf{transF}(\mathbf{0}) = \emptyset \\
\mathsf{transI}(\mathcal{I}|\mathcal{I}') & = \mathsf{transI}(\mathcal{I}) \cup \mathsf{transI}(\mathcal{I}') \\
\mathsf{transF}(\mathcal{I}|\mathcal{I}') & = \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}') \\
\mathsf{transI}(\mathcal{I};\mathcal{I}') & = \begin{cases} \mathsf{transI}(\mathcal{I}') & \text{if } \mathsf{transI}(\mathcal{I}) = \emptyset \\ \mathsf{transI}(\mathcal{I}) & \text{otherwise} \end{cases} \\
\mathsf{transF}(\mathcal{I};\mathcal{I}') & = \begin{cases} \mathsf{transF}(\mathcal{I}) & \text{if } \mathsf{transF}(\mathcal{I}') = \emptyset \\ \mathsf{transF}(\mathcal{I}') & \text{otherwise} \end{cases} \\
\mathsf{transI}(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) & = \mathsf{transI}(\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}) = \{\mathtt{R} \to \mathtt{R}\} \\
\mathsf{transF}(\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) & = \begin{cases} \{\mathtt{R} \to \mathtt{R}\} & \text{if } \begin{array}{c}\mathsf{transF}(\mathcal{I}) \cup \\ \mathsf{transF}(\mathcal{I}')\end{array} = \emptyset \\ \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}') & \text{otherwise} \end{cases} \\
\mathsf{transF}(\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}) & = \begin{cases} \{\mathtt{R} \to \mathtt{R}\} & \text{if } \mathsf{transF}(\mathcal{I}) = \emptyset \\ \mathsf{transF}(\mathcal{I}) & \text{otherwise} \end{cases} \\
\mathsf{transI}(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}) & = \{\mathtt{R} \to \mathtt{R}\} \\
\mathsf{transF}(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}) & = \begin{cases} \{\mathtt{R} \to \mathtt{R}\} & \text{if } \mathsf{roles}(\mathcal{I}) \subseteq \{\mathtt{R}\} \\ \bigcup_{\mathtt{R}' \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{\mathtt{R}\}} \{\mathtt{R}' \to \mathtt{R}\} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4.12: Auxiliary functions transI and transF.

Definition 10 (Connectedness)**.** *A DIOC process $\mathcal{I}$ is connected if each subterm $\mathcal{I}';\mathcal{I}''$ of $\mathcal{I}$ satisfies*

$$\forall\, \mathtt{R}_1 \to \mathtt{R}_2 \in \mathsf{transF}(\mathcal{I}'), \forall\, \mathtt{S}_1 \to \mathtt{S}_2 \in \mathsf{transI}(\mathcal{I}'')\ .\ \{\mathtt{R}_1, \mathtt{R}_2\} \cap \{\mathtt{S}_1, \mathtt{S}_2\} \neq \emptyset$$

Connectedness can be checked efficiently.

Theorem 1 (Connectedness-check complexity)**.**
The connectedness of a DIOC process $\mathcal{I}$ can be checked in time $O(n^2 \log(n))$, where n is the number of nodes in the abstract syntax tree of $\mathcal{I}$.

The proof of the theorem is reported in Appendix A.1.

We remind that we allow only connected updates. Indeed, replacing a scope with a connected update always results in a deadlock- and race-free DIOC. Thus, one just needs to statically check connectedness of the starting program and of the updates, and there is no need to perform expensive runtime checks on the whole application after updates have been performed.

4.7 Correctness

In the previous sections we have presented DIOCs, DPOCs, and described how to derive a DPOC from a given DIOC. This section presents the main technical result of this work, namely the correctness of the projection. Moreover, as a consequence of the correctness, we prove that properties like deadlock-freedom, termination, and race-freedom are preserved by the projection.

Correctness here means that the weak traces of a connected DIOC coincide with the weak traces of the projected DPOC.

Definition 11 (Trace equivalence). *A DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ and a DPOC system $\langle \mathbf{I}, \mathcal{N} \rangle$ are (weak) trace equivalent iff their sets of (weak) traces coincide.*

The proof strategy to prove our main result relies on *i*) the definition of a notion of bisimilarity which implies weak trace equivalence and *ii*) the definition of a suitable bisimulation relating each well-annotated connected DIOC system with its projection.

Definition 12 (Weak System Bisimulation). *A* weak system bisimulation *is a relation $\mathcal{R}$ between DIOC systems and DPOC systems such that if $(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}', \mathcal{N} \rangle) \in \mathcal{R}$ then:*

- *if $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle$ then $\langle \mathbf{I}', \mathcal{N} \rangle \xrightarrow{\eta_1}, \ldots, \xrightarrow{\eta_k} \xrightarrow{\mu} \langle \mathbf{I}''', \mathcal{N}''' \rangle$ with $\forall\, i \in [1 \ldots k], \eta_i \in \{o^* : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x); o^* : \mathtt{R}_1(X) \rightarrow \mathtt{R}_2(); \tau\}$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle) \in \mathcal{R}$;*

- *if $\langle \mathbf{I}', \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}''' \rangle$ with $\eta \in \{o^? : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x); o^* : \mathtt{R}_1(X) \rightarrow \mathtt{R}_2(); \surd; \mathcal{I}; \texttt{no-up}; \mathbf{I}''',$
  $\tau\}$ then one of the following two conditions holds:*

  - *$\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\eta} \langle \Sigma'', \mathbf{I}', \mathcal{I}'' \rangle$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle) \in \mathcal{R}$ or*
  - *$\eta \in \{o^* : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \rightarrow \mathtt{R}_2(), \tau\}$ and $(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle) \in \mathcal{R}$*

Due to the existence of a bisimulation between each well-annotated connected DIOC system with its projection we can prove that the projection is correct. Formally:

Theorem 2 (Correctness)**.** *For each initial, connected DIOC process $\mathcal{I}$, each state Σ, each set of updates* $\mathbf{I}$*, the DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ and the DPOC system $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ are weak trace equivalent.*

We report the full proof of Theorem 2 in Appendix A.2.

Properties. Since the projection preserves weak traces, we have that trace-based properties of the DIOC are inherited by the DPOC. A first examples of such properties is *deadlock freedom*.

Definition 13 (Deadlock freedom)**.** *An* internal *DIOC (resp. DPOC)* trace *is obtained by removing transitions labelled* $\mathbf{I}$ *from a DIOC (resp. DPOC) trace. A DIOC (resp. DPOC) system is* deadlock free *if all its maximal finite internal traces have $\surd$ as last label.*

Intuitively, internal traces are needed since labels $\mathbf{I}$ do not correspond to activities of the application and may be executed also after application termination. The fact that after a $\surd$ only rule updates are possible is captured by the following lemma.

Lemma 1. *For each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates* $\mathbf{I}$*, if $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\surd} \langle \Sigma', \mathbf{I}', \mathcal{I}' \rangle$ then the only transitions of $\langle \Sigma', \mathbf{I}', \mathcal{I}' \rangle$ have label* $\mathbf{I}''$ *for some* $\mathbf{I}''$*.*

Proof. The proof is by case analysis on the rules which can derive a transition with label $\surd$. All the cases are easy. □

Since by construction initial DIOCs are deadlock free we have that also the DPOC obtained by projection is deadlock free.

Corollary 1 (Deadlock freedom)**.** *For each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates* $\mathbf{I}$ *the DPOC system $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ is deadlock free.*

Proof. Let us first prove that for each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates $\mathbf{I}$, the DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ is deadlock free. This amounts to prove that all its maximal finite internal traces have $\surd$ as last label. For each trace, the proof is by induction on its length, and for each length by structural induction on $\mathcal{I}$. The proof is based on the fact that $\mathcal{I}$ is initial. The induction considers a reinforced hypothesis, saying also that:

- $\surd$ may occur *only* as the last label of the internal trace;
- all the DIOC systems in the sequence of transitions generating the trace, but the last one, are initial.

We have a case analysis on the top-level operator in $\mathcal{I}$. Note that in all the cases at least a transition is derivable.

Case 0 not allowed since we assumed an initial DIOC.

Case 1 trivial because by Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{END}}\rceil$ and Lemma 1 its only internal trace is $\surd$.

Case $x@\mathtt{R} = e$ the only applicable rule is $\lfloor^{\mathrm{DIOC}}|_{\mathrm{ASSIGN}}\rceil$ that in one step leads to a **1** process. The thesis follows by inductive hypothesis on the length of the trace.

Case $o^? : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ the only applicable Rule is $\lfloor^{\mathrm{DIOC}}|_{\mathrm{INTERACTION}}\rceil$, which leads to an assignment. Then the thesis follows by inductive hypothesis on the length of the trace.

Case $\mathcal{I};\mathcal{I}'$ the first transition can be derived either by Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{SEQUENCE}}\rceil$ or $\lfloor^{\mathrm{DIOC}}|_{\mathrm{SEQ\text{-}END}}\rceil$. In the first case the thesis follows by induction on the length of the trace. In the second case the trace coincides with a trace of $\mathcal{I}'$, and the thesis follows by structural induction.

Case $\mathcal{I}|\mathcal{I}'$ the first transition can be derived either by Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{PARALLEL}}\rceil$ or by Rule [PAR-END]. In the first case the thesis follows by induction on the length of the trace. In the second case the thesis follows by Lemma 1, since the label is $\surd$.

Case `if` $b@\mathtt{R}\ \{\mathcal{I}\}$ `else` $\{\mathcal{I}'\}$ the first transition can be derived using either Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{IF\text{-}THEN}}\rceil$ or Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{IF\text{-}ELSE}}\rceil$. In both the cases the thesis follows by induction on the length of the trace.

Case `while` $b@\mathtt{R}\ \{\mathcal{I}\}$ the first transition can be derived using either Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{WHILE\text{-}UNFOLD}}\rceil$ or Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{WHILE\text{-}EXIT}}\rceil$. In both the cases the thesis follows by induction on the length of the trace.

Case `scope` $@\mathtt{R}\ \{\mathcal{I}\}$ the first transition can be derived using either Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{UP}}\rceil$ or Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{NOUP}}\rceil$. In both the cases the thesis follows by induction on the length of the trace.

The weak internal traces of the DIOC coincide with the weak internal traces of the DPOC by Theorem 2, thus the finite weak internal traces end with $\surd$. The same holds for the finite strong internal traces, since label $\surd$ is preserved when moving between strong and weak traces, and no transition can be added after the $\surd$ thanks to Lemma 1. □

DPOCs also inherit termination from terminating DIOCs.

Definition 14 (Termination)**.** *A DIOC (resp. DPOC) system terminates if all its internal traces are finite.*

Note that with arbitrary sets of updates DIOC termination is never granted if they contain at least a scope, since it can always be replaced by a non-terminating update, or it can trigger an infinite chain of updates. Thus, to exploit this result, one should add constraints on the set of updates ensuring DIOC termination.

Corollary 2 (Termination)**.** *If the DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ terminates and $\mathcal{I}$ is connected then the DPOC system $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ terminates.*

Proof. It follows from the fact that only a finite number of auxiliary actions are added when moving from DIOCs to DPOCs. □

Other interesting proprieties derived from weak trace equivalence are freedom from races and orphan messages. A race occurs when the same receive (resp. send) may interact with different sends (resp. receives). In our setting, an orphan message is an enabled send that is never consumed by a receive. Orphan messages are more relevant in asynchronous systems, where a message may be sent, and stay forever in the network, since the corresponding receive operation may never become enabled. However, even in synchronous systems orphan messages should be avoided: the message is not communicated since the receive is not available, hence a desired behaviour of the application never takes place due to synchronisation problems.

Trivially, DIOCs avoid races and orphan messages since send and receive are bound together in the same construct. Differently, at the DPOC level, since all receive of the form $\iota\colon \mathbf{i}.o^? : x$ `from` $\mathtt{R}_1$ in role $\mathtt{R}_2$ may interact with the sends of the form $\iota\colon \mathbf{i}.o^? : e$ `to` $\mathtt{R}_2$ in role $\mathtt{R}_1$, races may happen. However, thanks to the correctness of the projection, race freedom holds also for the projected DPOCs.

Corollary 3 (Race freedom)**.** *For each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates $\mathbf{I}$, if $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_n} \langle \mathbf{I}', \mathcal{N} \rangle$, where $\eta_i \in \{\tau, o^? : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \surd, \mathcal{I}, \mathsf{no\text{-}up}, \mathbf{I}\}$ for each $i \in \{1, \ldots, n\}$, then in $\mathcal{N}$ there are no two sends (resp. receives) which can interact with the same receive (resp. send).*

Proof. We have two cases, respectively for public and private operations.

For public operations, thanks to Lemma 7, case C1, for each global index ξ there are at most two communication events with global index ξ. The corresponding DPOC terms can be enabled only if they are outside of the body of a while loop. Hence, their index coincides with their global index. Since the index prefixes the operation, then no interferences with other sends or receives are possible.

For private operations, the reasoning is similar. Note, in fact, that sends or receives with the same global index can be created only by a unique DIOC construct, but they are never enabled together. This can be seen by looking at the definition of the projection. □

As far as orphan messages are concerned, they may appear in infinite DPOC computations since a receive may not become enabled due to an infinite loop. However, as a corollary of trace equivalence, we have that terminating DPOCs are orphan-message free.

Corollary 4 (Orphan-message freedom)**.** *For each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates $\mathbf{I}$, if $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma)\rangle \xrightarrow{\eta_1} \cdots \xrightarrow{\eta_n} \xrightarrow{\surd} \langle \mathbf{I}', \mathcal{N}\rangle$, where $\eta_i \in \{\tau, o^? : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \surd, \mathcal{I}, \mathsf{no\text{-}up}, \mathbf{I}\}$, then $\mathcal{N}$ contains no sends.*

Proof. The proof is by case analysis on the rules which can derive a transition with label $\surd$. All the cases are easy. □

CHAPTER 5

Adaptable Interaction-Oriented Choreographies in Jolie

5.1 Introduction

In this section, we present AIOCJ (Adaptable Interaction-Oriented Choreographies in Jolie), a development framework for adaptable distributed applications [55]. AIOCJ is one of the possible instantiations of our theory of Dynamic Choreographies presented in Chapter 4 and it gives a tangible proof of the expressiveness and feasibility of our approach, enabling adaptation of distributed programs (cf. Chapter 3).

We say that AIOCJ is an instance of our theory because it follows the theory, but it provides mechanisms to resolve the non-determinism related to the choice of whether to update or not and on which update to select. Indeed, in AIOCJ updates are chosen and applied according to the state of the system and of its running environment. To this end, updates are embodied into adaptation rules, which specify when and whether a given update can be applied, and to which scopes. AIOCJ also inherits all the correctness guarantees provided by our theory, in particular: *i*) applications are free from deadlocks and races by construction, *ii*) applications remain correct after any step of adaptation, and *iii*) it is possible to add and remove adaptation rules at runtime.

Below we give a brief overview of the AIOCJ framework by introducing its components: the Integrated Development Environment (IDE), the AIOCJ compiler, and the Runtime Environment.

Integrated Development Environment. AIOCJ supports the writing of programs and adaptation rules in the Adaptable Interaction-Oriented Choreography (AIOC) language, an extension of the DIOC language. We discuss the main novelties of the AIOC language in § 5.1.1. AIOCJ offers an integrated environment for developing programs and adaptation rules that supports syntax highlighting and real-time[1] syntax checking. Since checking for *connectedness* (see § 4.6) of programs and adaptation rules is polynomial (as proven by Theorem 1), the IDE also performs real-time checks on connectedness of programs and rules.

Compiler. The AIOCJ IDE also embeds the AIOCJ compiler, which implements the procedure for projecting AIOCs and adaptation rules into distributed executable code. The implementation of the compiler is based on the rules for projecting DIOCs, described in § 4.4. The target language of the AIOCJ compiler is the Jolie [92] language, which sports primitives similar to those of our DPOC language. Given an AIOC program, the AIOCJ compiler produces one Jolie program, also called service, for each role in the source AIOC. The compilation of an AIOC rule produces one Jolie service for each role and an additional service that describes the applicability condition of the rule. All these services are enclosed into an Adaptation Server, described below.

Runtime Environment. The AIOCJ runtime environment comprises a few Jolie services that support the execution and adaptation of compiled programs. The main services of the AIOCJ runtime environment are the Adaptation Manager, Adaptation Servers, and the Environment. The compiled services interact both among themselves and with an Adaptation Manager, which is in charge of managing the adaptation protocol. Adaptation Servers contain adaptation rules, and they can be added or removed dynamically, thus enabling dynamic changes in the set of rules, as specified by Rule $\lfloor^{\text{DIOC}}|_{\text{Change-Updates}}\rceil$. When started, an Adaptation Server registers itself at the Adaptation Manager. The Adaptation Manager invokes the registered Adaptation Servers to check whether their adaptation rules are applicable. In order to check whether an adaptation rule is applicable, the corresponding Adaptation Server evaluates its applicability condition. Applicability conditions may refer to the state of the role which coordinates the update, to properties of the scope, and to properties of the environment, stored in the Environment service.

In the remainder of this Section we detail the grammar of the AIOC language used by AIOCJ in § 5.1.1, we illustrate the use of AIOCJ on a simple example in § 5.2, we discuss some relevant implementation aspects of AIOCJ in § 5.3, and we provide provide a preliminary validation in § 5.4.

5.1.1 DIOC Language Extensions in AIOCJ

AIOCs extend DIOCs with:

[1]In this context, real-time means “while the developer is writing AIOC programs and rules”.

- the definition of adaptation rules, instead of updates, that include the information needed to evaluate their applicability condition;
- the definition of constructs to express the deployment information needed to implement real-world distributed applications.

Below we describe in detail, using Extended Backus-Naur Form [93], the new or refined constructs introduced by the AIOC language.

Function inclusions. The AIOC language can exploit functionalities provided by external services via the `include` construct. The syntax is as follows.

```
Include ::= include ID [,ID]* from "URL" [with PROTOCOL]+
```

This allows one to reuse existing legacy code and to interact with third-party external applications. As an example, the `Seller` of our running example can exploit an external database to implement the functionality for price retrieval `getPrice`, provided that such a functionality is exposed as a service. If the service is located at `"socket://myService:8000"` and accessible via the `"HTTP"` protocol we enable its use with the following inclusion:

```
include getPrice from "socket://myService:8000" with "HTTP"
```

This feature enables a high degree of integration since AIOCJ supports all protocols provided by the underlying Jolie language, which include TCP/IP, RMI, SOAP, XML/RPC and their encrypted correspondents over SSL.

External services perfectly fit the theory described in previous sections, since they are seen as functions, and thus introduced in expressions. In order for the theory to apply they need to satisfy the condition of never blocking and always returning a value (possibly an error notification), exactly as other expressions.

Adaptation rules. Adaptation rules extend DIOC updates, and are a key ingredient of the AIOCJ framework. The syntax of adaptation rules is as follows:

```
Rule ::= rule {
  [ Include ]*
  on { Condition }
  do { Choreography }
}
```

The applicability condition of the adaptation rule is specified using the keyword `on`, while the code to install in case adaptation is performed (which corresponds to the DIOC update) is specified using the keyword `do`. Optionally, adaptation rules can include functions they rely on.

The `Condition` of an adaptation rule is a propositional formula which specifies when the rule is applicable. To this end it can exploit three sources of information: local variables of the coordinator of the update, environmental variables, and properties of the scope to which the adaptation rule is applied. Environmental variables are meant to capture contextual information that is not under the control of the application (e.g., temperature, time, available resources, . . .). To avoid amibiguities, local variables of the coordinator are not prefixed, environment variables are prefixed by `E`, and properties of the scope are prefixed by `N`.

For example, if we want to apply an adaptation rule only to those scopes whose property `name` is equal to the string `"myScope"` we can use as applicability condition the formula `N.name == "myScope"`.

Scopes. As described above, scopes in AIOC also feature a set of properties (possibly empty). Scope properties describe the current implementation of the scope, including both functional and non-functional properties. Such properties are declared by the programmer, and the system only use them to evaluate the applicability condition of adaptation rules, to decide whether a given adaptation rule can be applied to a given scope. Thus the syntax for scopes in AIOC is:

```
scope @ID { Choreography }
[ prop { Properties } ]?
```

where clause `prop` introduces a list of comma-separated assignments of the form `N.ID = Expression`.

For instance, the code

```
scope @A {
  // AIOC code
} prop { N.name = "myScope"}
```

specifies that the scope has a property `name` set to the string `"myScope"`, thus satisfying the condition of the adaptation rule above.

Programs. AIOC programs have the following structure.

```
Program ::=
  [ Include ]*
  preamble {
    starter: Role
    [ Location ]*
  }
  aioc { Choreography }
```

where `Include` allows one to include external functionalities, as discussed above, and keyword `aioc` introduces the behaviour of the program, which is a DIOC apart for the fact that scopes may define properties, as specified above.

The keyword `preamble` introduces deployment information, i.e., the definition of the `starter` of the AIOC and the `Location` of participants.

The definition of a `starter` is mandatory and designs which role is in charge of waiting for all other roles to be up and running before starting the actual computation. Any role can be chosen as `starter`, but the chosen one needs to be started first when running the distributed application.

`Locations` define where the participants of the AIOC will be deployed. They are specified using the keyword `location`:

```
Location ::= location@ID:"URL"
```

where ID is the name of a role (e.g., `Role1`) and URL specifies where the service can be found (e.g., `"socket://Role1:8001"`). When not explicitly defined, the projection automatically assigns a distinct local TCP/IP location to each role.

5.2 AIOCJ Practice

Here we present a brief description of how a developer can write an adaptable distributed system in AIOCJ, execute it, and change its behaviour at runtime by means of adaptation rules. For simplicity, we reuse here the minimal example presented in the Introduction, featuring a scope that encloses a price offer from the `Seller` to the `Buyer` and an update — here an adaptation rule — that provides a discount for the `Buyer`. We report the AIOC program in the upper part and the adaptation rule in the lower part of Figure 5.1.

At Line 1 of the AIOC program we have the `preamble`. The preamble specifies deployment information, and, in particular, defines the `starter`, i.e., the service that ensures that all the participants are up and running before starting the actual computation. No locations are specified, thus default ones are used. The actual code is at Lines 3–6, where we declare a scope. At Line 6 we define a property `scope_name` of this scope with value `"price_inquiry"`.

Figure 5.2 depicts the process of compilation ① and execution ② of the AIOC. From left to right, we write the AIOC and we compile it into a set of executable Jolie services (Buyer service and Seller service). To execute the projected system, we first launch the Adaptation Manager and then the two compiled services, starting from the Seller, which is the `starter`. Since there is no compiled adaptation rule, the result of the execution is the offering of the standard price to the `Buyer`.

Now, let us suppose that we want to adapt our system to offer discounts in the Fall season. To do that, we can write the adaptation rule shown in the lower part of Figure 5.1.

Since we want to replace the scope for the `price_inquiry`, we define that this rule applies only to scopes with property `scope_name` set to `"price_inquiry"`. Furthermore, since we want the update to apply only in the Fall season, we also specify that the environment variable `E.season` should match the value `"Fall"`. The value of `E.season` is retrieved from the Environment Service.

The body of the adaptation rule specifies the same behaviour described in Fig-

```
preamble{ starter: Seller }
aioc {
 scope @Seller {
  order_price@Seller = getPrice( order );
  offer: Seller( order_price ) -> Buyer( prod_price )
} prop { N.scope_name = "price_inquiry" }
```

```
rule {
 on { N.scope_name == "price_inquiry" and E.season == "Fall" }
 do {
  cardReq: Seller( _ ) -> Buyer( _ );
  card_id@Buyer = getInput( "Insert your customer card ID");
  cardRes: Buyer( card_id ) -> Seller( buyer_id );
  if isValid( buyer_id )@Seller {
   order_price@Seller = getDiscountedPrice( order, buyer_id );
  } else {
   order_price@Seller = getPrice( order )
  };
  offer: Seller( order_price ) -> Buyer( prod_price )
  }
}
```

Figure 5.1: An AIOC program (upper part) and an applicable adaptation rule (lower part).

ure 4.1 in the Introduction: the `Seller` asks the `Buyer` to provide its `card_id`, which the `Seller` uses to provide a discount on the price of the ordered product. We depict the inclusion of the new adaptation rule and the execution of the adaptation at point ③ of Figure 5.2 (outlined with dashes). From right to left, we write the rule and we compile it. The compilation of a (set of) adaptation rule(s) in AIOCJ produces a service, called Adaptation Server, that the Adaptation Manager can query to fetch adaptation rules at runtime. The compilation of the adaptation rule can be done while the application is running. After the compilation, the generated Adaptation Server is started and registers on the Adaptation Manager. Since the rule relies on the environment, in order to check its applicability condition we also need the Environment service to be running. In order for the adaptation rule to apply we need the environment variable season to have value `"Fall"`.

5.3 Implementation

AIOCJ is composed of two elements: the AIOCJ Integrated Development Environment (IDE), named AIOCJ − ecl, and the adaptation middleware that enables

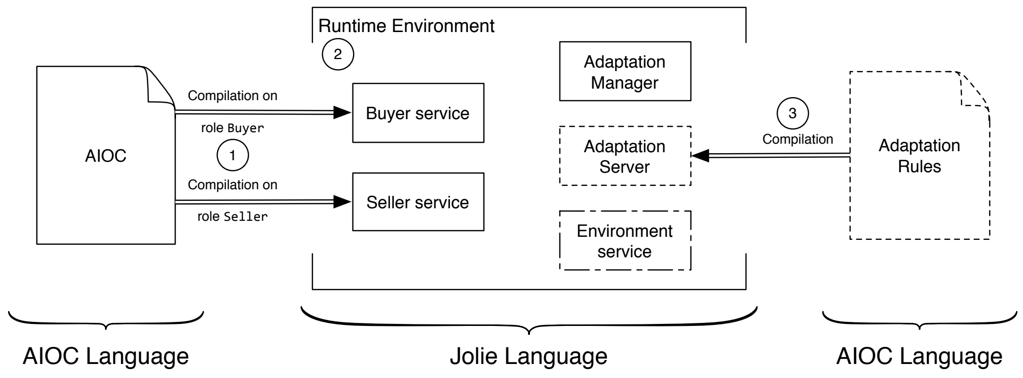


Figure 5.2: Representation of the AIOCJ framework — Projection and execution of the example in Figure 5.1.

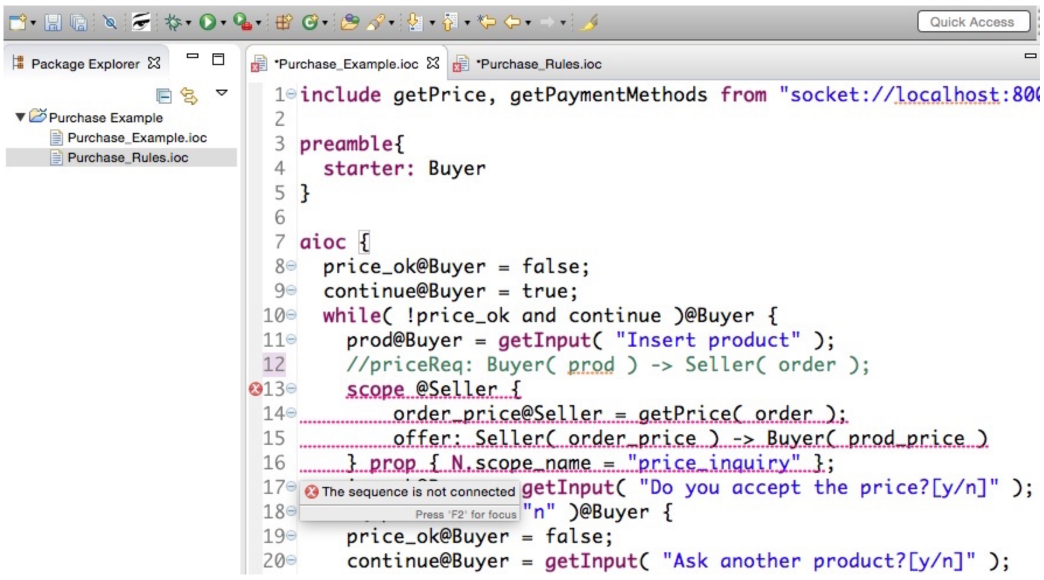
AIOC programs to adapt, called AIOCJ – mid.

AIOCJ – ecl is a plug-in for Eclipse [94] based on Xtext [95]. Starting from a grammar, Xtext generates the parser for programs written in the AIOC language. Result of the parsing is an abstract syntax tree (AST) we use to implement *i*) the checker of connectedness for AIOC programs and adaptation rules and *ii*) the generation of Jolie code for each role. Since the check for connectedness has polynomial computational complexity (cf. § 4.6) it is efficient enough to be performed on-the-fly while editing the code. Figure 5.3 shows AIOCJ – ecl notifying the error on the first non-connected instruction (Line 13).

As already mentioned, we chose Jolie as target language of the compilation of AIOCJ because its semantics and language constructs naturally lend themselves to translate our theoretical results into practice. Indeed, Jolie supports architectural primitives like dynamic embedding, aggregation, and redirection, which ease the compilation of AIOCs.

Each scope at the AIOC level is projected into a specific sub-service for each role. The roles run the projected sub-services by embedding them and access them via redirection. In this way, we implement adaptation by disabling the default sub-service and by redirecting the execution to a new one, obtained from the Adaptation server.

When at runtime the coordinator of the update reaches the beginning of that scope, it queries the Adaptation Manager for adaptation rules to apply. The Adaptation Manager queries each Adaptation Server sequentially, based on their order of registration. On its turn, each Adaptation Server checks the applicability condition of each of its rules. The first rule whose applicability condition holds is applied. The adaptation manager sends to the coordinator the updates that are then distributed to the other involved roles. In each role, the new code replaces



```
include getPrice, getPaymentMethods from "socket://localhost:80

preamble{
  starter: Buyer
}

aioc {
  price_ok@Buyer = false;
  continue@Buyer = true;
  while( !price_ok and continue )@Buyer {
    prod@Buyer = getInput( "Insert product" );
    //priceReq: Buyer( prod ) -> Seller( order );
    scope @Seller {
        order_price@Seller = getPrice( order );
        offer: Seller( order_price ) -> Buyer( prod_price )
    } prop { N.scope_name = "price_inquiry" };
                  getInput( "Do you accept the price?[y/n]" );
                  "n" )@Buyer {
    price_ok@Buyer = false;
    continue@Buyer = getInput( "Ask another product?[y/n]" );
```

The sequence is not connected

Figure 5.3: Check for connectedness: sequence.

the old one.

For reason of performances, the implementation differs from the theory in two respects. First, adaptation rules are compiled statically and not by the coordinator of the update when the update is applied. Second, to ensure that operations coming from distinct constructs do not interfere — what is guaranteed in the theory by prefixing operations with fresh indexes — we statically check that each program and each adaptation rule never features two interactions using the same operation in parallel (we call this property connectedness for parallel), and we run AIOCs coming from different programs or updates in different namespaces, thus forbidding interferences between them.

5.4 Validation

In this section, we give a preliminary empirical validation of our implementation. The main aim is to test how our mechanisms for adaptation impact on performances.

In the literature, to the best of our knowledge, there is no approach to adaptation based on choreography programming. Thus, it is difficult to directly compare our results with other existing approaches. Moreover, we are not aware of any established benchmark to evaluate adaptive applications. For this reason, we tested AIOCJ performances by applying it to two typical programming patterns: *pipes* and *fork-joins*.

Since we are interested in studying the cost of adaptation, our scenarios contain minimal computation and are particularly affected by the overhead of the adaptation process. Clearly, the percentage of the overhead due to adaptation will be far lower in real scenarios, which are usually more computationally intensive. In the first scenario, we program a pipe executing n tasks (in a pipe, the output of task t_i is given as input to task t_{i+1}, for $i \in \{1, \ldots, n-1\}$). To keep computation to a minimum, each task simply computes the increment function.

In the fork-join scenario, n tasks are computed in parallel. Each task processes one character of a message of length n, shifting it by one position. The message is stored in an external service.

The code of both scenarios is in Appendix B.1.

To enable adaptation, each task is enclosed in a scope. We test both scenarios with an increasing number of tasks $n \in \{10, 20, \ldots, 100\}$ to study how performances scale as the number of adaptation scopes increases. We evaluate performances in different contexts, thus allowing us to understand the impact of different adaptation features, such as scopes, adaptation servers, and adaptation rules.

Context 1 no scopes, no adaptation servers, no rules;

Context 2 each task is enclosed in a scope, no adaptation servers, no rules;

Context 3 each task is enclosed in a scope, one adaptation server, no rules;

Context 4 as Context 3, but now the adaptation server contains 50 rules. Each rule is applicable to a unique scope i, and no rule is applicable to scopes with $i > 50$. The rules are stored in random order.

Context 5 as Context 4, but with 100 rules, one for each scope.

Each rule in Contexts 4 and 5 is applicable to one specific scope only (through a unique property of the scope), hence when testing for 50 rules, only the first 50 scopes adapt.

In order to reduce variability of performances, we repeated every test 5 times. We performed our tests on a machine equipped with a 2.6GHz quad-core Intel Core i7 processor and 16GB RAM. The machine runs Mavericks 10.9.3, Java 1.7.55, and Jolie r.2728. Figure 5.4 shows the tests for the pipe (left) and the fork-join (right). Both charts display on the x-axis the number of tasks/scopes and on the y-axis the execution time in milliseconds.

As expected, in both scenarios there is a significant gap between Contexts 1 and 2. In words, the introduction of scopes has a strong effect on performances. The ratio is 1:13 for the pipe scenario and 1:5.5 for the fork-join scenario. This is due to the auxiliary communications needed to correctly execute a scope. The

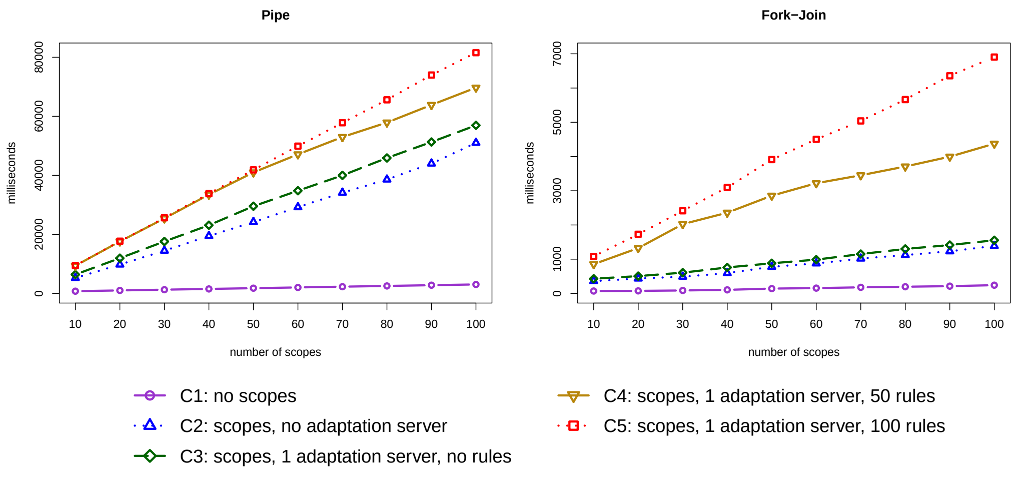


Figure 5.4: Times of execution of the pipe (left) and the fork-join (right) scenarios

observed overhead is higher in the pipe scenario, since different scopes check for adaptation in sequence, while this is done in parallel for the fork-join scenario.

Adding an adaptation server (from Context 2 to Context 3) has little impact on performances: 19% of decay for pipe, and 17% for fork-join. That is reasonable, considered that Context 3 just adds one communication wrt Context 2.

On the contrary, there is a notable difference when adding rules to the adaptation server (Context 4 is 1.4 times slower than Context 3 for the pipe scenario, 2.9 for the fork-join scenario). In Contexts 4 and 5, performances are really close up to 50 scopes (in the pipe scenario they almost overlap) although Context 5 has twice the rules of Context 4. This illustrates that the time to test for applicability of rules is negligible. Hence, the highest toll on performances is related to actual adaptation, since it requires to transfer and embed the new code. This is particularly evident in the fork-join scenario where multiple adaptations are executed in parallel and the adaptation server becomes a bottleneck. This problem can be mitigated using multiple distributed adaptation servers.

The fact that the most expensive operations are scope execution and actual adaptation is highlighted also by the results below. The table shows the cost of different primitives, including scopes in different contexts. Times are referred to 5 executions of the sample code in Appendix B.1.

As future work we will exploit these results to increase the performances of our framework, concentrating on the bottlenecks highlighted above. For instance, scope execution (as well as conditionals and cycles) currently requires many auxiliary communications ensuring that all the processes agree on the chosen path. In many cases, some of these communications are not needed, since a process will eventually discover the chosen path from the protocol communications. Static

Test	Time (ms)	Test	Time (ms)
assignment	**2.2**	scope, 1 adaptation server, 1 matching rule	**280.6**
interaction	**4.2**	scope, 1 adaptation server, 50 rules, none matching	**254.2**
if statement	**16.6**	scope, 1 adaptation server, 50 rules, 1 matching	**338.6**
scope, no adaptation server	**129.4**	scope, 1 adaptation server, 100 rules, none matching	**310.2**
scope, 1 adaptation server, no rule	**203.8**	scope, 1 adaptation server, 100 rules, 1 matching	**385**

analysis can discover redundant communications and remove them. Another improvement is letting the adaptation server send the new code directly to the involved roles, skipping the current forward chain.

Part III

Applied Choreographies

CHAPTER 6

Applied Choreographies

There is nothing as practical as a good theory.

Kurt Z. Lewin

6.1 Introduction

Our work on Dynamic Choreographies in Chapter 4 and the implementation described in Chapter 5 proved that choreographies are a suitable model to tackle challenging and unsolved problems of distributed systems.

However, we highlight that in the development of AIOCJ, we took a sensible departure from the theoretical model of DIOCs. Indeed, in our theory we defined an EPP that targets the DPOC language, a process-level language that bases message passing on *names*, like DIOCs do. Synchronisation on names (as in CCS and the π-calculus [49, 5]) simplifies the theoretical treatment and it is used in many other works on choreographies as programming abstractions [10, 12, 43, 96, 13, 97]. Nevertheless, the compiler provided by AIOCJ targets the Jolie language [92, 50], which handles communications with *message correlation*, a standard technique in Service-Oriented Computing and Web Services that routes messages inspecting the data they carry (e.g., headers) [36, 98]. This makes the EPP provided by AIOCJ much more technically involved than its formal specification, including the management of underlying data structures (e.g., message queues) and unexpected — not present in the theoretical treatment — communications in the resulting executable code. We remark that a similar occurrence exists between the choreographic language Chor [99] and its theoretical model [13] and, more in general, that it is a typical characteristic of implementations of process calculi, see, e.g., [100, 52].

This key difference between formal models and implementations can compromise the benefits of choreographic programming: the correctness-by-construction approach and the clear specification of the communications carried out during execution. Thus we ask:

How can we formalise the implementation of communications in choreographies?

A satisfactory answer should preserve the correctness-by-construction guarantees of choreography models down to the level of how communications are concretely implemented.

A challenging task that requires the definition of a model with *i*) the typical clarity of choreography languages and with *ii*) all the necessary details to formally reason on how to support communications at the lower level.

Our answer is to develop a theory of *Applied Choreographies* (AC). AC provides the simple and intuitive syntax of choreography languages along with the most advanced features of recent choreography models like modularity, asynchrony, parallelism, and the dynamic creation of multiparty sessions and processes. The key contribution of AC lies in its semantics: equipped with a novel notion of *deployment*, it lets us reason on how messages are routed between participants. Deployments separate the logic of choreographies (message passing, creation of sessions and processes) from their implementation. Notably, by just changing the definition of deployments and their effects we can easily formalise different implementation models, e.g., distributed objects [101], correlation, etc..

In this work, we chose to model correlation because clarifying the relationship between this communication model and choreographies is a relevant and challenging task, whose solution immediately yields a practical impact. Indeed, most practitioners use choreographies to describe correct distributed systems, but they have to deal with how channel-based interactions (e.g., as in WS-CDL [8]) can be correctly implemented with message correlation (used in most major frameworks, including WS-BPEL [36], Java/JMS, C#/.NET, etc.).

Here, we model correlation on notions from Service-Oriented Computing, the setting where choreographies are used the most as design tool (cf. § 2.4). We pinpoint the key theoretical problems and formalise the guiding principles developers should follow to obtain correct implementations. On this result, we define a correct compiler from AC choreographies to the theoretical model behind the Jolie language.

Contributions. We list below our main contributions:

Applied Choreographies. AC is a choreography model for the modular development of choreographies based on asynchronous message passing. AC captures how communications among processes are concretely supported via message correlation (§ 6.2). The main novel aspect of AC is a notion of *deployment* for choreographies which, for each process in the choreography, keeps track of *i*) the location of the service in which it executes (multiple processes can run in the same service), *ii*) its input message queues (for asynchronous communications), and *iii*) its state (the value of its variables). This yields a close representation of how real-world service-oriented scenarios implement communications, which is the basis for all of our results.

Type system. We modelled real-world message passing in AC to formalise how programs may encounter execution errors. For example, it may be impossible to route a message from one endpoint to another, due to missing information in the deployment of a choreography (e.g., a suitable input queue at the receiver). This is a significant departure from previous choreography models, where it is assumed that communications between two processes can always be performed [10, 43, 12, 102, 96, 13] (cf. Chapter 4). In § 6.3 we present a typing discipline for AC that prevents communication errors. Our type language is that of standard multiparty session types [43]. This is a remarkable feature of AC: despite the additional complexity of its semantics, the complexity of the types a programmer needs to specify to check a choreography remain unchanged.

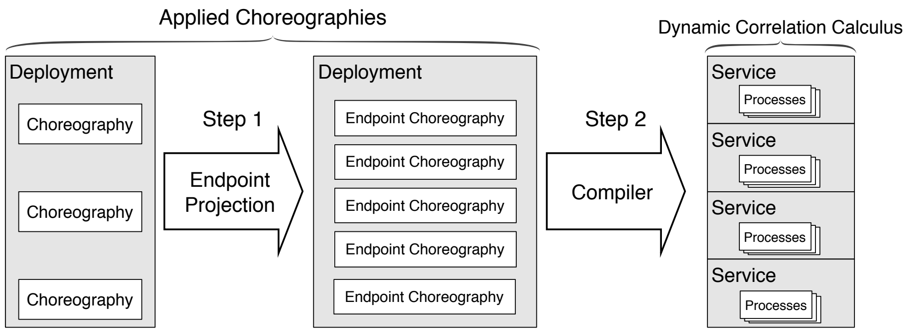


Figure 6.1: Overall methodology of Applied Choreographies.

Compilation of AC programs. We define a two-step methodology to transform a choreography in AC into programs of the *Dynamic Correlation Calculus* (DCC), a model for executable code based on message correlation. We depict our methodology in Figure 6.1. The first step is a source-to-source Endpoint Projection (as seen in [14]) which projects a choreography describing the behaviour of many participants to a series of choreography modules, called *Endpoint Choreographies*, each describing the behaviour of a single participant. The second step is a compilation from endpoint choreographies to DCC programs, which define the behaviour of *services* following the Service-Oriented model. Specifically, DCC formalises a syntax and a semantics for a fragment of the Jolie language. We chose Jolie because its reference implementation is based on a formal specification [98] called

Correlation Calculus (CC). DCC improves CC by adding message queues that can be created at runtime, a necessary feature to support our choreography model. We prove that the compiled processes implement the behaviour of the EPP and therefore, that of the originating AC program.

6.2 Applied Choreography Language

We introduce Applied Choreographies (AC) that, on key elements of choreography calculi like *processes*, *sessions*, and *roles*, introduces the notion of *locations*.

6.2.1 Syntax

In the syntax of AC (Figure 6.2) C denotes a choreography, p, q processes, $\mathtt{A}, \mathtt{B}$ roles, k sessions, o operations, l locations, X procedures, and x variables[1]. We consider all sets of identifiers disjoint. Processes are independent execution units that proceed in parallel. They can communicate with each other through sessions. Roles track which role each process plays in a session. As in standard multiparty session types [43] Roles are the basis for our typing discipline in § 6.3. Locations represent publicly reachable addresses, where we assume that an always-available service supports the creation and the execution of new processes. We introduce the notion of location to model the deployment of processes into services. Following the SOC model, a service is a container, reachable at a defined address (the location), where processes execute in parallel. In § 6.6, when compiling ACs to the lower language DCC, we map each location to a concrete service implementation. Using locations makes us depart from previous choreography languages, which do not consider the deployment of processes in their models. AC includes complete and partial actions to support modularity, as in previous choreography models [14]. A complete action specifies the behaviour of all participants involved in the action. A partial action describes the behaviour of only some participants. Partial actions enable compositionality: choreographies with compatible partial actions can be composed in parallel.

Complete Actions. Term $(start)$ denotes session initiation: process p starts a new session k together with processes $\tilde{\mathsf{q}}$. p, called *active process*, is already running, whereas each process q in $\widetilde{l.\mathsf{q}}$, called *service process*, is dynamically created at its respective location l in $\widetilde{l.\mathsf{q}}$. Service locations can be used repeatedly to

[1]Variables are *paths* to traverse structured data, e.g., x can be a path $\underline{\mathbf{x.y.z}}$ where "." is the path nesting operator. § 6.2.2 formalises variables and paths.

spawn multiple processes, even inside recursions. We assume $\widetilde{l.\mathsf{q}}$ always non-empty. Term (com) models a communication: on session k, process p sends to process q a message for operation o; the message carries the evaluation of expression e on the local state of p whilst x is the variable where q will store the content of the message.

Partial Actions. In term (req), process p requests the creation of some external processes at their respective locations $\tilde{l}$ to start a new session k. $\widetilde{l.\mathtt{B}}$ means that each location l in $\widetilde{l.\mathtt{B}}$ is expected to spawn a process that behaves as specified by the related role $\mathtt{B}$. The dual of (req) is term (acc), which provides the implementation of service processes. Specifically, term (acc) defines a reusable choreography module that accepts, at locations $\tilde{l}$, the creation of processes $\tilde{\mathsf{q}}$ playing their respective roles $\tilde{\mathtt{B}}$. Following the design idea that services should always be available, shared by other models [96, 13], we assume that all (acc) terms in a choreography are at top level (not guarded by other actions). Term $(send)$ models the sending of a message, for operation o, from process p to an external process playing role $\mathtt{B}$ in session k. Dually, in term $(recv)$, process q receives a message from an external process playing role $\mathtt{A}$ in session k; q proceeds with the continuation corresponding to the operation where the message was received, e.g., receiving the message on o_j, q proceeds with continuation C_j — like in the standard branching found in session-oriented calculi [44].

Other Terms. In a conditional $(cond)$ process p evaluates a condition e in its local state to choose between the continuations C_1 and C_2. Term (par) is the standard parallel composition of choreographies, as in [14, 97]. Terms (def), $(call)$, and $(inact)$ denote respectively the standard definition of recursive procedures, procedure calls, and inaction. Some terms bind identifiers in continuations. In terms $(start)$ and (acc), the session identifier k and the process identifiers $\tilde{\mathsf{q}}$ are bound (as they are freshly created). All other identifiers are free. In terms (com) and $(recv)$, the variables used by the receiver to store the message are bound (x and all the x_i, respectively). In term (req), the session identifier k is bound. Finally, in term (def), the procedure identifier X is bound. In the remainder, we omit $\mathbf{0}$ or irrelevant variables (e.g., in communications with empty messages).

Remark 4. *As in [13], AC terms, except for $(start)$, (req), and (acc), do not need to be annotated with roles as we can infer them from session identifiers. However, we include roles in all AC terms to ease the treatment of our typing discipline in § 6.3.*

$$
\begin{array}{rlll}
C ::= & \eta; C & (\mathit{seq}) & \mid \ \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} & (\mathit{cond}) \\
\mid & C_1 \mid C_2 & (\mathit{par}) & \mid \ \mathsf{def}\ X\langle\tilde{\mathsf{p}}\rangle = C'\ \mathsf{in}\ C & (\mathit{def}) \\
\mid & X & (\mathit{call}) & \mid \ \mathbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]}; C & (\mathit{acc}) \\
\mid & \mathbf{0} & (\mathit{inact}) & \mid \ k : \mathtt{A}\ \texttt{->}\ \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} & (\mathit{recv})
\end{array}
$$

$$
\begin{array}{rllll}
\eta ::= & \mathbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]} & (\mathit{start}) & \mid \ \mathbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}} & (\mathit{req}) \\
\mid & k : \mathsf{p}[\mathtt{A}].e\ \texttt{->}\ \mathsf{q}[\mathtt{B}].o(x) & (\mathit{com}) & \mid \ k : \mathsf{p}[\mathtt{A}].e\ \texttt{->}\ \mathtt{B}.o & (\mathit{send})
\end{array}
$$

Figure 6.2: Choreography Calculus - Syntax

6.2.2 Semantics

The semantics of AC is one of our major contributions: it formally captures, on the level of choreographies, the real-world communication mechanism found in SOC, called message correlation [36]. We first give an informal overview of this communication mechanism. Communications in SOC are asynchronous: each process has a set of FIFO input queues that act as buffers, managed by its enclosing service. Each queue is equipped with some data, here called *correlation key*, that can be used to distinguish (identify) the queue among the many present inside a service. When a service receives a message from the network, it inspects the content of the message for a portion of data that matches the correlation key of one of its queues. If a queue can be found, the message is inserted at the end of it. The process owning the queue will be able to consume the message later on in its execution. Thus, when a sender process p sends a message to a receiver process q, it needs to know *i*) the location of the service where q is running and *ii*) the correlation key of one of the queues owned by q. In practice, the correlation key in the message can be a part of the message payload itself or be in some separate headers (in this work we abstract from such details). Below, we formalise the semantics of AC by equipping choreographies with *deployments*, ranged over by D. We use deployments to formalise the elements of SOC that we have just informally described, in particular the *state* and *message queues* of processes located at services. We define each element separately.

Data and Process state. Data in SOC is typically structured following a tree-like format, e.g., XML [27] and JSON [103]. In this work, we use trees to represent both messages and the state of running processes (as in [98]). Formally, we consider a set $\mathcal{T}$ of rooted trees, ranged over by t, where edges are labelled by names, ranged over by $\underline{\mathrm{x}}, \underline{\mathrm{y}}, \cdots$. We assume that all outgoing edges of a node have distinct labels and that only leafs contain values, which can be either a location l or some basic data (integer, string, etc.). Variables in AC, ranged over by x, are formalised as paths to traverse a tree: $x, y, z \ ::= \ \underline{\mathrm{x}}.x \mid \varepsilon$. ε is the empty path; we often omit the tailing ε in paths. Given a path x and a nonempty tree t, we denote by $x(t)$ the node reached following the path x in t. Observe that $x(t)$ is partially defined since the path x may not be valid in t in general. By a slight abuse of notation, when $x(t)$ is a leaf we denote by $x(t)$ also the value of the node. In our semantics, we will also use the replacement operator $t \lhd (\, x, t' \,)$. If $x(t)$ is defined , $t \lhd (\, x, t' \,)$ returns the tree obtained replacing in t the subtree rooted in $x(t)$ by t'. If $x(t)$ is undefined, $t \lhd (\, x, t' \,)$ adds the smallest chain of empty nodes to t such that $x(t)$ is defined and inserts t'.

Deployment. A deployment D is an overloaded partial function on locations and processes. Intuitively, we map locations to the set of processes running at that location, and we map processes to their local state and input queues. Therefore, given a location l, we read $D(l) = \{\tilde{\mathsf{p}}\}$ as "the processes $\tilde{\mathsf{p}}$ are running at the location l" (for any process p and deployment D, we assume that each process p is always located at only one location). For processes, instead, given a process p then $D(\mathsf{p})$ returns a pair (t, M), where t is a tree representing the local state of p and M is a *queue map*. A queue map defines the input queues that p can use to receive messages from other processes. Formally, a queue map M is a partial function of type $M : \mathcal{T} \rightharpoonup \mathbf{Seq}(\mathcal{O} \times \mathcal{T})$[2]. A map $M(t_c) = \widetilde{(o, t)}$ means that the process owning M has an input queue containing $\widetilde{(o, t)}$ (a sequence of messages[3]) that *correlates* with the data t_c, called the correlation key of the queue. In our semantics, we use correlation keys to formalise our mechanism of message correlation. When receiving a message with some correlation data, our semantics places the message in the queue pointed by the correlation key that matches the correlation data of the message. In a message (o, t), o is the operation used by the sender and t is the payload of the message. In the remainder, for $D(\mathsf{p}) = (t, M)$, we use the shortcuts $D(\mathsf{p}).\mathsf{st}$ to refer to t (the state of p) and $D(\mathsf{p}).\mathsf{que}$ to refer to M (the queues of p), respectively.

Observe that, in our model, programmers do not need to specify the deployment of a choreography[4]. Concretely, for any choreography program C with no free session names (all sessions are under a start term), we can define a *default deployment* where all active processes are assigned to some location and given an empty state and queue map. Below, we denote the set of free process names in a choreography C with $\mathsf{fp}(C)$, and we write $t_\perp$ for the empty tree.

[2]$\mathbf{Seq}(\_)$ is the type constructor of sequences.

[3]We denote the empty list ε and a list of messages $(o_1, t_1) :: \cdots :: (o_n, t_n)$. We use $::$ rather than commas for a clearer definition of the semantics of AC.

[4]As in other languages states of programs are not specified by programmers.

Definition 15 (Default Deployment)**.** *Let C be a choreography with no free session names. Then, D is a default deployment for C if for all* $\mathsf{p} \in \mathsf{fp}(C) : D(\mathsf{p}) = (t_\perp, \emptyset)$ *and* $\mathsf{p} \in D(l)$ *for some l.*

Reductions. We present the semantics for AC in terms of reductions of the form $D, C \to D', C'$, where D, C is a *running choreography*. Formally, the reduction relation $\to$ is the smallest closed under the rules reported in Figure 6.3. In the rules for session creation and communications, we make use of the auxiliary relation $D, \delta \blacktriangleright D'$ to model how effect δ changes a deployment D into D'. δ ranges over

$$\delta ::= \textbf{start}\ k : \ \widetilde{l.\mathsf{p}[\mathtt{A}]} \ \mid \ k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o \ \mid \ k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].o(\underline{\mathbf{x}})$$

denoting, from left to right, the start of a session and sending and reception of a message. Below we separately discuss our main rules for $D, C \to D', C'$, along with the definition of $D, \delta \blacktriangleright D'$ for each δ. We report the semantics of effects in Figure 6.4.

Rule $\lfloor^{\mathrm{C}}|_{\textbf{Start}}\rceil$. Rule $\lfloor^{\mathrm{C}}|_{\mathrm{Start}}\rceil$ starts a fresh (denoted by $\#$[5]) session k' with a complete action, together with some fresh processes $\tilde{\mathsf{r}}$ (k' and $\tilde{\mathsf{r}}$ replace respectively k and $\tilde{\mathsf{q}}$ in the continuation). Intuitively, in the premise $D, \delta \blacktriangleright D'$ for $\delta = \textbf{start}\ k : \ \widetilde{l.\mathsf{p}[\mathtt{A}]}$, we obtain D' from D adding the needed information to support the newly created session in the continuation C'. In particular, we need to add information about the location of each new process and to set up the state and queue maps of each process involved in the new session to enable communications via message correlation. We formalise this requirement in the predicate of *session support*, sup:

Definition 16 (Session Support (sup))**.** *We say that $(t, \{M_\mathtt{A}\}_{\mathtt{A} \in \tilde{\mathtt{A}}})$ is a session support for a deployment D and some located roles $\widetilde{l.\mathtt{A}}$, denoted by the predicate* $\mathsf{sup}(t, \{M_\mathtt{A}\}_{\mathtt{A} \in \tilde{\mathtt{A}}}, D, \widetilde{l.\mathtt{A}})$*, if and only if:*

- (Locations) *For all $l.\mathtt{A} \in \widetilde{l.\mathtt{A}}$, $\underline{\mathbf{A.l}}(t) = l$.*
- (Correlation Keys) *For all pairwise-distinct $l.\mathtt{A}, l'.\mathtt{B}$ in $\widetilde{l.\mathtt{A}}$, $\underline{\mathbf{A.B}}(t) = t_c$ and $M_\mathtt{B}(t_c) = \varepsilon$ for some t_c such that $t_c \notin \mathsf{dom}(D(\mathsf{s}).\mathsf{que})$ for all $\mathsf{s} \in D(l')$.*

The predicate $\mathsf{sup}(t, \{M_\mathtt{A}\}_{\mathtt{A} \in \tilde{\mathtt{A}}}, D, \ \widetilde{l.\mathtt{A}})$ holds if the tree t, called *session descriptor*, contains the locations of the processes playing the roles in the session and the correlation keys that they use. Note that in t we use roles (which are static

[5]We let $\#$ informal for a simpler presentation, yet it is easy to formalise freshness of names on D.

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o \quad D, \eta \blacktriangleright D'}{D,\ \eta; C \quad \to \quad D',\ C} \ \lfloor^{\mathrm{C}}|_{\textsc{Send}}\rceil$$

$$\frac{j \in I \quad D, k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o_j(x_j) \blacktriangleright D'}{D,\ k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \quad \to \quad D',\ C_j} \ \lfloor^{\mathrm{C}}|_{\textsc{Recv}}\rceil$$

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x) \quad D, k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o \blacktriangleright D'}{D,\ \eta; C \quad \to \quad D',\ k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C} \ \lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil$$

$$\frac{i = 1 \text{ if } \mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{ true, } i = 2 \text{ otherwise}}{D,\ \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \quad \to \quad D,\ C_i} \ \lfloor^{\mathrm{C}}|_{\textsc{Cond}}\rceil$$

$$\frac{D, C_1 \to D', C_1'}{D, \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \quad \to \quad D', \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'} \ \lfloor^{\mathrm{C}}|_{\textsc{Ctx}}\rceil$$

$$\frac{\mathcal{R} \in \{\equiv, \simeq_{\mathsf{C}}\} \quad C_1 \mathcal{R} C_1' \quad D, C_1' \to D', C_2' \quad C_2' \mathcal{R} C_2}{D, C_1 \quad \to \quad D', C_2} \ \lfloor^{\mathrm{C}}|_{\textsc{Eq}}\rceil$$

$$\frac{D, C_1 \quad \to \quad D', C_1'}{D, C_1 \mid C_2 \quad \to \quad D', C_1' \mid C_2} \ \lfloor^{\mathrm{C}}|_{\textsc{Par}}\rceil$$

$$\frac{\#\tilde{\mathsf{r}} \quad \#k' \quad \mathsf{p} \in D(l) \quad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l.\mathsf{r}[\mathtt{B}]} \quad D, \delta \blacktriangleright D'}{D,\ \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C \quad \to \quad D',\ C[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]} \ \lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$$

$$\frac{\begin{array}{c} i \in \{1, \ldots, n\} \quad \#k' \\ \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{\widetilde{l_i.\mathtt{B}_i}\} \quad \#\tilde{\mathsf{r}} \quad \{\tilde{\mathsf{r}}\} = \bigcup_i \{\tilde{\mathsf{r}}_i\} \quad \mathsf{p} \in D(l) \\ \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], l_1.\mathsf{r}_1[\mathtt{B}_1], \ldots, l_n.\mathsf{r}_n[\mathtt{B}_n] \quad D, \delta \blacktriangleright D' \end{array}}{\begin{array}{c} D, \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C \mid \prod_i \left(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\right) \quad \to \\ D',\ C[k'/k] \mid \prod_i \left(\ C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\ \right) \mid \prod_i \left(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\right) \end{array}} \ \lfloor^{\mathrm{C}}|_{\textsc{PStart}}\rceil$$

Figure 6.3: Choreography calculus, semantics.

$$\frac{\begin{array}{c} l \in \underline{\mathbf{k.B.l}}(D(\mathsf{p}).\mathsf{st}) \qquad \mathsf{q} \in D(l) \\ t_c = \underline{\mathbf{k.A.B}}(D(\mathsf{p}).\mathsf{st}) \qquad D(\mathsf{q}).\mathsf{que}(t_c) = \tilde{m} \qquad t_m = \mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) \end{array}}{D, k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o \blacktriangleright D\big[\mathsf{q} \mapsto \big(D(\mathsf{q}).\mathsf{st}, D(\mathsf{q}).\mathsf{que}[t_c \mapsto \tilde{m} :: (o, t_m)]\big)\big]} \ \lfloor^{\mathrm{D}}|_{\textsc{Send}}\rceil$$

$$\frac{t_c = \underline{\mathbf{k.A.B}}(D(\mathsf{q}).\mathsf{st}) \quad D(\mathsf{q}).\mathsf{que}(t_c) = (o, t_m) :: \tilde{m}}{D, k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].o(x) \blacktriangleright D\big[\mathsf{q} \mapsto \big(D(\mathsf{q}).\mathsf{st} \lhd (\, x, t_m \,), D(\mathsf{q}).\mathsf{que}[t_c \mapsto \tilde{m}]\big)\big]} \ \lfloor^{\mathrm{D}}|_{\textsc{Recv}}\rceil$$

$$\frac{\begin{array}{c} \mathsf{sup}(t, \{M_{\mathtt{C}}\}_{\mathtt{C} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}}, D, (l.\mathtt{A}, \widetilde{l.\mathtt{B}})) \quad l'.\mathsf{q}[\mathtt{B}] \in \widetilde{l.\mathsf{q}[\mathtt{B}]} \quad t_{\mathsf{p}} = D(\mathsf{p}).\mathsf{st} \quad M_{\mathsf{p}} = D(\mathsf{p}).\mathsf{que} \\ D' = D\ \ [l' \mapsto D(l') \cup \{\mathsf{q}\}]\big[\mathsf{p} \mapsto \big(t_{\mathsf{p}} \lhd (\, \underline{\mathbf{k}}, t\,), M_{\mathsf{p}} \cup M_{\mathtt{A}}\big)\big]\big[\mathsf{q} \mapsto \big(t_{\perp} \lhd (\, \underline{\mathbf{k}}, t\,), M_{\mathtt{B}}\big)\big] \end{array}}{D, \mathbf{start}\, k : l.\mathsf{p}[\mathtt{A}], \widetilde{l.\mathsf{q}[\mathtt{B}]} \blacktriangleright D'} \ \lfloor^{\mathrm{D}}|_{\textsc{Start}}\rceil$$

Figure 6.4: Semantics of effects on D

names) instead of process identifiers because in the message correlation mechanism found in SOC, a sender does not know the process identifier of the intended receiver (this will be reflected in the development of our target language, DCC, in § 6.5). The set $\{M_{\mathtt{A}}\}_{\mathtt{A} \in \tilde{\mathtt{A}}}$ contains the respective queue map $M_{\mathtt{A}}$ for each role $\mathtt{A}$. $M_{\mathtt{A}}$ must contain an empty queue correlating with the corresponding key in t for each other role in the session, e.g., if role $\mathtt{A}$ receives from $\mathtt{B}$, $M_{\mathtt{A}}$ contains an empty queue correlating with the key at $\underline{\mathbf{B.A}}$ in t. Thus, each role has a queue to receive messages from each other role, as in standard multiparty session types with session-role channels [104]. Correlation keys must be fresh within the location of the receiver[6].

Rule $\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$ applies if Rule $\lfloor^{\mathrm{D}}|_{\textsc{Start}}\rceil$ applies. In $\lfloor^{\mathrm{D}}|_{\textsc{Start}}\rceil$, we choose in a non-deterministic way the session descriptor t and the queue maps $\{M_{\mathtt{C}}\}_{\mathtt{C} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}}$ under the only requirement that they respect predicate sup. This requirement makes our semantics very general as it supports potentially many session descriptors and therefore different implementations (e.g., based on cookies, random sets of data, API keys, etc.), as long as they comply with our definition of session support (see § 7.2). In $\lfloor^{\mathrm{D}}|_{\textsc{Start}}\rceil$, according to the previous definition, we copy the session descriptor under $\underline{\mathbf{k}}$, the path named after the session. This eases the access to the structure, since all in-session interactions happen under a certain session name k, which is globally fresh and therefore cannot clash with other pre-existing structures.

Remark 5. *In $\lfloor^{D}|_{\textsc{Start}}\rceil$ the session descriptor assigned to each process is not minimal, indeed it contains unused data like correlation keys used by other processes.*

[6]In AC, like in SOC, the sender delivers its message in the queue correlating with the key of the message and owned by a process located at the location of the receiver. To make correlation deterministic we forbid duplicate keys within a location.

However, since it does not alter our result, we favoured this simpler definition.

Rule $\lfloor^{\mathsf{C}}|_{\textbf{SEND}}\rceil$. The rule describes a partial sending and, as shown in Figs. 6.3 and 6.4, it updates the deployment with Rule $\lfloor^{\mathsf{D}}|_{\text{SEND}}\rceil$ to account for asynchronous message passing. We comment the conditions of $\lfloor^{\mathsf{D}}|_{\text{SEND}}\rceil$, referring paths as applied on the state of the sender p. The first four conditions (from top left to down right) guess the receiving process q, which is that process located at l — as retrieved under $\underline{\mathbf{k.B.l}}$ — and that owns the queue correlating with the key at $\underline{\mathbf{k.A.B}}$. This models real-world message correlation, where the sender guesses the receiver from its location and the correlation key of the message. $t_m = \mathsf{eval}(e,\ D(\mathsf{p}).\mathsf{st})$ is the content of the message, result of the evaluation of expression e on the state of p. $\lfloor^{\mathsf{D}}|_{\text{SEND}}\rceil$ modifies D such that q stores in the queue correlating with the key at $\underline{\mathbf{k.A.B}}$ the message (o, t_m).

Rule $\lfloor^{\mathsf{C}}|_{\textbf{RECV}}\rceil$. Rule $\lfloor^{\mathsf{C}}|_{\text{RECV}}\rceil$ implements a partial reception, updating the deployment with Rule $\lfloor^{\mathsf{D}}|_{\text{RECV}}\rceil$. In particular, $\lfloor^{\mathsf{C}}|_{\text{RECV}}\rceil$ records on which of the available operations $(o_i, i \in I)$ q received a message from the process playing role $\mathtt{A}$. In $\lfloor^{\mathsf{D}}|_{\text{RECV}}\rceil$, the first condition finds the queue that q, playing $\mathtt{B}$, uses to receive from $\mathtt{A}$. The second one provides a new deployment D' if the head of the queue has a message on operation o. If it does, D' removes the message from the head of the queue and copies in the state of q, under path x, the content of the message t_m.

Other rules. Rule $\lfloor^{\mathsf{C}}|_{\text{COM}}\rceil$ describes a complete communication. Intuitively, the reduction of a complete communication *i)* has the same effect on D as a $(send)$ term and *ii)* it reduces the (com) to a $(recv)$ on the same operation. Thanks to deployments and effects, we could model complete asynchronous communications in a fairly simple way wrt previous approaches (see § 7.2). Formally, the effect of Rule $\lfloor^{\mathsf{C}}|_{\text{COM}}\rceil$ on D is $\delta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o$ which applies Rule $\lfloor^{\mathsf{D}}|_{\text{SEND}}\rceil$. In the reductum, the (com) term reduces to a partial reception on operation o. Rules $\lfloor^{\mathsf{C}}|_{\text{COND}}\rceil$, $\lfloor^{\mathsf{C}}|_{\text{CTX}}\rceil$, and $\lfloor^{\mathsf{C}}|_{\text{PAR}}\rceil$ are standard, while rule $\lfloor^{\mathsf{C}}|_{\text{EQ}}\rceil$ accounts for the structural congruence $\equiv$ and the swapping relation $\simeq_{\mathsf{C}}$. The structural congruence, defined as the smallest congruence supporting α-conversion and satisfying the rules below

$$\begin{array}{cc} C \mid C' \equiv C' \mid C & (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3) \\ \mathsf{def}\ X = C'\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} & \mathsf{def}\ X = C'\ \mathsf{in}\ C[X] \equiv \mathsf{def}\ X = C'\ \mathsf{in}\ C[C'] \end{array}$$

abstracts from purely syntactic differences in processes and treats recursion in a standard way. As in [13], the swap relation $\simeq_{\mathsf{C}}$ exchanges the order of some actions. This enables more interleaving among processes. We report the rules of

$C_c =$
```
1.  req $k_d$ : c[C] <=> $l_A$.A, $l_{DM}$.DM, $l_L$.L;
2.  $k_d$ : c[C].mkReq() -> A.$get$;
3.  def STORE =
4.     $k_d$ : DM -> c[C].{
5.        $pkt(\underline{\mathbf{bb}})$;
6.           $k_s$ : c[U].$\underline{\mathbf{bb}}$ -> f[F].$append(\underline{\mathbf{data}})$;
7.           STORE ,
8.        $chksum(\underline{\mathbf{cs}})$;
9.           $k_s$ : c[U].$\underline{\mathbf{cs}}$ -> f[F].$check(\underline{\mathbf{cs}})$;
10.          if f.check($\underline{\mathbf{fname}}, \underline{\mathbf{cs}}$) {
11.             $k_s$ : f[F] -> c[U].$saved$
12.          } else {
13.             $k_s$ : f[F] -> c[U].$discarded$
14.    }}
15.  in
16. $k_d$ : A -> c[C].{
17.    $ok$;
18.       start $k_s$ : c[U] <=> $l_C$.f[F];
19.       $k_s$ : c[U].name() -> f[F].$create(\underline{\mathbf{fname}})$;
20.       STORE ,
21.    $ko$ }
```

$C_s =$
```
1.  acc $k_d$ : $l_A$.a[A], $l_{DM}$.dm[DM], $l_L$.l[L];
2.  $k_d$ : C -> a[A].$get(\underline{\mathbf{req}})$;
3.  if a.$isValid(\underline{\mathbf{req}})$ {
4.     $k_d$ : a[A].$\underline{\mathbf{req.rsc}}$ -> dm[DM].$ok(\underline{\mathbf{rsc}})$;
5.     $k_d$ : a[A] -> C.$ok$;
6.     $k_d$ : dm[DM].logok($\underline{\mathbf{rsc}}$) -> l[L].$log(\underline{\mathbf{log}})$;
7.     def TRANSFER =
8.        if dm.more($\underline{\mathbf{rsc}}$) {
9.           $k_d$ : dm[DM].next($\underline{\mathbf{rsc}}$) -> C.$pkt$;
10.          TRANSFER
11.       } else {
12.          $k_d$ : dm[DM].chksum($\underline{\mathbf{rsc}}$) -> C.$chksum$
13.       }
14.     in
15.     TRANSFER
16. } else {
17.    $k_d$ : a[A].$\underline{\mathbf{req.rsc}}$ -> dm[DM].$ko(\underline{\mathbf{rsc}})$;
18.    $k_d$ : a[A] -> C.$ko$;
19.    $k_d$ : dm[DM].logko($\underline{\mathbf{rsc}}$) -> l[L].$log(\underline{\mathbf{log}})$
20. }
```

Figure 6.5: Choreography Example

the swap relation in Appendix D.1. As an example, consider the Rule below that swaps η and η' if they share no processes (returned by $\mathsf{pn}(\eta)$).

$$\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset \quad \Rightarrow \quad \eta; \eta' \simeq_{\mathsf{C}} \eta'; \eta \qquad \lfloor^{\text{CS}}|_{\text{EtaEta}}\rceil$$

Swapping terms means that, although a choreography defines the global order in which its processes send and receive messages, this order can change at runtime. Despite the change, we guarantee to preserve the order of messages between each couple of processes in a session. Rule $\lfloor^{\text{C}}|_{\text{PStart}}\rceil$ starts a new session synchronising a partial choreography that requests to start a session with other choreographies that can accept the request. The premise of the rule $\{\widetilde{l.\text{B}}\} = \biguplus_i \{\widetilde{l_i.\text{B}_i}\}$ ($\biguplus$ being the union of the disjoint lists of located roles) requires that, in the accepting choreographies, the list of locations and their supported roles match the corresponding list of the request. The accepting choreographies remain available afterwards, for reuse. The rest of the rule is similar to $\lfloor^{\text{C}}|_{\text{Start}}\rceil$.

6.2.3 An example

We show an example of a system for verified file transfer written in AC. We use the example in § 6.6 to illustrate our compilation.

Verified file transfer. The program *i*) validates the file request of a Client on a Server, *ii*) transfers a file in multiple parts, and *iii*) verifies the transfer with a checksum. The Server logs all requests. We report in Figure 6.5 the code of program C, parallel composition of the two partial choreographies $C = C_c \mid C_s$. C_c and C_s respectively define the client- and the server-side code.

In C, process c — the only active process — plays the role of the Client C. In C_s, a plays the Access Manager (A), dm the Download Manager (DM) and l the Logger (L). In C_c (Line 18 of C_c) process f plays F, accessing the file system. We locate A, DM, L, and F at respectively $l_{\mathtt{A}}$, $l_{\mathtt{DM}}$, $l_{\mathtt{L}}$, and $l_{\mathtt{C}}$.

Lines 1 of C_c and C_s start session k_d between c, a, dm, and l. At Lines 2 of C_c and C_s, c makes a request for a file to a. Lines 4-15 of C_s define the outcome of a valid request (Line 3). Following the Lines, a forwards the resource request to dm and confirms (on ok) to c the request. dm asks l to log the request. Lines 7-14 define the recursive procedure TRANSFER, called at Line 15, for the multipart file download. If dm has more packets of the resource, it sends the next to c on operation pkt and TRANSFER repeats. Else dm ends k_d sending to c the checksum of the file. Lines 17-19 of C_s specify the outcome of an invalid request: a notifies dm and c of the failed attempt (ko) and dm asks to log the event to l.

Observe that C_c defines at Lines 3-15 a procedure to STORE the file. At Line 18 of C_c, after the approval, c (playing user U) starts a new session k_s with f, asking f to create a file to store the incoming packets. Then, in STORE, if c receives a new packet of bytes from dm (Line 5), it asks f to append them to the local file and STORE repeats. Else c receives the checksum from dm (Line 8) and c forwards it to f. Finally, f notifies c whether it $saved$ or $discarded$ the file wrt the checksum.

Remark 6. *To illustrate compositionality of ACs, we (logically) divided the example into two choreographies, (resp. the client- and server-side described above). However, we could specify the system as single choreography, e.g.,*

1. ***start*** k_d : c[C] <=> $l_{\mathtt{A}}$.a[A], $l_{\mathtt{DM}}$.dm[DM], $l_{\mathtt{L}}$.l[L]; 2. k_d : c[C].mkReq() -> a[A].get($\underline{\mathbf{req}}$); ...

We comment some features of AC programs in the example.

Session Descriptors. At Lines 1 of C_c and C_s c requests to start a new session k_d with some service processes a, dm, and l. We describe the structure of the session descriptor, stored in the state of the mentioned processes under path $\underline{\mathbf{kd}}$.

We report in Figure 6.6 such structure. Following the path $\underline{\mathbf{A}}$, we find a subtree with *i*) a node $\underline{\mathbf{l}}$ that stores the location $l_{\mathtt{A}}$ of a and *ii*) a set of nodes, named after all the other roles in k_d, leading to subtrees containing the correlation keys related to the roles in the paths. For example, the path $\underline{\mathbf{A.C}}$ points the tree $t_{\mathtt{AC}}$, which contains the correlation key used by a to send messages to c.

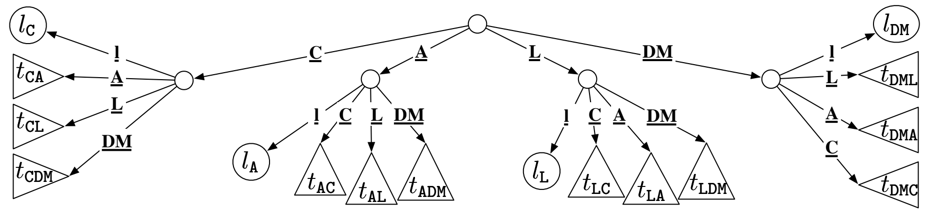


Figure 6.6: Example of structure of Session Descriptor

Parallelism. Lines 5-6 of C_s show how Rule $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$ can exchange the order of execution of actions. The actions at Lines 5 and 6 regard different processes (resp. a, s, and l) and the two instructions can swap along Rule $\lfloor^{\text{CS}}|_{\text{ETAETA}}\rceil$ of the swap relation. A possible reduction of C, starting from Line 5 of C_s, can apply Rule $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$ to swap Lines 5 and 6, reduce Line 6 with Rule $\lfloor^{\text{C}}|_{\text{COM}}\rceil$ to a partial reception for process l, and swap back the two Lines. Next, either a delivers its message with $\lfloor^{\text{C}}|_{\text{SEND}}\rceil$ or $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$ applies, letting l consume its message. Observe that the swap is non-deterministic, allowing other possible executions.

Asynchrony. AC supports asynchronous communication with queues, however, we need the swap relation to let a process send or receive a message asynchronously. Consider Lines 4-5 of C_s. We can apply $\lfloor^{\text{C}}|_{\text{COM}}\rceil$ on Line 4 and let a send its message to dm. This reduces Line 4 to a partial reception on dm. Then, we can apply $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$, swap the redex of Line 4 with Line 5, and let Line 5 execute with $\lfloor^{\text{C}}|_{\text{SEND}}\rceil$. Finally, we let dm consume its message with $\lfloor^{\text{C}}|_{\text{RECV}}\rceil$ on Line 4.

6.3 Typing

In this section we define our typing discipline for Applied Choreographies. Our typing checks the behaviour of sessions against protocols, given as multiparty session types [43, 105]. The main novelty of our type system is checking that the evolution of a deployment (states for message correlation and queues for asynchronous messaging) correctly implements the sessions described in a program, ensuring absence of errors such as deadlocks. We detail this part in § 6.3.3.

6.3.1 Types and Type Projection

Global and Local types. As in standard multiparty session types, we use *global types* to represent protocols from a global viewpoint and *local types* to describe the behaviour of each participant. Our type system checks that the local types, that abstract the behaviour of processes in a choreography, coherently follow a global type. We report below the syntax of global types G and local types T.

$$\begin{array}{rll} G ::= & \mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\}_i \\ \mid & \mathsf{rec}\ \mathbf{t}; G \\ \mid & \mathbf{t} \\ \mid & \mathsf{end} \end{array} \qquad \begin{array}{rll} T ::= & \alpha.\{o_i(U_i); T_i\}_i \\ \mid & \mathsf{rec}\ \mathbf{t}; T \\ \mid & \mathbf{t} \\ \mid & \mathsf{end} \end{array}$$

$$\alpha ::= !\mathtt{A} \mid ?\mathtt{B} \qquad U ::= S\{\underline{\mathbf{x}}_i : U_i\}_i \qquad S ::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{str} \mid \ldots$$

A global type $\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\}_i$ abstracts a communication, where A can send to B a message on any of the operations o_i and continue with the respective continuation G_i. A carried type U types the tree value exchanged in the message. A tree type $S\{\underline{\mathbf{x}}_i : U_i\}_i$ abstracts a tree with root value S (a basic type) and subtrees reachable from the root node by following $\underline{\mathbf{x}}_i$ with respective types U_i (our notation recalls that for record types in [106]). In local types, $!\mathtt{A}.\{o_i(U_i); T_i\}_i$ abstracts the sending of a message of type U_i to role A on one of the operations o_i, with continuation T_i. Dually, $?\mathtt{A}.\{o_i(U_i); T_i\}_i$ abstracts the offering of an input choice for all the operations o_i, with continuation T_i. Other terms for recursion and termination (end) are standard.

Type Projection. To relate global types to the behaviour of endpoints, we project a global type G onto the local type of a single role. We report in Figure 6.7 the projection of global types, defined following [14]. $[\![G]\!]_{\mathtt{A}}$ denotes the projection of G onto the role A. Intuitively, $[\![G]\!]_{\mathtt{A}}$ gives an encoding of the local actions expected by role A in the global type G. When projecting a communication we require the local behaviour of all roles not involved in it to be merged with the merging operator $\sqcup$. Like in [14] $T \sqcup T'$ is isomorphic to T and T' up to branching, where all branches of T or T' with distinct operations are also included.

6.3.2 Type checking

We present our system that guarantees that sessions follow their types.

Environments. We define our typing environments $\Gamma, \Gamma', \ldots$ as:

$$\begin{array}{rlll} \Gamma ::= & \Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle & (\textit{service typing}) & \mid\ \Gamma, \mathsf{p}.x : U \quad (\textit{variable typing}) \\ \mid & \Gamma, k[\mathtt{A}] : T & (\textit{local typing}) & \mid\ \Gamma, X : \Gamma \quad (\textit{procedure typing}) \\ \mid & \Gamma, \mathsf{p} : k[\mathtt{A}] & (\textit{ownership}) & \mid\ \emptyset \quad (\textit{empty env.}) \end{array}$$

A service typing $\tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$ types with G all sessions created by contacting the services at the locations $\tilde{l}$. We explain the role annotations: A is the role that the

$$[\![\mathsf{t}]\!]_{\mathtt{A}} = \mathsf{t} \qquad [\![\mathsf{end}]\!]_{\mathtt{A}} = \mathsf{end} \qquad [\![\mathsf{rec}\ \mathbf{t}; G]\!]_{\mathtt{A}} = \begin{cases} \mathsf{rec}\ \mathbf{t}; [\![G]\!]_{\mathtt{A}} & \text{if } \mathtt{A} \in G \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\}_i]\!]_{\mathtt{C}} = \begin{cases} !\mathtt{B}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{C}}\}_i & \text{if } \mathtt{C} = \mathtt{A} \\ ?\mathtt{A}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{C}}\}_i & \text{if } \mathtt{C} = \mathtt{B} \\ \bigsqcup_i [\![G_i]\!]_{\mathtt{A}} & \text{otherwise} \end{cases}$$

Figure 6.7: Choreography Calculus - Global Type Projection

active process (the starter) should play; roles $\tilde{\mathtt{B}}$ are the roles respectively played by each l in $\tilde{l}$ (each l plays one role, so the length of $\tilde{\mathtt{B}}$ is the same as the length of $\tilde{l}$); finally, $\tilde{\mathtt{C}}$ (where $\tilde{\mathtt{C}} \subseteq \tilde{\mathtt{B}}$) are the roles implemented by the choreography that we are typing. The annotation $\tilde{\mathtt{C}}$ enables composition of choreographies yet ensuring that only one choreography implements a specific role, as in [14]. When we write $\Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$, we always assume that: *i)* $\{\mathtt{A}, \tilde{\mathtt{B}}\} = \mathsf{roles}(G)$, where roles returns the set of roles in G; *ii)* the locations $\tilde{l}$ are ordered lexicographically; and, *iii)* the locations in l do not appear in any other service typing in Γ. A local typing $k[\mathtt{A}] : T$ states that role $\mathtt{A}$ in session k follows the local type T. We assume that roles in a session are typed by a single local typing, as in standard multiparty session types [43]. An ownership typing $\mathsf{p} : k[\mathtt{A}]$ states that process p owns the role $\mathtt{A}$ in session k: each process can participate in multiple sessions, but can play only one role in each of such sessions. Hence, a process p may appear in more than one ownership typings in a Γ, but never more than once per session. The other typings for variables and recursive procedures are standard.

Typing Judgements and Rules. A judgement $\Gamma \vdash C$ states that the choreography C follows the specifications given in Γ. We report in Figure 6.8 some select typing rules to derive valid typing judgements and comment the rules below. We report the full typing rules in Appendix D.2.

Rule $\lfloor^{\mathrm{T}}|_{\mathrm{START}}\rceil$ types a session start. In the first premise, the service typing $\tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle$ checks that the continuation implements all the roles in protocol G. The auxiliary function init[7] intuitively returns an environment containing all the ownerships and local typings to correctly type a freshly-started session. The type of each process is the local type projection of the global type G on the role

[7]Formally $\mathsf{init}(\,\widetilde{\mathsf{p}[\mathtt{A}]}, k, G\,) = \{\mathsf{q} : k[\mathtt{B}],\ k[\mathtt{B}] : [\![G]\!]_{\mathtt{B}} \mid \mathsf{q}[\mathtt{B}] \in \widetilde{\mathsf{p}[\mathtt{A}]}\}$.

$$\frac{j \in I \quad \Gamma \vdash \mathsf{p}\colon k[\mathtt{A}], \mathsf{q}\colon k[\mathtt{B}] \quad \Gamma \vdash \mathsf{p}.e\colon U_j \quad \Gamma, \mathsf{q}.x\colon U_j, k[\mathtt{A}]\colon T_j, k[\mathtt{B}]\colon T'_j \vdash C}{\Gamma, k[\mathtt{A}]\colon !\mathtt{B}.\{o_i(U_i); T_i\}_{i\in I}, k[\mathtt{B}]\colon ?\mathtt{A}.\{o_i(U_i); T'_i\}_{i\in I} \vdash k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o_j(x); C} \lfloor^{\mathrm{T}}|_{\textsc{Com}}\rceil$$

$$\frac{j \in I \quad \Gamma \vdash \mathsf{p}\colon k[\mathtt{A}] \quad \Gamma \vdash \mathsf{p}.e\colon U_j \quad \Gamma, k[\mathtt{A}]\colon T_j \vdash C}{\Gamma, k[\mathtt{A}]\colon !\mathtt{B}.\{o_i(U_i); T_i\}_{i\in I} \vdash k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o_j; C} \lfloor^{\mathrm{T}}|_{\textsc{Send}}\rceil$$

$$\frac{\Gamma \vdash \mathsf{q}\colon k[\mathtt{B}] \quad \forall j \in I.\, \Gamma, \mathsf{q}.x_j\colon U_j, k[\mathtt{B}]\colon T_j \vdash C_j}{\Gamma, k[\mathtt{B}]\colon ?\mathtt{A}.\{o_i(U_i); T_i\}_{i\in I} \vdash k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_j(x_j); C_j\}_{j\in I\cup J}} \lfloor^{\mathrm{T}}|_{\textsc{Recv}}\rceil$$

$$\frac{\Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \mathsf{init}\big(\widetilde{\mathsf{r}[\mathtt{C}]}, k, G\big) \vdash C \quad \widetilde{\mathsf{r}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C} \lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$$

$$\frac{\Gamma_i \vdash C_i \quad \Gamma_2 \vdash C_2}{\Gamma_1 \circ \Gamma_2 \vdash C_1 \mid C_2} \lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil \qquad \frac{\Gamma, \mathsf{p} : k[\mathtt{A}], k[\mathtt{A}] : [\![G]\!]_{\mathtt{A}} \vdash C \quad \Gamma \vdash \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle}{\Gamma \vdash \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C} \lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil$$

$$\frac{\mathsf{end}(\Gamma) \quad \Gamma'' \subseteq \Gamma'}{\Gamma, \Gamma'', X\langle\tilde{\mathsf{p}}\rangle : \Gamma' \vdash X\langle\tilde{\mathsf{p}}\rangle} \lfloor^{\mathrm{T}}|_{\textsc{Call}}\rceil \qquad \frac{\tilde{l} \subseteq \tilde{l}' \quad \Gamma, \tilde{l}' : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle, \mathsf{init}\big(\widetilde{\mathsf{q}[\mathtt{C}]}, k, G\big) \vdash C}{\Gamma, \tilde{l}' : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \vdash \textbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{C}]}; C} \lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil$$

Figure 6.8: Choreography Calculus - Typing Rules (selected)

owned by the process in the session. Similar to $\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$, Rule $\lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil$ performs the checks only for the process requesting the creation of a new session. Dually, $\lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil$ checks that the processes created after a request correctly implement their expected behaviour.

Rule $\lfloor^{\mathrm{T}}|_{\textsc{Com}}\rceil$ types a complete communication, checking that: *i*) the chosen operation o_j is among the ones that the sender can select according to its local type; *ii*) similarly, o_j is among the ones offered by the receiver according to its local type; *iii*) the sender and the receiver processes own their respective roles in the session; *iv*) the expression of the sender (e) has the type[8] U_j expected by the protocol; *v*) the receiver uses the reception variable accordingly in the continuation C; and *vi*) processes p and q proceed according to their respective types in Γ. Similar to rule $\lfloor^{\mathrm{T}}|_{\textsc{Com}}\rceil$, $\lfloor^{\mathrm{T}}|_{\textsc{Send}}\rceil$ and $\lfloor^{\mathrm{T}}|_{\textsc{Recv}}\rceil$ respectively check $(send)$ and $(recv)$ actions.

Rule $\lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil$ uses the *role distribution operator* $\Gamma_1 \circ \Gamma_2$, from [14], to merge service typings and to check that choreographies executing in parallel do not implement overlapping roles at locations. Formally, $\Gamma_1 \circ \Gamma_2$ is defined if Γ_1 and Γ_2 type different processes[9] and for $\Gamma_i = \Gamma'_i, \Gamma^l_i$, $i \in \{1, 2\}$ where Γ^l_i contains only

[8]The judgement $\vdash t : U$ reads as "tree t has type U".

[9]Formally, we can write $\Gamma_1 \circ \Gamma_2$ if $\mathsf{pn}(\Gamma_1) \cap \mathsf{pn}(\Gamma_2) = \emptyset$ for $\mathsf{pn}(\Gamma) = \{\mathsf{p} \mid \mathsf{p}\colon k[\mathtt{A}] \in \Gamma\}$.

service typings

$$\Gamma_1 \circ \Gamma_2 = \Gamma_1' \,,\, \Gamma_2' \,,\, \Gamma_1^l \circ \Gamma_2^l$$
$$\Gamma_1^l \circ \Gamma_2^l = \left\{ \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \mid \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{D}}\rangle \in \Gamma_1^l \ \wedge\ \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{E}}\rangle \in \Gamma_2^l \ \wedge\ \tilde{\mathtt{D}} \uplus \tilde{\mathtt{E}} = \tilde{\mathtt{C}} \right\}$$

All the other typing rules are standard.

6.3.3 Runtime Typing

To prove that well-typed AC programs never go wrong, we need to pay attention to how their deployments evolve at runtime. For example, in Rule $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$ the sender needs the necessary information in its state to "find" the receiver through correlation: a remarkable difference wrt previous works on choreographies, where such conditions do not exist and choreographies can always continue execution (see, e.g., [10, 107, 96, 13]). To address this issue, we extend our typing discipline to check runtime states.

Wrong Deployments. We need to prevent deployments from "going wrong" during execution. Intuitively, we say that a deployment is wrong wrt a choreography if any of these conditions holds:

- *(uninitialised variables)* processes have undefined variables;
- *(incompatible session descriptors)* processes in a session store different locations or keys for the same session descriptor;
- *(correlation race)* a correlation key is used for more than one queue at a location;
- *(protocol violations)* a message queue does not contain messages as expected by the protocol of the session in which it is used.

Wrong deployments may cause unpredictable executions or undesired behaviours, e.g., deadlocks. We illustrate the consequences of having wrong deployments with this simple running choreography: $D, k : \mathsf{p}[\mathtt{A}].\underline{\mathrm{y}} \texttt{ -> } \mathsf{q}[\mathtt{B}].\underline{\mathrm{x}}; \mathbf{0}$.

The only way this choreography can reduce is to apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$ in a reduction derivation. The second premise of Rule $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$ requires a deployment reduction of the form $D, k : \mathsf{p}[\mathtt{A}].\underline{\mathrm{y}} \texttt{ -> B} \blacktriangleright D'$ for some D'. Hence, the deployment D must respect the conditions given in Rule $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$. Let us see how a wrong deployment D may cause problems, following the above list:

- *(uninitialised variables)* Assume that D is such that $D(\mathsf{p}).\mathsf{st}$ is a tree with no node under path $\underline{\mathrm{y}}$; then the condition $\mathsf{eval}(\underline{\mathrm{y}}, D(\mathsf{p}).\mathsf{st})$ given in $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$ is undefined and $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$ cannot be applied, causing the choreography to get stuck.

- *(incompatible session descriptors)* Assume $\mathsf{q} \in D(l')$ for $l \neq l'$ and $\underline{\mathbf{k.B.l}}(D(\mathsf{p}).\mathsf{st}) = l$. Again, $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$ cannot apply and we have a deadlock, caused by the sender "pointing" at the wrong receiving service. A similar case happens if $t \notin \mathsf{dom}(D(\mathsf{q}).\mathsf{que})$ and $\underline{\mathbf{k.A.B}}(D(\mathsf{p}).\mathsf{st}) = t$ because we cannot find the queue of the addressee.
- *(correlation race)* Assume D is such that $D(l) = \{\mathsf{q}, \mathsf{r}\}$ (r and q are at the same location). Assume also $D(\mathsf{q}).\mathsf{que} = D(\mathsf{r}).\mathsf{que} = M$ for some M such that $\underline{\mathbf{k.A.B}}(D(\mathsf{p}).\mathsf{st}) \in M$, i.e., both q and r have a queue correlating with the key that p uses to send its message. Since $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$ guesses the receiver from its location and correlation key, p non-deterministically delivers its message to either q or r. In the second case, we get:

$$D, k : \mathsf{p}[\mathtt{A}].\underline{\mathbf{y}} \;\texttt{->}\; \mathsf{q}[\mathtt{B}].o(\underline{\mathbf{x}}); \mathbf{0} \;\rightarrow\; D', k : \mathtt{A} \;\texttt{->}\; \mathsf{q}[\mathtt{B}].\underline{\mathbf{x}}; \mathbf{0}$$

  where in D' the queue of r contains the message received from p. The choreography is now deadlocked because q cannot consume the expected message from p.
- *(protocol violations)* Assume that $D(\mathsf{q}).\mathsf{que} = M$ for some $M(\ \underline{\mathbf{k.A.B}}(D(\mathsf{p}).\mathsf{st})\ ) = (o', t')$ where $o \neq o'$, i.e., q has a message in the queue used by p. If we let the choreography reduce like at the previous point, it ends up deadlocked. After the reduction, the queue used by p contains in its head the message (o', t') and Rule $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ cannot apply as it expects to find a message for o at that position.

Below we extend our type system to prove that, given a well-typed choreography, our semantics never produces wrong deployments (provided that we do not start from a wrong deployment). Observe that this development is transparent to programmers, since default deployments are never wrong.

Runtime Global Types. We extend the syntax of global types to capture partial runtime states, following the idea presented in [108]:

$$G ::= \quad \cdots \quad | \quad \mathtt{A} \;\rightsquigarrow\; \mathtt{B}.o(U); G$$

where the new term $\mathtt{A} \;\rightsquigarrow\; \mathtt{B}.o(U)$ means that the sender $\mathtt{A}$ has sent the message but the receiver $\mathtt{B}$ has still to consume it.

Semantics of Global Types. To express the (abstract) execution of protocols, we give a semantics for global types. Formally, $G \to G'$ is the smallest relation on the recursion-unfolding of global types satisfying the rules below

$$\begin{array}{rl}
\lfloor{}^{\mathrm{G}}|_{\mathrm{SEND}}\rceil & j \in I \quad \Rightarrow \quad \mathtt{A} \text{ -> } \mathtt{B}.\{o_i(U_i); G_i\} \to \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(U_j); G_j \\
\lfloor{}^{\mathrm{G}}|_{\mathrm{RECV}}\rceil & \mathtt{A} \rightsquigarrow \mathtt{B}.o(U); G \to G \\
\lfloor{}^{\mathrm{G}}|_{\mathrm{REC}}\rceil & G[\mathsf{rec}\ \mathbf{t}; G/\mathbf{t}] \to G' \quad \Rightarrow \quad \mathsf{rec}\ \mathbf{t}; G \to G' \\
\lfloor{}^{\mathrm{G}}|_{\mathrm{SWAP}}\rceil & G_1 \simeq_{\mathsf{G}} G_2 \wedge G_2 \to G_2' \wedge G_2' \simeq_{\mathsf{G}} G_1' \quad \Rightarrow \quad G_1 \to G_1'
\end{array}$$

The rules are similar to those for AC. Rule $\lfloor{}^{\mathrm{G}}|_{\mathrm{SEND}}\rceil$ allows a sender role to proceed before the corresponding receiver has actually received the message. Based on the selected operation, e.g., o_j, the Rule reduces the type to a reception followed by the respective continuation (G_j). The reception is executed in Rule $\lfloor{}^{\mathrm{G}}|_{\mathrm{RECV}}\rceil$. In $\lfloor{}^{\mathrm{G}}|_{\mathrm{SWAP}}\rceil$, we model parallelism with the relation for global types $\simeq_{\mathsf{G}}$, which follows the same intuition of $\simeq_{\mathsf{C}}$. Swapping allows us to model asynchrony in global types as done in AC. We report the rules of $\simeq_{\mathsf{G}}$ in Appendix D.3. As an example, consider the swap between a reception and a complete communication:

$$\lfloor{}^{\mathrm{GS}}|_{\mathrm{COMRECV}}\rceil\ \{\mathtt{A},\mathtt{B}\} \cap \{\mathtt{D}\} = \emptyset \Rightarrow \begin{array}{l} \mathtt{A} \text{ -> } \mathtt{B}.\{o_i(U_i); \mathtt{C} \rightsquigarrow \mathtt{D}.o(U); G_i\} \\ \simeq_{\mathsf{G}}\ \mathtt{C} \rightsquigarrow \mathtt{D}.o(U); \mathtt{A} \text{ -> } \mathtt{B}.\{o_i(U_i); G_i\} \end{array}$$

Runtime Type checking and Typing Rules. We extend the typing rules given in the previous section to check runtime terms. The extension consists in *i*) new terms for Γ, and *ii*) the introduction of rule $\lfloor{}^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ to type runtime choreographies. We extend the grammar of typing environments Γ as follows:

$$\begin{array}{rll}
\Gamma ::= & \ldots & \\
| & \Gamma, \mathsf{p}@l & (\mathit{location}) \\
| & \Gamma, b[k]^{\mathtt{A}}_{\mathtt{B}} : T & (\mathit{buffer\ typing})
\end{array}$$

$\Gamma, \mathsf{p}@l$ states that process p runs at location l. A buffer typing $b[k]^{\mathtt{A}}_{\mathtt{B}} : T$ types the messages in the queue where the process implementing role $\mathtt{B}$ in session k receives messages from role $\mathtt{A}$. To relate the typings of queues (see Definition 18) to the buffer types expected by the protocol of sessions, we define the *buffer type projection* $[\![G]\!]^{\mathtt{A}}_{\mathtt{B}}$, which returns the expected buffer type of role $\mathtt{B}$ from $\mathtt{A}$ in G. $[\![G]\!]^{\mathtt{A}}_{\mathtt{B}}$ extracts from G the partial receptions of the form $\mathtt{A} \rightsquigarrow \mathtt{B}.o(U)$, translating it to a local type $?\mathtt{A}.o(U)$. Below we report the definition of $[\![G]\!]^{\mathtt{A}}_{\mathtt{B}}$ for the only interesting case of receptions. The full definition is reported in Appendix D.4.

$$[\![\mathtt{C} \rightsquigarrow \mathtt{D}.o(U); G]\!]^{\mathtt{A}}_{\mathtt{B}} = \begin{cases} ?\mathtt{A}.o(U); [\![G]\!]^{\mathtt{A}}_{\mathtt{B}} & \text{if } \mathtt{C} = \mathtt{A} \wedge \mathtt{D} = \mathtt{B} \\ [\![G]\!]^{\mathtt{A}}_{\mathtt{B}} & \text{otherwise} \end{cases}$$

We also extend type projection to handle receptions:

$$[\![\mathtt{A} \rightsquigarrow \mathtt{B}.o(U); G]\!]_{\mathtt{C}} = \begin{cases} ?\mathtt{A}.o(U); [\![G]\!]_{\mathtt{C}} & \text{if } \mathtt{C} = \mathtt{B} \\ [\![G]\!]_{\mathtt{C}} & \text{otherwise} \end{cases}$$

We now proceed defining the *partial coherence* predicate $\mathsf{pco}(\Gamma)$, which holds if and only if for all sessions k, the local and buffer typings of k follow (are projection of) the same global type G. The idea is that, since D is a global deployment, we can check for partial coherence of all sessions in Rule $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$.

Definition 17 (Partial Coherence)**.** *We write* $\mathsf{pco}(\Gamma)$ *when, for all sessions k in Γ, there exists a global type G such that,*

$$\forall\ k[\mathtt{B}]\colon T \in \Gamma \Rightarrow T = [\![G]\!]_{\mathtt{B}}\ \wedge\ \forall\ \mathtt{A} \in \mathsf{roles}(G)/\{\mathtt{B}\} \Rightarrow \Gamma \vdash b[k]^{\mathtt{A}}_{\mathtt{B}}\colon\ [\![G]\!]^{\mathtt{A}}_{\mathtt{B}}$$

Finally, we define the Rule to type a running choreography:

$$\frac{\mathsf{pco}(\Gamma) \quad \Gamma \vdash D \quad \Gamma \vdash C}{\Gamma \vdash D, C}\ \lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$$

A judgement $\Gamma \vdash D, C$ states that C and D are coherent according to Γ and all sessions in Γ are coherent. Γ is an abstraction between D and C and guarantees D to not go wrong. $\Gamma \vdash D$ checks D to be well-typed (not wrong) wrt to Γ. Formally:

Definition 18 (Deployment Judgements)**.**

$$\Gamma \vdash D \iff \begin{cases} \forall\, l \in D,\ \forall\, \{\mathsf{p},\mathsf{q}\} \subseteq D(l),\ \ \mathsf{dom}(D(\mathsf{p}).\mathsf{que})\ \cap\ \mathsf{dom}(D(\mathsf{q}).\mathsf{que}) = \emptyset \\ \forall\, \mathsf{p}.x\colon U \in \Gamma,\ \vdash x(\ D(\mathsf{p}).\mathsf{st}\ )\colon U \\ \forall\, \{\mathsf{p}\colon k[\mathtt{A}], \mathsf{q}\colon k[\mathtt{B}]\} \subseteq \Gamma,\ \ \underline{\mathbf{k}}(\ D(\mathsf{p}).\mathsf{st}\ ) = \underline{\mathbf{k}}(\ D(\mathsf{q}).\mathsf{st}\ ) \\ \forall\, \mathsf{p}\colon k[\mathtt{A}] \in \Gamma, \Gamma \vdash \mathsf{p}@l\ \wedge\ \mathsf{p} \in D(l)\ \wedge\ \underline{\mathbf{k.A.l}}(\ D(\mathsf{p}).\mathsf{st}\ ) = l \\ \forall\, \mathsf{p}\colon k[\mathtt{A}] \in \Gamma \wedge \forall\, b[k]^{\mathtt{B}}_{\mathtt{A}}\colon T \in \Gamma,\ \ \mathsf{bte}(\mathtt{B}, D(\mathsf{p}).\mathsf{que}(\underline{\mathbf{k.B.A}}(D(\mathsf{p}).\mathsf{st}))) = T \end{cases}$$

We comment, from top to bottom, the checks performed by $\Gamma \vdash D$:

- locations must have unique keys: for all locations in D and for all pairs of processes in that location, the queue maps of the two processes have no common correlation key (i.e., the domain of their queue maps are distinct).
- Γ and D must agree on the type of variables: for each typing $\mathsf{p}.x\colon U$ in Γ, D must associate x, in the state of process p, to a value of type U;
- session descriptors must match: for all couples of processes p and q, playing the respective roles $\mathtt{A}$ and $\mathtt{B}$ in a session k in Γ, the session descriptors for k stored by p and q are the same;
- Γ and D must agree on the location of all processes: for each process within a session k: *i*) its location according to Γ, *ii*) its location according to D, and *iii*) its location in the session descriptor of k must coincide;
- Γ and D must agree on the state of all queues: for each process p playing role $\mathtt{A}$ in a session k in Γ and for each role $\mathtt{B}$ such that the buffer type

$$
\begin{array}{l}
G_d = \texttt{C -> A}.\mathit{get}\left(\left\{\begin{array}{l}\underline{\textbf{creds}} : \textbf{str},\\ \underline{\textbf{rsc}} : \textbf{str}\end{array}\right\}\right);\\
\qquad \texttt{A -> DM}.\left\{\begin{array}{l}\mathit{ko}(\textbf{str}); \texttt{A -> C}.\mathit{ko}(); \texttt{DM -> L}.\mathit{log}(\textbf{str}),\\ \mathit{ok}(\textbf{str}); \texttt{A -> C}.\mathit{ok}(); \texttt{DM -> L}.\mathit{log}(\textbf{str}); \mathsf{rec}\ \mathbf{t}; \texttt{DM -> C}.\left\{\begin{array}{l}\mathit{pkt}(\textbf{bytes}); \mathbf{t},\\ \mathit{chksum}(\textbf{str})\end{array}\right\}\end{array}\right\}
\end{array}
$$

$$
G_s = \texttt{U -> F}.\mathit{create}(\textbf{str}); \mathsf{rec}\ \mathbf{s}; \texttt{U -> F}.\left\{\begin{array}{l}\mathit{append}(\textbf{bytes}); \mathbf{s},\\ \mathit{check}(\textbf{str}); \texttt{F -> U}.\left\{\begin{array}{l}\mathit{saved}(),\\ \mathit{discarded}()\end{array}\right\}\end{array}\right\}
$$

Figure 6.9: Protocols example

$b[k]^{\texttt{B}}_{\texttt{A}}$: $T \in \Gamma$, the extracted buffer type of $\underline{\textbf{k.A.B}}(D(\texttt{p}).\text{st})$ must be equal T. To check this last requirement, we define the *buffer typing extractor* bte($\texttt{B}, \tilde{m}$) which, given a queue $\tilde{m}$ and a sender role B, returns the buffer type of $\tilde{m}$[10].

Typing Example

We show, in Figure 6.9, how we can write the protocols that types the example choreographies in § 6.5. Notably, we define two independent protocols that are implemented, composed, and interleaved at runtime by the choreographies they type. The first protocol, called G_d, specifies the interaction of the system that allows a Client to authenticate on a Server, download a file from it, and verify its integrity, also logging all requests. The roles in the protocol are the same used in § 6.5, i.e., C for the Client, DM for the download manager, A for the access manager and L for the logger.

Following G_d, C asks to A to get a file with a message that must contain two values of type **str**: **creds** for the credentials of C and **rsc** for the resource identifier of the file (e.g., a URL). A notifies DM whether the credentials and the resource identifier are valid or not:

- If they are not valid, A forwards to DM the outcome of the validation (**str**) on operation ko. A also sends an empty message[11] to C on operation ko as response to the failed attempt. Finally, DM ends the branch forwarding to L, on log, the error received from A.

- If they are valid, A sends to DM the identifier of the file on operation ok. A confirms the request to C with an empty message on operation ok. DM asks

[10]Formally, let $\vdash t_1 : U_1, \cdots, \vdash t_n : U_n$, and $\tilde{m} = (o_1, t_1) :: \cdots :: (o_n, t_n)$ then bte($\texttt{A}, \tilde{m}$) $= ?\texttt{A}.o_1(U_1);\ \cdots\ ;\ ?\texttt{A}.o_n(U_n)$.

[11]empty parentheses are a short-cut for type **unit**

to log the request of the file (str) to L. The protocol continues with the definition of procedure rec t where DM sends to C either: *i*) a chunk (packet) of bytes of the file, on pkt, and repeats t, or *ii*) the checksum of the file, on $chksum$, if there are no more packets, ending the protocol.

The second protocol, G_s, defines the interaction between the User U of an operative system and a process F that accesses the filesystem. In G_s, U asks to F to $create$ a new file (str). Then the protocol defines the procedure rec s. U can either ask F to *i*) $append$ a new packet of bytes to the file, repeating s or *ii*) $check$ the file against its checksum (str). Based on the outcome of the checksum, F either notifies U that it $saved$ the file, or $discarded$ it.

6.3.4 Properties

We close this section with the main guarantees of our type system.

First, our semantics preserves well-typedness:

Theorem 3 (Subject Reduction)**.** $\Gamma \vdash D, C$ *and* $D, C \to D', C'$ *imply* $\Gamma' \vdash D', C'$ *for some* Γ'*.*

We now relate the behaviour of sessions in a well-typed choreography to their respective global types. We denote $[\![G]\!]_k$ the projection of a global type G for a session k and let $[\![G]\!]_k$ be the set of local and buffer typings as obtained by the projection of G on each of its roles:

Definition 19 (Global Type Projection)**.**

$$[\![G]\!]_k = \begin{array}{c} \{k[\mathtt{A}] \colon [\![G]\!]_{\mathtt{A}} \mid \mathtt{A} \in \mathsf{roles}(G)\}\,, \\ \{b[k]^{\mathtt{A}}_{\mathtt{B}} \colon [\![G]\!]^{\mathtt{A}}_{\mathtt{B}} \mid \mathtt{A} \in \mathsf{roles}(G), \mathtt{B} \in \mathsf{roles}(G)/\{\mathtt{A}\}\} \end{array}$$

We say that a reduction is "at session k" if it is obtained by consuming a communication term for session k (as in [43]), and we write $k \notin \Gamma$ when k does not appear in any local typing in Γ. Then we have:

Theorem 4 (Session Fidelity)**.** *Let* $\Gamma, \Gamma_k \vdash D, C$*,* $k \notin \Gamma$*. Then,* $D, C \to D', C'$ *with a redex at session* k *implies that, for some* G *and* Γ'*,* $k \notin \Gamma'$*, (i)* $\Gamma_k \subseteq [\![G]\!]_k$*, (ii)* $G \to G'$*, (iii)* $\Gamma'_k \subseteq [\![G']\!]_k$*, and (iv)* $\Gamma', \Gamma'_k \vdash D', C'$*.*

Theorem 4 states that all communications on sessions follow the expected protocols (Γ' may differ from Γ for the instantiation of a new variable).

We report the full proof of Theorem 3 and that of Theorem 4 (below) in Appendix E.1.

We can now present one of our major results: well-typed applied choreographies never deadlock when all the necessary participants are defined. Let the coherence predicate co be defined as follows:

Definition 20 (Coherence). $\mathsf{co}(\Gamma)$ *holds iff* $\forall\ k \in \Gamma\ ,\ \exists\ G$ *s.t.*

- $\tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \in \Gamma\ \wedge\ \tilde{\mathtt{C}} = \tilde{\mathtt{B}}$ *and*
- $\forall\ \mathtt{A} \in \mathsf{roles}(G)$, $k[\mathtt{A}] : T \in \Gamma\ \wedge\ T = [\![G]\!]_{\mathtt{A}}\ \wedge\ \forall\ \mathtt{B} \in \mathsf{roles}(G)/\{\mathtt{A}\},\ b[k]^{\mathtt{B}}_{\mathtt{A}} = [\![G]\!]^{\mathtt{B}}_{\mathtt{A}}$

Coherence extends partial coherence to check that *i*) all needed services to start new sessions are present and *ii*) all the roles in every open session are correctly implemented by some processes. Coherent and well-typed systems are deadlock-free:

Theorem 5 (Deadlock-freedom). $\Gamma \vdash D, C$ *and* $\mathsf{co}(\Gamma)$ *imply that either (i)* $C \equiv \mathbf{0}$ *or (ii) there exist* D' *and* C' *such that* $D, C \to D', C'$.

We report the full proof of Theorem 5 in Appendix E.2.

6.4 Endpoint Projection

We now present the Endpoint Projection and its properties. The EPP returns a correct composition[12] of endpoint choreographies that implements the behaviour of a given choreography. Intuitively, an endpoint (applied) choreography is an applied choreography that does not contain complete actions. Formally, let $\mathsf{fp}(C)$ return the set of free processes in a choreography C:

Definition 21 (Endpoint Choreography [14]). *We say that a choreography C is an endpoint choreography if C does not contain complete actions and one of the two following conditions holds:*

$$i)\ C = \textbf{\textit{acc}}\ k : l.\mathsf{q}[\mathtt{B}]; C' \wedge \{\mathsf{q}\} = \mathsf{fp}(C'); \qquad ii)\ \mathsf{fp}(C) = \{\mathsf{p}\}$$

An endpoint choreography defines the identity of only one process: either i) a service process or ii) an active process. Notably, our definition of EPP captures the description of the behaviour of single endpoints up to complete choreographies.

To give the definition of the EPP of a choreography, we first define the notion of *process projection*, which defines the behaviour of a single process p in a choreography C, written $[\![C]\!]_{\mathsf{p}}$. We report the formalisation of $[\![C]\!]_{\mathsf{p}}$ in Appendix D.6. Intuitively, the process projection follows the structure of the originating choreography. A $(start)$ projects to a (req) on the active process and a set of (acc) terms on the service processes. Likewise, a (com) projects to a partial $(send)$ for

[12]Here and in the following “composition” means “parallel composition”.

the sender and a partial $(recv)$ for the receiver. Any partial term is projected as it is for its respective process, following the structure of the choreography. We report below the projections of $(recv)$ and $(cond)$ terms which use the *merging* partial operator $\sqcup$ [96] (reported in Appendix D.7) to merge the behaviour of all processes in their branches.

$$\llbracket k : \mathsf{p[A]}.e \texttt{ -> } \mathsf{q[B]}.o(x); C \rrbracket_{\mathsf{r}} = \begin{cases} k : \mathsf{p[A]}.e \texttt{ -> } \mathsf{B}.o; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ k : \mathsf{A} \texttt{ -> } \mathsf{q[B]}.o(x); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \rrbracket_{\mathsf{r}} = \begin{cases} \mathsf{if}\ \mathsf{p}.e\ \{\llbracket C_1 \rrbracket_{\mathsf{r}}\}\ \mathsf{else}\ \{\llbracket C_2 \rrbracket_{\mathsf{r}}\} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C_1 \rrbracket_{\mathsf{r}} \sqcup \llbracket C_2 \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$C \sqcup C'$ is defined only for endpoint choreographies and returns an endpoint choreography isomorphic to C and C' up to branching, i.e., with all branches on distinct operations. We show the only special rule of $\sqcup$ for merging $(recv)$ terms.

$$k : \mathtt{A} \texttt{ -> } \mathsf{p[B]}. \{o_i(x_i); C_i\}_{i \in I} \ \sqcup\ k : \mathtt{A} \texttt{ -> } \mathsf{q[B]}. \left\{o_j(x_j); C'_j\right\}_{j \in J} =$$
$$k : \mathtt{A} \texttt{ -> } \mathsf{p[B]}. \left\{ \begin{array}{c} \{\ o_i(x_i); C_i\ \}_{i \in I/J} \quad \cup \quad \{\ o_i(x_i); C'_i\ \}_{i \in J/I} \\ \cup \quad \{o_i(x_i); C_i \sqcup C'_i\}_{i \in I \cap J} \end{array} \right\}$$

On the definition of process projection, we define the EPP of a whole system. Here, and in the following, $\lfloor C \rfloor_l$ is the grouping operator that returns the set of grouped (bound) service processes at the same location. We show only the rules (reported in full in Appendix D.8) of $\lfloor C \rfloor_l$ for $(start)$ and (acc) as the other rules are trivially recursive applications.

$$\left\lfloor \textbf{start}\ k : \mathsf{p[D]} \Leftrightarrow \widetilde{l.\mathsf{q[B]}}; C \right\rfloor_l = \left\lfloor \textbf{acc}\ k : \widetilde{l.\mathsf{q[B]}}; C \right\rfloor_l$$

$$\left\lfloor \textbf{acc}\ k : \widetilde{l.\mathsf{q[B]}}; C \right\rfloor_l = \begin{cases} \{\mathsf{r}\} \cup \lfloor C \rfloor_l & \text{if } l.\mathsf{r[A]} \in \widetilde{l.\mathsf{q[B]}} \\ \lfloor C \rfloor_l & \text{otherwise} \end{cases}$$

Definition 22 (Endpoint Projection)**.** *Let C be a term of AC. The endpoint projection of C, denoted by* $\llbracket C \rrbracket$*, is defined as:*

$$\llbracket C \rrbracket = \prod_{\mathsf{p}\, \in\, \mathsf{fp}(C)} \llbracket C \rrbracket_{\mathsf{p}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{p}\, \in\, \lfloor C \rfloor_l} \llbracket C \rrbracket_{\mathsf{p}} \right)$$

The EPP of a choreography C is the composition of the projections of all active processes and the merged projections of the service processes in C.

6.4.1 Projection Example

$$\llbracket C \rrbracket_{\mathsf{a}} = \begin{cases} 1.\ \textbf{acc}\ k_d : l_{\mathtt{A}}.\mathsf{a}[\mathtt{A}]; \\ 2.\ k_d : \mathtt{C} \text{ -> } \mathsf{a}[\mathtt{A}].get(\underline{\mathbf{req}}); \\ 3.\ \mathsf{if}\ \mathsf{a}.isValid(\underline{\mathbf{req}})\ \{ \\ 4.\quad k_d : \mathsf{a}[\mathtt{A}].\underline{\mathbf{req.rsc}} \text{ -> } \mathtt{S}.ok; \\ 5.\quad k_d : \mathsf{a}[\mathtt{A}] \text{ -> } \mathtt{C}.ok \\ 6.\ \}\ \mathsf{else}\ \{ \\ 7.\quad k_d : \mathsf{a}[\mathtt{A}].\underline{\mathbf{req.rsc}} \text{ -> } \mathtt{S}.ko; \\ 8.\quad k_d : \mathsf{a}[\mathtt{A}] \text{ -> } \mathtt{C}.ko \\ 9.\ \} \end{cases} \qquad \llbracket C \rrbracket_{\mathsf{dm}} = \begin{cases} 1.\ \textbf{acc}\ k_d : l_{\mathtt{S}}.\mathsf{dm}[\mathtt{DM}]; \\ 2.\ k_d : \mathtt{A} \text{ -> } \mathsf{dm}[\mathtt{DM}].\{ \\ 3.\quad ok(\underline{\mathbf{rsc}}); \\ 4.\qquad k_d : \mathsf{dm}[\mathtt{DM}].\mathsf{logok}(\underline{\mathbf{req}}) \text{ -> } \mathtt{L}.log; \\ 5.\quad \cdots, \\ 6.\quad ko(\underline{\mathbf{rsc}}); \\ 7.\qquad k_d : \mathsf{dm}[\mathtt{DM}].\mathsf{logko}(\underline{\mathbf{req}}) \text{ -> } \mathtt{L}.log \\ 8.\ \} \end{cases}$$

$$\llbracket C \rrbracket_{\mathsf{l}} = \Big\{ 1.\ \textbf{acc}\ k_d : l_{\mathtt{L}}.\mathsf{l}[\mathtt{L}]; \qquad 2.\ k_d : \mathtt{S} \text{ -> } \mathsf{l}[\mathtt{L}].log(\underline{\mathbf{log}});$$

Figure 6.10: EPPs of $C_c \mid C_s$. Projection on a, dm (excepts), and l.

As an example, we project the choreography $C = C_c \mid C_s$ presented in § 6.2.3. Applying the definition of EPP to C we obtain

$$\llbracket C \rrbracket = \llbracket C \rrbracket_{\mathsf{c}} \mid \llbracket C \rrbracket_{\mathsf{f}} \mid \llbracket C \rrbracket_{\mathsf{a}} \mid \llbracket C \rrbracket_{\mathsf{dm}} \mid \llbracket C \rrbracket_{\mathsf{l}}$$

Figure 6.10 reports some excerpts of $\llbracket C \rrbracket$, i.e., $\llbracket C \rrbracket_{\mathsf{a}}$, $\llbracket C \rrbracket_{\mathsf{dm}}$, and $\llbracket C \rrbracket_{\mathsf{l}}$. We illustrate how $\llbracket C \rrbracket$ implements the same behaviour of C.

Start. The start of a new session, like the one at Lines 1 of C_c and C_s, applies the same effect in the projection. Let $\llbracket C \rrbracket'_{\mathsf{p}}$ be the continuation of the process projection of C on p after Line 1. The execution of Lines 1 of the process projections composing $\llbracket C \rrbracket$ is:

$$D,\ \begin{array}{l} \textbf{req}\ k_d : \mathsf{c}[\mathtt{C}] \Leftrightarrow l_{\mathtt{A}}.\mathsf{a}[\mathtt{A}], l_{\mathtt{DM}}.\mathsf{dm}[\mathtt{DM}], l_{\mathtt{L}}.\mathsf{l}[\mathtt{L}]; \llbracket C \rrbracket'_{\mathsf{c}} \\ \mid \textbf{acc}\ k_d : l_{\mathtt{A}}.\mathsf{a}[\mathtt{A}]; \llbracket C \rrbracket'_{\mathsf{a}} \mid \textbf{acc}\ k_d : l_{\mathtt{DM}}.\mathsf{dm}[\mathtt{DM}]; \llbracket C \rrbracket'_{\mathsf{s}} \\ \mid \textbf{acc}\ k_d : l_{\mathtt{L}}.\mathsf{l}[\mathtt{L}]; \llbracket C \rrbracket'_{\mathsf{l}} \mid \llbracket C \rrbracket_{\mathsf{f}} \end{array} \to D',\ \llbracket C \rrbracket'_{\mathsf{c}} \mid \llbracket C \rrbracket'_{\mathsf{a}} \mid \llbracket C \rrbracket'_{\mathsf{dm}} \mid \llbracket C \rrbracket'_{\mathsf{l}} \mid \llbracket C \rrbracket_{\mathsf{f}}$$

Let the location of process c be $l_{\mathtt{C}}$ in D. The effect of the start is $\delta = \textbf{start}\ k_d : l_{\mathtt{C}}.\mathsf{c}[\mathtt{C}], l_{\mathtt{A}}.\mathsf{a}[\mathtt{A}], l_{\mathtt{DM}}.\mathsf{dm}[\mathtt{DM}], l_{\mathtt{L}}.\mathsf{l}[\mathtt{L}]$ which can generate the same D' as the start at Lines 1 of C_c and C_s.

Communications. The EPP projects a complete communication to a partial send for the sender and a partial receive for the receiver. Line 6 of C_s is $k_d : \mathsf{dm}[\mathtt{DM}].\mathsf{logok}(\underline{\mathbf{rsc}}) \text{ -> } \mathsf{l}[\mathtt{L}].log(\underline{\mathbf{log}})$ and in $\llbracket C \rrbracket$ it is projected into a partial send for dm (at Line 4 of $\llbracket C \rrbracket_{\mathsf{dm}}$) and a partial receive for l (at Line 2 of $\llbracket C \rrbracket_{\mathsf{l}}$). The semantics of Line 6 of C_s and that of Line 4 of $\llbracket C \rrbracket_{\mathsf{dm}}$ and Line 2 of $\llbracket C \rrbracket_{\mathsf{l}}$ is equal as the complete action breaks into a partial send, equal to the effect of Line 4 of $[\![C]\!]_{\mathsf{dm}}$, and continues as a partial receive, equal to $[\![C]\!]_{\mathsf{l}}$.

Conditionals. The EPP of a conditional merges the behaviour of all processes in its branches as branched receptions. For example, at Line 3 of C_s, process a evaluates a condition that either branches in the behaviour described at Lines 4–15 or at Lines 17–19. At Line 3 of $[\![C]\!]_{\mathsf{a}}$ the conditional is preserved verbatim. For the other processes present in the branches of the conditional, the EPP merges their behaviours on a branched reception: $[\![C]\!]_{\mathsf{dm}}$ projects the conditional on the branches guarded by operations ok or ko (resp. at Lines 3 and 6). $[\![C]\!]_{\mathsf{l}}$ merges the branches of the conditional into a single branch on operation log, at Line 2.

6.4.2 Properties

We present the properties of EPP. We prove that the EPP implements the same behaviour of its originating choreography, we build on the foundation that the EPP of a choreography is still typable and then we establish a bisimilarity relation between the semantics of the EPP and the originating choreography.

First we state our type preservation result using the *minimal typing* of choreographies $\vdash_{\mathsf{min}}$, which types the branches in rules $\lfloor^{\mathrm{T}}|_{\mathrm{SEND}}\rceil$ and $\lfloor^{\mathrm{T}}|_{\mathrm{RECV}}\rceil$ using the respective minimal branch types. As in [13], $\vdash_{\mathsf{min}}$ (defined in Appendix E.3.1) takes into account that, due to merging, the EPP of a complete choreography may still offer branches that the originating choreography discarded with a conditional. Thus we have:

Theorem 6 (EPP Typing Preservation). $\Gamma \vdash_{\mathsf{min}} D, C$ *implies* $[\![\Gamma]\!] \vdash_{\mathsf{min}} D, [\![C]\!]$

Where $[\![\Gamma]\!]$, defined in Appendix E.3.2, replaces the typings of recursive procedures in Γ with the typing of each procedure for each endpoint process taking part in it.

Combining the above Theorem with Theorem 4 we can prove:

Theorem 7 (EPP Operational Correspondence). *Let D, C be well-typed. Then,*

1. *(Completeness) $D, C \rightarrow D', C'$ implies $D, [\![C]\!] \rightarrow D', C''$ and $[\![C']\!] \prec C''$.*
2. *(Soundness) $D, [\![C]\!] \rightarrow D', C'$ implies $D, C \rightarrow D', C''$ and $[\![C'']\!] \prec C'$.*

In the theorem above $C \prec C'$ is the *pruning relation* (see Appendix E.3.4), a strong typed bisimilarity [96] such that C has some unused branches and always-available accepts.

We report the full proofs of Theorem 6 and Theorem 7 in Appendix E.3.

6.5 Dynamic Correlation Calculus Language

We introduce the Dynamic Correlation Calculus (DCC), the target language of our compilation, modelled on the Correlation Calculus [98] (CC) and extended to support the dynamic creation of queues. The reason to target CC is that it is the formal model of the executable language Jolie [92]. In this way, our results are immediately applicable to an actual implemented language, yet preserving the formality and simplicity of a theoretical approach. Moreover, the semantics of CC formalises message correlation à-la SOC, as in the standard service-oriented language BPEL [36], by following similar concepts to those that we used in our semantics for AC. However, it is complex to implement multiparty sessions in CC because processes are statically linked to a single queue and, through it, to a single correlation key. For these reasons, we extend CC to DCC by enabling processes to create queues at runtime and selectively read messages from them.

$$
\begin{array}{rlll}
S ::= & \langle B_s, P\rangle_l & (service) \\
\mid & S \mid S' & (network)
\end{array}
\qquad
\begin{array}{rlll}
B_s ::= & !(x); B & (accept) \\
\mid & \mathbf{0} & (inact)
\end{array}
$$

$$
\begin{array}{rll|ll}
B ::= & ?@e_1(e_2); B & (request) & \sum_i [o_i(x_i)\ \texttt{from}\ e]\ \{B_i\} & (choice) \\
\mid & o(x)\ \texttt{from}\ e; B & (input) & o@e_1(e_2)\ \texttt{to}\ e_3; B & (output) \\
\mid & \mathsf{if}\ e\ \{B_1\}\ \mathsf{else}\ \{B_2\} & (cond) & \mathrm{X} & (call) \\
\mid & \mathsf{def}\ X = B'\ \mathsf{in}\ B & (def) & x = e; B & (assign) \\
\mid & \mathsf{cq}(x); B & (cqueue) & \mathbf{0} & (inact)
\end{array}
$$

Figure 6.11: Dynamic Correlation Calculus, syntax

Syntax. The syntax of DCC (in Figure 6.11) is divided in two layers: *services*, ranged over by S, and *processes*, ranged over by P. The term $(service)$ describes a service, located at l, as a container of a *start behaviour* B_s and a (system of) processes P. A start behaviour creates new processes on request: following the syntax of B_s, a service can spawn a new process that stores the message of the request under the bound variable x and implements the behaviour B (see below). The term $(network)$ supports interactions among services. The process layer P defines the structure of processes, which run inside services, and their composition. In the syntax, a process is the association of a behaviour B, a state t, and a queue map M. The state t and the map M are defined exactly like the states and queue maps found in AC (§ 6.2). Still from AC, we also make use of names for operations (o), procedures (X), and variables (x, which we recall are paths). Expressions, ranged over by e, are evaluated at runtime on the state of the related process. Terms $(input)$ and $(output)$ model communications. In $(input)$ the process stores in x a message `from` the head of the queue correlating with e and expectedly received on operation o. Dually, $(output)$ models the delivery of a message on operation o with content e_2; e_1 defines the location of the service where the addressee (process) is running, whilst e_3 is the key that correlates with the receiving queue of the addressee. The term $(choice)$ models a branching input-choice. The argument of the choice is modelled after $(input)$. If one of the inputs can execute — receiving a message from the queue correlating with e on operation o_i — it discards all other receptions and executes the related behaviour B_i. Term $(request)$ is the dual of $(accept)$ and asks the service located at e_1 to spawn a new process, passing to it the message in e_2. Finally $(cqueue)$ models the creation of a new queue. After the creation, variable x contains the key that correlates with the new queue. All other terms of the syntax are standard.

Semantics. In Figure 6.12 we report the semantics of DCC, given as rules for a reduction relation $\to$. Relation $\to$ is closed under a structural congruence $\equiv$ defined in the standard way (reported in Appendix D.9); in particular, it supports commutativity and associativity for parallel composition. We comment the rules. Rule $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$ models the reception of messages: if the queue correlating with e has a message on operation o then the message is removed from the queue and its content is assigned to the variable x in the state of the process. Rule $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$ adds to M an empty queue (ε) correlating with a fresh key, stored in x. The key is unique within the service of the process, to avoid ambiguity, but we impose no requirements on its structure (it can be any tree as long as it is unique in the enclosing service). Rule $\lfloor^{\text{DCC}}|_{\text{SEND}}\rceil$ models the delivery of a message between processes in different services. In the Rule, the message from the sender is added to (the end of) the correlating queue of the receiver. Similarly, Rule $\lfloor^{\text{DCC}}|_{\text{START}}\rceil$ models the matching of a request to create a new process with the service that accepts it. The newly created process has B as its behaviour, a state initialised with the content of the request message, and an empty queue map.

6.6 Compiler from AC to DCC and Properties

We present our main result: a formally-correct compiler from AC to DCC that models how applied choreographies can be implemented in real-world languages.

6.6.1 Compiler

$\boxed{D, C}^{\Gamma}$ denotes the compilation of a well-typed running choreography D, C into an operationally-equivalent network of DCC services. Before giving the formal

$$\frac{t_c = \mathsf{eval}(e,t) \quad M(t_c) = (o,t') :: \tilde{m}}{o(x)\ \mathtt{from}\ e; B \cdot t \cdot M \to B \cdot t \lhd (\, x, t' \,) \cdot M[t_c \mapsto \tilde{m}]} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Recv}}\rceil$$

$$\frac{B \cdot t \cdot M \to B' \cdot t' \cdot M'}{\mathsf{def}\ X = B_1\ \mathsf{in}\ B \cdot t \cdot M \to \mathsf{def}\ X = B_1\ \mathsf{in}\ B' \cdot t' \cdot M'} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Ctx}}\rceil$$

$$\frac{i = 1 \text{ if } \mathsf{eval}(e,t) = \text{true}, i = 2 \text{ otherwise}}{\mathsf{if}\ e\ \{B_1\}\ \mathsf{else}\ \{B_2\} \cdot t \cdot M \to B_i \cdot t \cdot M} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Cond}}\rceil$$

$$\frac{j \in I \quad t_c = \mathsf{eval}(e,t) \quad M(t_c) = (o_j,t') :: \tilde{m}}{\sum_{i \in I} [o_i(x_i)\ \mathtt{from}\ e]\ \{B_i\}\ \cdot t \cdot M \to B_j \cdot t \lhd (\, x_j, t' \,) \cdot M[t_c \mapsto \tilde{m}]} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Choice}}\rceil$$

$$\frac{t' = \mathsf{eval}(e,t)}{x = e; B \cdot t \cdot M \quad \to \quad B \cdot t \lhd (\, x, t' \,) \cdot M} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Assign}}\rceil$$

$$\frac{P \quad \to \quad P'}{\langle B_s, P \mid P_1 \rangle_l \quad \to \quad \langle B_s, P' \mid P_1 \rangle_l} \ \lfloor^{\mathrm{DCC}}|_{\textsc{PPar}}\rceil$$

$$\frac{P = \mathsf{cq}(x); B \cdot t \cdot M \quad t_c \notin \bigcup_i \mathsf{dom}(M_i) \cup \mathsf{dom}(M) \quad t' = t \lhd (\, x, t_c \,)}{\langle B_s,\ P \mid \prod_i B_i \cdot t_i \cdot M_i \rangle_l \to \langle B_s,\ B \cdot t' \cdot M[t_c \mapsto \varepsilon] \mid \prod_i B_i \cdot t_i \cdot M_i \rangle_l} \ \lfloor^{\mathrm{DCC}}|_{\textsc{CQ}}\rceil$$

$$\frac{\begin{array}{ccc} P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \cdot M & \mathsf{eval}(e_1,t) = l & \mathsf{eval}(e_3,t) = t_c \\ \mathsf{eval}(e_2,t) = t_m & \multicolumn{2}{c}{M'' = M'[t_c \mapsto M'(t_c) :: (o,t_m)]} \end{array}}{\langle B_s,\ P \mid B' \cdot t' \cdot M' \mid P_1 \rangle_l \quad \to \quad \langle B_s,\ B \cdot t \cdot M \mid B' \cdot t' \cdot M'' \mid P_1 \rangle_l} \ \lfloor^{\mathrm{DCC}}|_{\textsc{InSend}}\rceil$$

$$\frac{\begin{array}{ccc} P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \cdot M & \mathsf{eval}(e_1,t) = l' & \mathsf{eval}(e_3,t) = t_c \\ \mathsf{eval}(e_2,t) = t_m & \multicolumn{2}{c}{M'' = M'[t_c \mapsto M'(t_c) :: (o,t_m)]} \end{array}}{\begin{array}{l} \langle B_s, P \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M' \mid P_2 \rangle_{l'} \\ \quad \to \quad \langle B_s, B \cdot t \cdot M \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M'' \mid P_2 \rangle_{l'} \end{array}} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Send}}\rceil$$

$$\frac{P_1 = ?@e_1(e_2); B_1 \cdot t_1 \cdot M_1 \quad \mathsf{eval}(e_1,t_1) = l \quad Q = B \cdot t_\perp \lhd (\, x, \mathsf{eval}(e_2,t_1) \,) \cdot \emptyset}{\langle !(x); B,\ P \rangle_l \mid \langle B'_s,\ P_1 \mid P_2 \rangle_{l'} \quad \to \quad \langle !(x); B,\ Q \mid P \rangle_l \mid \langle B'_s,\ B_1 \cdot t_1 \cdot M_1 \mid P_2 \rangle_{l'}} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Start}}\rceil$$

$$\frac{P = ?@e_1(e_2); B \cdot t \cdot M \quad \mathsf{eval}(e_1,t) = l \quad Q = B \cdot t_\perp \lhd (\, x, \mathsf{eval}(e_2,t) \,) \cdot \emptyset}{\langle !(x); B',\ P \mid P_1 \rangle_l \quad \to \quad \langle !(x); B',\ Q \mid B \cdot t \cdot M \mid P_1 \rangle_l} \ \lfloor^{\mathrm{DCC}}|_{\textsc{InStart}}\rceil$$

$$\frac{S \equiv S_1 \quad S_1 \to S'_1 \quad S'_1 \equiv S'}{S \quad \to \quad S'} \ \lfloor^{\mathrm{DCC}}|_{\textsc{Eq}}\rceil \qquad \frac{S_1 \to S'_1}{S_1 \mid S \quad \to \quad S'_1 \mid S} \ \lfloor^{\mathrm{DCC}}|_{\textsc{SPar}}\rceil$$

Figure 6.12: Correlation Calculus, semantics

definition of $\boxed{D, C}^{\Gamma}$, we need to introduce some additional notation. Let $\mathcal{C}[\cdot]$ be an AC context defined as usual: $\mathcal{C}[\cdot] ::= \cdot \;\mid\; \mathcal{C}[\cdot] \mid C \;\mid\; C \mid \mathcal{C}[\cdot]$. We then formalise the filtering operators $C|_{l}$ and $C|_{\mathsf{p}}$ as follows:

$$C|_{l} = C' \text{ if } C = \mathcal{C}[C'] \text{ and } C' = \textbf{acc}\ k : l.\mathsf{p}[\mathtt{A}]; C'',\ \mathbf{0} \text{ otherwise}$$
$$C|_{\mathsf{p}} = C' \text{ if } C = \mathcal{C}[C'] \text{ and } \{\mathsf{p}\} = \mathsf{fp}(C'),\ \mathbf{0} \text{ otherwise}$$

Intuitively, $C|_{l}$ returns the accept term at location l in C and $C|_{\mathsf{p}}$ returns the endpoint choreography of process p in C. Next, we denote with $\boxed{C}^{\Gamma}$ the compilation of a process projection defined on the rules reported in Figure 6.13 (and commented below). Below, we define our compiler abusing the notation $l \in \Gamma$ to say that $\mathsf{p}@l \in \Gamma$ for some p or that there is a service typing in Γ containing l.

$$\boxed{\textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C}^{\Gamma} = \quad \mathsf{start}(\ k,\ l'.\mathtt{A},\ \widetilde{l.\mathtt{B}}\ ); \boxed{C}^{\Gamma},\ \mathsf{p}@l' \in \Gamma$$

$$\boxed{\textbf{acc}\ k : l.\mathsf{q}[\mathtt{B}]; C}^{\Gamma} = \quad \mathsf{accept}(\ k,\ \mathtt{B}, \Gamma(\tilde{l})\ ); \boxed{C}^{\Gamma},\ l \in \tilde{l},\ \tilde{l} \in \Gamma$$

$$\boxed{k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; C}^{\Gamma} = \quad o@\underline{\mathbf{k.B.l}}(e)\ \mathtt{to}\ \underline{\mathbf{k.A.B}}; \boxed{C}^{\Gamma}$$

$$\boxed{k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I}}^{\Gamma} = \quad \sum_{i \in I} [o_i(x_i)\ \mathtt{from}\ \underline{\mathbf{k.A.B}}]\ \{\boxed{C_i}^{\Gamma}\}$$

$$\boxed{\mathsf{def}\ X = C'\ \mathsf{in}\ C}^{\Gamma} = \quad \mathsf{def}\ X = \boxed{C'}^{\Gamma} \mathsf{in}\ \boxed{C}^{\Gamma}$$

$$\boxed{\mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}}^{\Gamma} = \quad \mathsf{if}\ e\ \{\boxed{C_1}^{\Gamma}\}\ \mathsf{else}\ \{\boxed{C_2}^{\Gamma}\}$$

$$\boxed{X}^{\Gamma} = X \qquad \boxed{\mathbf{0}}^{\Gamma} = \mathbf{0}$$

$$\mathsf{start}(k, l_{\mathtt{A}}.\mathtt{A}, \widetilde{l_{\mathtt{B}}.\mathtt{B}}) =$$

$$\underbrace{\bigodot_{\underline{\mathbf{I}} \in \{\mathtt{A},\tilde{\mathtt{B}}\}} \underline{\mathbf{k.I.l}} = l_{\mathtt{I}}}_{(s_1)\text{ locations}}\ ;\ \underbrace{\bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \left(\overbrace{\mathsf{cq}(\underline{\mathbf{k.I.A}})}^{(s_{2.1})}; \overbrace{?@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}})}^{(s_{2.2})}; \overbrace{\mathit{sync}(\underline{\mathbf{k}})\ \mathtt{from}\ \underline{\mathbf{k.I.A}}}^{(s_{2.3})} \right)}_{(s_2)\text{ creation of correlation keys and service processes}};\ \underbrace{\bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \mathit{start}@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}})\ \mathtt{to}\ \underline{\mathbf{k.A.I}}}_{(s_3)\text{ session start}}$$

$$\mathsf{accept}(k, \mathtt{B}, G\langle \mathtt{A} | \tilde{\mathtt{C}} | \tilde{\mathtt{D}} \rangle) =$$

$$\underbrace{!(\underline{\mathbf{k}})}_{(a_1)}\ ;\ \underbrace{\bigodot_{\underline{\mathbf{I}} \in \{\mathtt{A},\tilde{\mathtt{C}}\}/\{\mathtt{B}\}} \left(\mathsf{cq}(\underline{\mathbf{k.I.B}}) \right)}_{(a_2)}\ ;\ \underbrace{\mathit{sync}@\underline{\mathbf{k.A.l}}(\underline{\mathbf{k}})\ \mathtt{to}\ \underline{\mathbf{k.B.A}}}_{(a_3)}\ ;\ \underbrace{\mathit{start}(\underline{\mathbf{k}})\ \mathtt{from}\ \underline{\mathbf{k.A.B}}}_{(a_4)}$$

Figure 6.13: Compiler from AC to DCC.

Definition 23 (Compilation)**.** *Let C be a parallel composition of endpoint choreographies and let $\Gamma \vdash D, C$ for some Γ and D. The compilation $\boxed{D, C}^{\Gamma}$ is:*

$$\boxed{D, C}^{\Gamma} = \prod_{l \in \Gamma} \left\langle \boxed{C|_l}^{\Gamma}, \prod_{\mathsf{p} \in D(l)} \boxed{C|_{\mathsf{p}}}^{\Gamma} \cdot D(\mathsf{p}).\mathsf{st} \cdot D(\mathsf{p}).\mathsf{que} \right\rangle_l$$

Intuitively, for each service $\langle B_s, P \rangle_l$ in the compiled network: $i)$ the start behaviour B_s is the compilation of the endpoint choreography in C accepting the creation of processes at location l; $ii)$ P is the parallel composition of the compilation of all active processes located at l, equipped with their respective states and queue maps according to D. Let us now comment the rules in Figure 6.13. We use the auxiliary notation $\odot$ for DCC behaviours, defined as $\bigodot_{i \in [1,n]} (B_i) = B_1; \ldots; B_n$.

Requests. Function start defines the compilation of (req) terms. start compiles (req) terms to create the queues and a part of the session descriptor of a valid session support (see Definition 16) for the starter. Given a session identifier k, the located role of the starter ($l_{\mathtt{A}}.\mathtt{A}$), and the other located roles in the session

$(\widetilde{l_{\mathtt{B}}.\mathtt{B}})$, start returns DCC code that: (s_1) includes in the session descriptor all the locations of the processes involved in the session and $(s_{2.1})$ all the keys correlating with the queues of the starter for the session. Then, $(s_{2.2})$ it requests the creation of all the service processes for the session, $(s_{2.3})$ waits for them to be ready using the reserved operation $sync$ and, finally, (s_3) sends them the complete session descriptor obtained after receiving from all processes their correlation keys (in the $sync$ step).

Accepts. (acc) terms define the start behaviour of the service accepting the creation of processes at a location. Given a session identifier k, the role $\mathtt{B}$ of the service process, and the service typing $G\langle\mathtt{A}|\tilde{\mathtt{C}}|\tilde{\mathtt{D}}\rangle$ of the location, function accept defines the compilation of (acc) terms. accept complements function start by compiling the code that (a_1) accepts the spawn of a new process which, in turn, (a_2) creates its queues (including their keys in the session descriptor passed by the starter), (a_3) returns the session descriptor to the starter, and (a_4) waits for the signal to start the session.

Other terms. We compile $(send)$ terms to $(output)$ terms. Notably, the compiled code contains the same elements used by the semantics of AC to implement correlation, i.e., the location of the receiver ($\underline{\mathbf{k.B.l}}$) and the key that correlates with its queue ($\underline{\mathbf{k.A.B}}$). $(recv)$ compiles to $(choice)$, which defines the path ($\underline{\mathbf{k.A.B}}$) of the key correlating with the receiving queue, used also in the semantics of AC. Terms (def), $(cond)$, $(call)$, and $(inact)$ compile to the relative terms in DCC.

6.6.2 Example of Compilation

We report in Figure 6.14 the compilation of the example in § 6.2.3.

We associate a default deployment D to $[\![C]\!]$ such that for some Γ and D *i)* $\Gamma \vdash D, [\![C]\!]$ holds; *ii)* the location of process c is $l_{\mathtt{C}}$. $l_{\mathtt{C}}$ is also the location of the service process f, which lets us to show how the compilation of active and service processes work. Figure 6.14 reports the result of the compilation $\boxed{D, [\![C]\!]}^{\Gamma}$ where, on top it shows the composition of compiled services and below it, from left to right, the compiled behaviours of processes c, a, and l.

Compiled Services. The network of compiled services, on top of Figure 6.14, shows that there is a compiled service for all locations ($l_{\mathtt{C}}, l_{\mathtt{A}}, l_{\mathtt{DM}}$, and $l_{\mathtt{L}}$) in C. In particular, S_c, the service located at $l_{\mathtt{C}}$, contains *i)* as its start behaviour the compilation of the process projection of f (which accepts requests on $l_{\mathtt{C}}$ in C) and *ii)* a running process whose behaviour is the compilation of the process projection of c, and whose state and queue map are those of c in D.

$$\overbrace{\left\langle \boxed{[\![C]\!]_{\mathsf{f}}}^{\Gamma}, \boxed{[\![C]\!]_{\mathsf{c}}}^{\Gamma} \cdot D(\mathsf{c}).\mathsf{st} \cdot D(\mathsf{c}).\mathsf{que} \right\rangle_{l_{\mathtt{C}}}}^{S_c} \mid \overbrace{\left\langle \boxed{[\![C]\!]_{\mathsf{a}}}^{\Gamma}, \mathbf{0} \right\rangle_{l_{\mathtt{A}}}}^{S_a} \mid \overbrace{\left\langle \boxed{[\![C]\!]_{\mathsf{dm}}}^{\Gamma}, \mathbf{0} \right\rangle_{l_{\mathtt{DM}}}}^{S_{dm}} \mid \overbrace{\left\langle \boxed{[\![C]\!]_{\mathsf{l}}}^{\Gamma}, \mathbf{0} \right\rangle_{l_{\mathtt{L}}}}^{S_l}$$

$$\boxed{[\![C]\!]_{\mathsf{c}}}^{\Gamma} = \begin{cases} 1.\ \underline{\mathbf{kd.C.l}} = l_{\mathtt{C}}; \\ 2.\ \underline{\mathbf{kd.A.l}} = l_{\mathtt{A}}; \\ 3.\ \underline{\mathbf{kd.DM.l}} = l_{\mathtt{DM}}; \\ 4.\ \underline{\mathbf{kd.L.l}} = l_{\mathtt{L}}; \\ 5.\ \mathsf{cq}(\underline{\mathbf{kd.A.C}}); \\ 6.\ ?@\underline{\mathbf{kd.A.l}}(\underline{\mathbf{kd}}); \\ 7.\ sync(\underline{\mathbf{kd}})\ \mathtt{from}\ \underline{\mathbf{kd.A.C}}; \\ 8.\ \ldots \\ 9.\ \ldots \\ 10.\ start@\underline{\mathbf{kd.A.l}}(\underline{\mathbf{kd}})\ \mathtt{to}\ \underline{\mathbf{kd.C.A}}; \\ 11.\ start@\underline{\mathbf{kd.S.l}}(\underline{\mathbf{kd}})\ \mathtt{to}\ \underline{\mathbf{kd.C.S}}; \\ 12.\ start@\underline{\mathbf{kd.C.l}}(\underline{\mathbf{kd}})\ \mathtt{to}\ \underline{\mathbf{kd.C.L}}; \\ 13.\ get@\underline{\mathbf{kd.A.l}}(\mathsf{mkReq}())\ \mathtt{to}\ \underline{\mathbf{kd.C.A}}; \\ 14.\ \ldots \end{cases} \qquad \boxed{[\![C]\!]_{\mathsf{a}}}^{\Gamma} = \begin{cases} 1.\ !(\underline{\mathbf{kd}}); \\ 2.\ \mathsf{cq}(\underline{\mathbf{kd.C.A}}); \\ 3.\ \mathsf{cq}(\underline{\mathbf{kd.S.A}}); \\ 4.\ \mathsf{cq}(\underline{\mathbf{kd.L.A}}); \\ 5.\ sync@\underline{\mathbf{kd.C.l}}(\underline{\mathbf{kd}})\ \mathtt{to}\ \underline{\mathbf{kd.A.C}}; \\ 6.\ start(\underline{\mathbf{kd}})\ \mathtt{from}\ \underline{\mathbf{kd.C.A}}; \\ 7.\ get(\underline{\mathbf{req}})\ \mathtt{from}\ \underline{\mathbf{kd.C.A}}; \\ 8.\ \mathsf{if}\ isValid(\underline{\mathbf{req}})\ \{ \\ 9.\ \quad ok@\underline{\mathbf{kd.S.l}}(\underline{\mathbf{req.rsc}})\ \mathtt{to}\ \underline{\mathbf{kd.A.S}}; \\ 10.\ \quad ok@\underline{\mathbf{kd.C.l}}()\ \mathtt{to}\ \underline{\mathbf{kd.A.C}} \\ 11.\ \}\ \mathsf{else}\ \{ \\ 12.\ \quad ko@\underline{\mathbf{kd.S.l}}(\underline{\mathbf{req.rsc}})\ \mathtt{to}\ \underline{\mathbf{kd.A.S}}; \\ 13.\ \quad ko@\underline{\mathbf{kd.C.l}}()\ \mathtt{to}\ \underline{\mathbf{kd.A.C}} \\ 14.\ \} \end{cases}$$

$$\boxed{[\![C]\!]_{\mathsf{l}}}^{\Gamma} = \begin{cases} 1.\ !(\underline{\mathbf{kd}}); 2.\ \mathsf{cq}(\underline{\mathbf{kd.C.L}}); 3.\ \mathsf{cq}(\underline{\mathbf{kd.A.L}}); 4.\ \mathsf{cq}(\underline{\mathbf{kd.S.L}}); 5.\ sync@\underline{\mathbf{k.C.l}}(\underline{\mathbf{kd}})\ \mathtt{to}\ \underline{\mathbf{kd.L.C}}; \\ 6.\ start(\underline{\mathbf{kd}})\ \mathtt{from}\ \underline{\mathbf{kd.C.L}}; 7.\ log(\underline{\mathbf{log}})\ \mathtt{from}\ \underline{\mathbf{kd.S.L}} \end{cases}$$

Figure 6.14: Compilation of $C = C_c \mid C_s$, behaviours of c (excepts), a, and l.

Starting a new session. Lines 1-/12 of $\boxed{[\![C]\!]_{\mathsf{c}}}^{\Gamma}$ show an excerpt of the compiled code of the request to start session k_d for process c: **req** k_d : $\mathsf{c}[\mathtt{C}] \Leftrightarrow l_{\mathtt{A}}.\mathtt{A}, l_{\mathtt{DM}}.\mathtt{DM}, l_{\mathtt{L}}.\mathtt{L}$. As defined by function start, at Lines 1-4 the process sets the locations of all involved roles in the Session Descriptor, rooted in $\underline{\mathbf{kd}}$. Lines 5/-7 define the interaction to start the process playing role $\mathtt{A}$: line-wise, the process creates the queue to received from role $\mathtt{A}$, it requests the creation of a new process at the relative location $\underline{\mathbf{kd.A.l}}$, passing to it the Session Descriptor, and, it waits for the process to return the updated Session Descriptor. Lines 10-/12 signal the start of the session to all the started processes. Lines 1-/6 of $\boxed{[\![C]\!]_{\mathsf{a}}}^{\Gamma}$ and $\boxed{[\![C]\!]_{\mathsf{l}}}^{\Gamma}$ show the code that complements the start. At the end of the procedure, $D, [\![C]\!]$ and $\boxed{D, [\![C]\!]}^{\Gamma}$ properly set a Session Support for session k_d.

In-session message passing. We show that the originating choreography and its compilation handle communications similarly. We compare the execution of Lines 2 of $[\![C]\!]_{\mathsf{c}}$[13] and $[\![C]\!]_{\mathsf{a}}$ with that of Line 13 of $\boxed{[\![C]\!]_{\mathsf{c}}}^{\Gamma}$ and Line 7 of $\boxed{[\![C]\!]_{\mathsf{a}}}^{\Gamma}$. Let D be the deployment associated with $[\![C]\!]'$, reduction of $[\![C]\!]$ to the considered Lines. $D, [\![C]\!]' \to D', [\![C]\!]''$ applying Rule $\lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$ on Line 2 of $[\![C]\!]_{\mathsf{c}}$. In D' the queue in which a receives from c contains the sent message: $D'(\mathsf{a}).\mathsf{que}(t_c) = (get, t_m)$, t_c correlation key rooted at $\underline{\mathbf{kd.C.A}}$ in the state of c and t_m evaluation

[13]Although not shown here, Line 2 of $[\![C]\!]_{\mathsf{c}}$ corresponds to Line 2 of C_c as partial actions are projected verbatim on their process by the EPP.

of function $\mathsf{mkReq}()$. Similarly, Line 13 of $\boxed{[\![C]\!]_{\mathsf{c}}}^{\Gamma}$ reduces with Rule $\lfloor^{\text{DCC}}|_{\text{SEND}}\rceil$. The Rule delivers the message on operation get, carrying the evaluation of e, to the service located at $\underline{\mathbf{kd.A.l}}$, which contains all service processes compilation of a, and in the queue correlating with $\underline{\mathbf{kd.A.C}}$. By the compilation, the compiled process relative to c has the state of c, hence, the value rooted at $\underline{\mathbf{kd.A.C}}$ is equal to t_c, $\underline{\mathbf{kd.A.l}}$ is equal to the location of $\mathtt{A}$, and the evaluation of e is equal to t_m. Moreover, since the network has just started (at Line 13 of $\boxed{[\![C]\!]_{\mathsf{c}}}^{\Gamma}$ and correspondent), we know that the receiving process had no message in the queue correlating with $\underline{\mathbf{kd.A.C}}$, thus, after the reduction, it only has message (get, t_m). The state of $D', [\![C]\!]''$ corresponds to the state of $\boxed{D', [\![C]\!]''}^{\Gamma}$. On this result, if $D', [\![C]\!]''$ reduces with Rule $\lfloor^{\text{C}}|_{\text{RECV}}\rceil$, $\boxed{D', [\![C]\!]''}^{\Gamma}$ can apply Rule $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$ and we are sure that *i*) the consumed message by the receiver (a and its compiled correspondent) is (get, t_m), *ii*) the relative queues are now empty, and *iii*) the receiver included in its state t_m under path $\underline{\mathbf{req}}$.

6.6.3 Properties

We present our main result: the operational correspondence between the semantics of a well-typed running Applied Choreography and the semantics of its compilation.

Theorem 8 (Compilation Operational Correspondence). *Let C be a composition of endpoint choreographies such that $\Gamma \vdash D, C$. Then we have that:*

1. *(Completeness) $D, C \to D', C'$ implies $\boxed{D, C}^{\Gamma} \to^{+} \boxed{D', C'}^{\Gamma'}$ for some Γ' such that $\Gamma' \vdash D', C'$.*
2. *(Soundness) $\boxed{D, C}^{\Gamma} \to^{*} S$ implies (i) $D, C \to^{*} D', C'$ and (ii) $S \to^{*} \boxed{D', C'}^{\Gamma'}$ for some D', C' and Γ' such that $\Gamma' \vdash D', C'$.*

We report the full proof of Theorem 8 in Appendix E.4.

Definition 23 is defined only for endpoint choreographies, however, using EPP, we can always reconduce a choreography to this case.

Lemma 2. *Let $\Gamma \vdash C$ then $[\![C]\!]$ is a composition of endpoint choreographies.*

Proof. Trivial from Definition 21 and Definition 23. □

We conclude by answering our research question. The compilation of the EPP of an Applied Choreography is always a correct implementation.

Corollary 5 (Applied Choreographies)**.** *Let* $\Gamma \vdash D, C$. *Then we have that:*

1. *(Completeness)* $D, C \to D', C'$ *implies* $\boxed{[\![D, C]\!]}^{\Gamma} \to^{+} \boxed{[\![D', C']\!]}^{\Gamma'}$ *for some* Γ' *such that* $\Gamma' \vdash D', C'$.
2. *(Soundness)* $\boxed{[\![D, C]\!]}^{\Gamma} \to^{*} S$ *implies (i)* $D, C \to^{*} D', C'$ *and (ii)* $S \to^{*} \boxed{[\![D', C']\!]}^{\Gamma'}$ *for some* D', C' *and* Γ' *such that* $\Gamma' \vdash D', C'$.

Proof. Trivial from the application of Theorem 8 and Lemma 2. □

Part IV

Conclusion

CHAPTER 7

Conclusion

In this final chapter we summarise the presented contributions, we discuss how they compare with related work, and define some direction of future investigation.

7.1 Adaptable Choreographies

In Chapters 4 and 5 we presented our theory for the safe dynamic update of distributed applications. Our model guarantees the absence of communication deadlocks and races by construction for the running distributed application, even in presence of updates that were unknown when the application was started. We build on the strong theoretical results of our model and develop AIOCJ, a programming framework that offers the minimality and conciseness of the choreographic approach yet compiling choreographic specifications to executable distributed programs that can adapt in a consistent and coherent way at runtime.

The two approaches closest to ours we are aware of are in the area of multiparty session types [43, 13, 78], and deal with dynamic software updates [17] and with monitoring of self-adaptive systems [109]. The main difference between [17] and our approach is that [17] targets concurrent applications which are not distributed and it relies on a check on the global state of the application to ensure that the update is safe. Such a check cannot be done by a single role, thus is impractical in a distributed setting. Furthermore, the language in [17] is much more constrained than ours, e.g., requiring each pair of participants to interact on a dedicated pair of channels, and assuming that all the roles not involved in a choice behave the same in the two branches. The approach in [109] is very different from ours, too. In particular, in [109] all the possible behaviours are available since the very beginning, both at the level of types and of processes, and a fixed adaptation function is used to switch between them. This difference derives from the distinction between self-adaptive applications, as they discuss, and applications updated from the outside, as in our case.

We also recall [110], which uses types to ensure safe adaptation. However,

[110] allows updates only when no session is active, while we change the behaviour of running DIOCs. Our work shares with [14] the interest in choreographies composition. However, [14] uses multiparty session types and only allows static parallel composition, while we replace a term inside an arbitrary context at runtime.

Extensions of multiparty session types with error handling [16, 111] share with us the difficulties in coordinating the transition from the expected pattern to an alternative pattern, but in their case the error recovery pattern is known since the very beginning, thus considerably simplifying the analysis.

We briefly compare some works that exploit choreographic descriptions for adaptation, but with very different aims. For instance, [112] defines rules for adapting the specification of the initial requirements for a choreography, thus keeping the requirements up-to-date in presence of run-time changes. Our approach is in the opposite direction: we are not interested in updating the system specification tracking system updates, but in programming and ensuring correctness of adaptation itself.

Other formal approaches to adaptation represent choreographies as annotated finite state automata. In [113] choreographies are used to propagate protocol changes to the other peers, while [114] presents a test to check whether a set of peers obtained from a choreography can be reconfigured to match a second one. Differently from ours, these works only provide change recommendations for adding and removing message sequences.

In principle, our update mechanism can be used to inject guarantees of freedom from deadlocks and races into existing approaches to adaptation (cf. Chapter 3). However, this task is cumbersome, due to the huge number and heterogeneity of those approaches. For each of them the integration with our techniques is far from trivial. Nevertheless, we already started it. Indeed, AIOCJ, we apply our technique to the adaptation mechanism described in [82]. While applications in [82] are not distributed and there are no guarantees on the correctness of the application after adaptation, applications in AIOCJ, based on the same adaptation mechanisms, are distributed and free from deadlocks and races by construction.

Furthermore, on the website [89], we give examples of how to integrate our approach with distributed [70] and dynamic [87] Aspect-Oriented Programming (AOP) and with Context-Oriented Programming (COP) (cf. § 3.1.2). We report in Appendix C two examples which respectively model Aspect- and Context-Oriented Programming. In general, we can deal with cross-cutting concerns like logging and authentication, typical of AOP, viewing pointcuts as empty scopes and advices as updates. Layers, typical of COP, can instead be defined by updates which can fire according to contextual conditions. We are also planning to apply our techniques to multiparty session types [43, 13, 78]. The main challenge here is to deal with multiple interleaved sessions. An initial analysis of the problem is presented in [115].

7.1.1 Adaptable Choreographies and their degree of flexibility

In Table 7.1, we report in § 3.1.3 how our solution of Adaptable Choreographies, as instantiated in AIOCJ, fits against the attributed of PAIS.

In summation, being a language-level solution, AIOCJ can implement the most part of decision deferral patterns, as reported in Table 3.1, from *Fully pre-specified processes* to *Ad-hoc composition*.

7.2 Applied Choreographies

In Chapter 6 we tackled the issue of formalising the implementation of choreography models. Our solution comprises *i*) a model of Applied Choreographies, *ii*) a type system to check AC against multiparty protocol specifications, and *iii*) a formally-correct compiler to obtain executable programs from choreographies.

The main novelty of AC regards its original semantics that abstracts the features of choreographies (message passing, creation of new sessions and processes) from their implementation. To this end we *i*) equip choreographies with a global deployment and *ii*) define a separate semantics of effects on deployments. Interestingly, deployments and effects hide the complexity of the implementation away from the semantics of choreographies. This means that we can preserve the semantics of choreographies and compose it with other definitions of deployment and semantics of effects to model different communication semantics (e.g., synchronous, asynchronous with buffers) and implementations (e.g., distributed objects [101]).

Deployments let us formalise how choreographies can go wrong (see §6.3.3) and prove that the theory of session types is useful not only to type communications on choreographies (like in [13, 14]) but also to check the correctness of deployments. We highlight that, except for the declaration of locations, AC has the same types and syntax from previous works [13, 14], i.e., developers only specify protocols and choreographies and do not deal with deployment information or correlation data.

Implementation Model. To the best of our knowledge, this is the first formal work on a compiler and an execution model for choreographies. All previous works on formal choreography languages only specify an EPP procedure towards a calculus based on name synchronisation, leaving the design of its concrete support to implementors.

Chor [99] and AIOCJ (cf. Chapter 5) are the only projects (that we are aware of) that aim at providing choreography languages with strong safety guarantees (e.g.,

Attribute	Degree/type	Description
Freedom	*modelling and composition*	Updates can be introduced at runtime and they are not bound to a pre-specified set of behaviours. Moreover, updates can compose other updates, e.g., using `scope` properties.
Planning approach	*plan-driven*, *iterative and continuous*, and *ad-hoc*	It is possible to implement all planning approaches by controlling which updates are available during the execution. The *plan-driven* approach provides only those updates that belong to the pre-specified model, the *iterative and continuous* approach changes the updates at runtime to adapt a coarse plan (implemented in the choreography) to upcoming requirements, and the *ad-hoc* approach do not constrain the presence of updates wrt a plan.
Scope	*regions* and *entire processes*	The basic usage of `scope`s is to enclose a region of a choreography, yet it is possible to update the whole process by enclosing the whole choreography into a `scope`.
Process perspective	*behaviours*, *data*, *functions*, *operational decisions*, *time*, and, *actors* (limited)	Updates can bring into the original choreography new *behaviours*, use different *data*, introduce new *functions* (which are liked to their *implementations*) and thanks to the presence of the `Environment`, it is also possible to encode *time*-triggered events. It is possible to change which *actors* perform tasks, yet they are limited to those that belonged to the original choreography.
Making and support of decision	*goal-based*, *rule-based*, *experience-based*, and *user decision*	AIOCJ implements the application of updates on rules. Updates can encode *goal-based* decisions by sharing a property that is common to the same goal. It is also possible to implement *experience-based* and *user decision* systems by e.g., letting experience systems and users change the values of environmental variables.
Degree of automation	*automation*, *system-supported*, and *manual*	It is possible to either let the framework apply updates in an automatic way or require (different degrees of) interaction with the user.

Table 7.1: Attributes of flexibility as implemented in AIOCJ.

deadlock-freedom). The languages respectively implement the formal models presented in [13] and Chapter 4. However, both implementations depart significantly from their respective formal models as they generate code based on message correlation whilst their formalisations of EPP use synchronisations on names. This gap between theoretical models and their implementations has two consequences: *i*) it breaks the correctness-by-construction guarantee of choreographies — there is no proof that the implementation correctly supports synchronisation on names — and *ii*) users must look into the complexity of the compilers to understand how the generated code implements the originating choreography. Implementations of other frameworks based on sessions share similar issues and follow different custom practices to implement the semantics of name synchronisation [52, 53, 54]. Thus, our work is a useful reference to formalise the implementation of session-based languages in general.

Choreography Language and Deployment. The fragment of endpoint choreographies in AC is remarkably similar to standard process languages based on sessions, e.g., those used in [44, 116, 43, 96]. Moreover, the syntax of AC resembles that of Compositional Choreographies [14], which introduced a notion of compositionality in choreographies. This similarity is intentional: our aim is to show that it is possible to provide a suitable implementation model for this kind of languages.

Notably, our semantics easily captures asynchronous message passing through the interplay of the swapping relation $\simeq_{\mathsf{C}}$, partial choreographies, and message queues. By contrast, previous works [13, 14] simulate asynchrony with encumbering ad-hoc rules that check if actions guarded by communications in continuations are allowed.

Several works have already analysed the behavioural expressiveness of choreographic descriptions as protocols (e.g., multiparty session types [108]) but, to the best of our knowledge, there are none on the expressiveness of choreographies meant as implementation languages. We leave this investigation as an interesting future work.

Integrating Dynamic Choreographies in Applied Choreographies. By integrating our theory of Dynamic Choreographies into that of Applied Choreographies we can proceed in the realisation of a formally proven programming framework for adaptable distributed systems. However, merging the two theoretical approaches constitutes a formidable challenge because we need to put together the typical static approach of typed sessions of ACs with dynamic distributed behaviours of Dynamic Choreographies. Some already-cited works in this direction are [43, 13, 78, 16, 111] whilst [117] discusses the main issues of integrating typed sessions that can update at runtime.

Delegation. Delegation is a standard feature of session-typed choreography models [13, 14] and enables to transfer the responsibility to continue a session from a process to another. We can support delegation in AC by adding a rule that atomically updates the session descriptors of all processes involved in a session when there is a delegation. However, we leave delegation to future work, as its introduction would bring more complexity in our compiler; specifically, we would have to compile appropriate communications to update the local session descriptors of the involved DCC processes. This is a renown issue of implementing delegation that we plan to address by formalising the techniques proposed in [52, 13].

Correlation keys. In the semantics of AC, we abstract how correlation keys are generated. With this loose definition we can capture several implementations, provided they satisfy the requirement of unicity of keys (wrt to locations). As future work, we plan to implement a language, based on AC, able to support custom procedures for the generation of correlation keys (e.g., from database queries, cookies, etc.).

Bibliography

[1] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. B. Postel, L. G. Roberts, and S. S. Wolff, “A brief history of the internet,” *CoRR*, vol. cs.NI/9901011, 1999. (Cited on pages 1 and 9.)

[2] G. F. Coulouris and J. Dollimore, *Distributed Systems: Concepts and Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988. (Cited on pages 1 and 9.)

[3] E. G. Coffman, M. Elphick, and A. Shoshani, “System deadlocks,” *ACM Comput. Surv.*, vol. 3, pp. 67–78, June 1971. (Cited on pages 1 and 11.)

[4] R. H. B. Netzer and B. P. Miller, “What are race conditions?: Some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, vol. 1, pp. 74–88, Mar. 1992. (Cited on pages 1 and 11.)

[5] D. Sangiorgi and D. Walker, *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001. (Cited on pages 2, 19, 20, 41, and 79.)

[6] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *ASPLOS*, pp. 329–339, ACM, 2008. (Cited on page 2.)

[7] C. Peltz, “Web services orchestration and choreography,” *Computer*, vol. 36, pp. 46–52, Oct 2003. (Cited on pages 2 and 14.)

[8] W3C WS-CDL Working Group, “Web services choreography description language version 1.0.” http://www.w3.org/TR/ws-cdl-10/, 2004. (Cited on pages 2, 16, and 80.)

[9] "Business Process Model and Notation." `http://www.omg.org/spec/BPMN/2.0/`. (Cited on pages 2 and 16.)

[10] Z. Qiu, X. Zhao, C. Cai, and H. Yang, "Towards the theoretical foundation of choreography," in *WWW*, (United States), pp. 973–982, IEEE Computer Society Press, 2007. (Cited on pages 2, 19, 79, 81, and 96.)

[11] M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centred programming for web services," in *Proc. of ESOP*, vol. 4421 of *LNCS*, pp. 2–17, Springer-Verlag, 2007. (Cited on pages 2, 19, 32, and 58.)

[12] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro, "Bridging the gap between interaction-and process-oriented choreographies," in *SEFM'08*, pp. 323–332, IEEE, 2008. (Cited on pages 2, 17, 19, 32, 79, and 81.)

[13] M. Carbone and F. Montesi, "Deadlock-freedom-by-design: multiparty asynchronous global programming," in *POPL*, pp. 263–274, 2013. (Cited on pages 2, 19, 20, 32, 79, 81, 83, 89, 96, 105, 117, 118, 119, 121, 122, and 203.)

[14] F. Montesi and N. Yoshida, "Compositional choreographies," in *CONCUR*, pp. 425–439, 2013. (Cited on pages 2, 19, 21, 81, 82, 83, 93, 94, 95, 102, 118, 119, 121, 189, 203, 205, and 216.)

[15] F. Montesi, "Kickstarting choreographic programming," *CoRR*, vol. abs/1502.02519, 2015. (Cited on page 2.)

[16] M. Carbone, K. Honda, and N. Yoshida, "Structured Interactional Exceptions in Session Types," in *Proc. of CONCUR'08*, vol. 5201 of *LNCS*, pp. 402–417, Springer, 2008. (Cited on pages 2, 118, and 121.)

[17] G. Anderson and J. Rathke, "Dynamic software update for message passing programs," in *APLAS*, vol. 7705 of *LNCS*, pp. 207–222, Springer, 2012. (Cited on pages 2 and 117.)

[18] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The interface message processor for the arpa computer network," in *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, AFIPS '70 (Spring), (New York, NY, USA), pp. 551–567, ACM, 1970. (Cited on page 9.)

[19] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. ACM*, vol. 19, pp. 395–404, July 1976. (Cited on page 9.)

[20] M. Weiser, “The computer for the 21st century,” *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, pp. 3–11, July 1999. (Cited on page 9.)

[21] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010. (Cited on page 9.)

[22] ITU-T, Geneva, “120: Message sequence chart (MSC),” 1996. (Cited on page 11.)

[23] G. Booch, *The unified modeling language user guide*. Pearson Education India, 2005. (Cited on page 11.)

[24] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret, “The world-wide web,” *Commun. ACM*, vol. 37, pp. 76–82, Aug. 1994. (Cited on page 12.)

[25] T. Berners-Lee, L. Masinter, M. McCahill, *et al.*, “Uniform resource locators (url),” 1994. (Cited on page 12.)

[26] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol–http/1.1,” 1999. (Cited on page 12.)

[27] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (xml),” *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998. (Cited on pages 12 and 84.)

[28] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, “Web services architecture, w3c working group note, 11 february 2004,” *World Wide Web Consortium, article available from: http://www. w3. org/TR/ws-arch*, 2004. (Cited on page 13.)

[29] E. Christensen, F. Curbera, and G. Meredith, “Web services description language (WSDL) 1.1. w3c,” tech. rep., Note 15, 2001, www. w3. org/TR/wsdl, 2001. (Cited on page 13.)

[30] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, “SOAP version 1.2 part 1: Messaging framework. w3c recommendation 24 june 2003,” *Retrieved from http://www. w*, vol. 3, 2003. (Cited on page 13.)

[31] M. P. Papazoglou, “Service-oriented computing: Concepts, characteristics and directions,” in *WISE 2003*, pp. 3–12, IEEE, 2003. (Cited on page 13.)

[32] Red Hat, “D-bus website.” http://www.freedesktop.org/wiki/Software/dbus/, 2014. (Cited on page 13.)

[33] E. Frank and I. Redmond, “Dcom: Microsoft distributed component object model,” *Redmond III November*, 1997. (Cited on page 13.)

[34] D. A. Chappell, *Enterprise service bus.* " O’Reilly Media, Inc.", 2004. (Cited on page 13.)

[35] SAP, “Sap remote function call.” http://help.sap.com/saphelp_nw04/helpdata/en/6f/1bd5b6a85b11d6b28500508b5d5211/content.htm, 2014. (Cited on page 14.)

[36] OASIS, “Web Services Business Process Execution Language.” http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. (Cited on pages 15, 32, 79, 80, 84, and 106.)

[37] D. Recordon and D. Reed, “Openid 2.0: a platform for user-centric identity management,” in *Proceedings of the second ACM workshop on Digital identity management*, pp. 11–16, ACM, 2006. (Cited on page 16.)

[38] G. Decker, O. Kopp, F. Leymann, and M. Weske, “Bpel4chor: Extending bpel for modeling choreographies,” in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 296–303, IEEE, 2007. (Cited on page 17.)

[39] R. M. Needham and M. D. Schroeder, “Using encryption for authentication in large networks of computers,” *Commun. ACM*, vol. 21, pp. 993–999, Dec. 1978. (Cited on page 18.)

[40] M. Bravetti and G. Zavattaro, “Towards a unifying theory for choreography conformance and contract compliance,” in *Proc. of SC’07*, vol. 4829, pp. 34–50, 2007. (Cited on pages 19 and 32.)

[41] S. Carpineti and C. Laneve, “A basic contract language for web services.,” in *ESOP’06*, LNCS, pp. 197–213, 2006. (Cited on page 19.)

[42] F. Montesi, *Choreographic Programming.* Ph.D. thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf. (Cited on page 19.)

[43] K. Honda, N. Yoshida, and M. Carbone, “Multiparty asynchronous session types,” in *Proc. of POPL*, vol. 43(1), pp. 273–284, ACM, 2008. (Cited on pages 19, 32, 79, 81, 82, 92, 94, 101, 117, 118, and 121.)

[44] K. Honda, V. Vasconcelos, and M. Kubo, "Language primitives and type disciplines for structured communication-based programming," in *ESOP'98*, vol. 1381 of *LNCS*, (Heidelberg, Germany), pp. 22–138, Springer-Verlag, 1998. (Cited on pages 19, 83, and 121.)

[45] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida, "Scribbling interactions with a formal foundation," in *Proc. of ICDCIT*, vol. 6536 of *LNCS*, pp. 55–75, Springer, 2011. (Cited on page 19.)

[46] PI4SOA, "http://www.pi4soa.org," 2008. (Cited on page 19.)

[47] Savara, "JBoss Community." `http://www.jboss.org/savara/`. (Cited on page 19.)

[48] F. Montesi, "Chor language website." `http://www.chor-lang.org`, 2014. (Cited on page 20.)

[49] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980. (Cited on pages 20 and 79.)

[50] F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with jolie," in *Web Services Foundations*, pp. 81–107, Springer, 2014. (Cited on pages 20 and 79.)

[51] M. A. Hiltunen and R. D. Schlichting, "Adaptive distributed and fault-tolerant systems," *International Journal of Computer Systems Science and Engineering*, vol. 11, pp. 125–133, 1995. (Cited on page 20.)

[52] R. Hu, N. Yoshida, and K. Honda, "Session-based distributed programming in java," in *ECOOP*, pp. 516–541, 2008. (Cited on pages 21, 79, 121, and 122.)

[53] R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda, "Practical interruptible conversations - distributed dynamic verification with session types and python," in *RV*, pp. 130–148, 2013. (Cited on pages 21 and 121.)

[54] R. Neykova and N. Yoshida, "Multiparty session actors," in *COORDINATION 2014, Proceedings*, pp. 131–146, 2014. (Cited on pages 21 and 121.)

[55] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. (Cited on pages 25 and 65.)

[56] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009. (Cited on page 25.)

[57] G. Salvaneschi, C. Ghezzi, and M. Pradella, “An analysis of language-level support for self-adaptive software,” *ACM Trans. Auton. Adapt. Syst.*, vol. 8, pp. 7:1–7:29, July 2013. (Cited on pages 25 and 26.)

[58] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Architecture-based runtime software evolution,” in *Proceedings of the 20th international conference on Software engineering*, pp. 177–186, IEEE Computer Society, 1998. (Cited on page 26.)

[59] J. Kramer and J. Magee, “Self-managed systems: an architectural challenge,” in *Future of Software Engineering, 2007. FOSE’07*, pp. 259–268, IEEE, 2007. (Cited on page 26.)

[60] S. R. White, D. M. Chess, J. O. Kephart, J. E. Hanson, and I. Whalley, “An architectural approach to autonomic computing,” in *Autonomic Computing, International Conference on*, pp. 2–9, IEEE Computer Society, 2004. (Cited on page 26.)

[61] C. Ghezzi, M. Pradella, and G. Salvaneschi, “An Evaluation of the Adaptation Capabilities in Programming Languages,” in *Proc. of SEAMS 2011*, pp. 50–59, ACM, 2011. (Cited on pages 26 and 32.)

[62] P. Maes, “Concepts and experiments in computational reflection,” in *ACM Sigplan Notices*, vol. 22, pp. 147–155, ACM, 1987. (Cited on page 27.)

[63] J. McCarthy, *LISP 1.5 programmer’s manual*. MIT press, 1965. (Cited on page 27.)

[64] T. Ledoux, “Implementing proxy objects in a reflective orb,” in *Proc. ECOOP*, vol. 97, Citeseer, 1997. (Cited on page 27.)

[65] J. Dowling, T. Schäfer, V. Cahill, P. Haraszti, and B. Redmond, “Using reflection to support dynamic adaptation of system software: A case study driven evaluation,” in *Reflection and Software Engineering*, pp. 169–188, Springer, 2000. (Cited on page 27.)

[66] J. Xu, B. Randell, and A. F. Zorzo, “Implementing software-fault tolerance in c++ and open c++: An object-oriented and reflective approach,” *Proc. CADTED*, vol. 96, pp. 224–229, 1996. (Cited on pages 27 and 28.)

[67] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997. (Cited on page 28.)

[68] A. Popovici, T. Gross, and G. Alonso, “Dynamic weaving for aspect-oriented programming,” in *Proceedings of the 1st international conference on Aspect-oriented software development*, pp. 141–147, ACM, 2002. (Cited on page 28.)

[69] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier, “JAC: A flexible solution for aspect-oriented programming in java,” in *Metalevel architectures and separation of crosscutting concerns*, pp. 1–24, Springer, 2001. (Cited on page 28.)

[70] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli, “JAC: an aspect-based distributed dynamic framework,” *Software: Practice and Experience*, vol. 34, no. 12, pp. 1119–1148, 2004. (Cited on pages 28 and 118.)

[71] R. Hirschfeld, P. Costanza, and O. Nierstrasz, “Context-oriented programming,” *Journal of Object Technology*, vol. 7, no. 3, 2008. (Cited on page 29.)

[72] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, “A comparison of context-oriented programming languages,” in *International Workshop on Context-Oriented Programming*, p. 6, ACM, 2009. (Cited on page 29.)

[73] C. Ghezzi, M. Pradella, and G. Salvaneschi, “Programming language support to context-aware adaptation: a case-study with erlang,” in *ICSE*, pp. 59–68, ACM, 2010. (Cited on pages 29 and 172.)

[74] M. Weske, *Business process management: concepts, languages, architectures*. Springer Science & Business Media, 2012. (Cited on page 30.)

[75] M. Dumas, W. M. Van der Aalst, and A. H. Ter Hofstede, *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, 2005. (Cited on page 30.)

[76] M. Reichert and B. Weber, *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media, 2012. (Cited on pages 30 and 31.)

[77] A. Charfi and M. Mezini, “Aspect-oriented web service composition with ao4bpel,” in *Web Services*, pp. 168–182, Springer, 2004. (Cited on page 32.)

[78] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani, “On global types and multi-party session,” *Logical Methods in Computer Science*, vol. 8, no. 1, 2012. (Cited on pages 32, 117, 118, and 121.)

[79] J. A. Bergstra and P. Klint, “The discrete time TOOLBUS - A software coordination architecture,” *Sci. Comput. Program.*, vol. 31, no. 2-3, pp. 205–229, 1998. (Cited on page 32.)

[80] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik, “Dynamic Adaptation of Fragment-Based and Context-Aware Business Processes,” in *Proc. of ICWS 2012*, pp. 33–41, 2012. (Cited on page 32.)

[81] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, “Constructing Adaptive Software in Distributed Systems,” in *Proc. of ICDCS’01*, vol. 6084, pp. 635–643, 2001. (Cited on page 32.)

[82] I. Lanese, A. Bucchiarone, and F. Montesi, “A Framework for Rule-Based Dynamic Adaptation,” in *Proc. of TGC 2010*, vol. 6084, pp. 284–300, 2010. (Cited on pages 32 and 118.)

[83] J. Zhang, H. Goldsby, and B. H. C. Cheng, “Modular Verification of Dynamically Adaptive Systems,” in *Proc. of AOSD’09*, pp. 161–172, ACM, 2009. (Cited on page 32.)

[84] L. Leite, G. Ansaldi Oliva, G. Nogueira, M. Gerosa, F. Kon, and D. Milojicic, “A systematic literature review of service choreography adaptation,” *Service Oriented Computing and Applications*, pp. 1–18, 2012. (Cited on page 32.)

[85] S. González, K. Mens, M. Colacioiu, and W. Cazzola, “Context traits: dynamic behaviour adaptation through run-time trait recomposition,” in *AOSD*, pp. 209–220, 2013. (Cited on page 32.)

[86] S. Goetz and I. Savga, “Exploring Role Based Adaptation,” in *RAM-SE*, pp. 21–26, Fakultät für Informatik, Universität Magdeburg, 2008. (Cited on page 32.)

[87] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley, “An aspect-oriented approach to dynamic adaptation,” in *WOSS*, pp. 85–92, ACM, 2002. (Cited on pages 32 and 118.)

[88] F. Montesi, C. Guidi, and G. Zavattaro, “Service-oriented programming with jolie,” in *Web Services Foundations*, pp. 81–107, 2014. (Cited on page 42.)

[89] "AIOCJ website." `http://www.cs.unibo.it/projects/jolie/aiocj.html`. (Cited on pages 58 and 118.)

[90] M. Carbone and F. Montesi, "Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming," in *POPL*, pp. 263–274, ACM, 2013. (Cited on page 58.)

[91] I. Lanese, F. Montesi, and G. Zavattaro, "Amending choreographies," in *WWV*, vol. 123, pp. 34–48, EPTCS, 2013. (Cited on page 58.)

[92] Jolie, "Programming Language." `http://www.jolie-lang.org/`. (Cited on pages 66, 79, and 106.)

[93] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference," in *IFIP Congress*, pp. 125–131, 1959. (Cited on page 67.)

[94] I. Eclipse, "The eclipse foundation," 2007. (Cited on page 71.)

[95] "Xtext website." `http://www.eclipse.org/Xtext/`. (Cited on page 71.)

[96] M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, p. 8, 2012. (Cited on pages 79, 81, 83, 96, 103, 105, 121, 189, and 216.)

[97] M. Carbone, F. Montesi, and C. Schürmann, "Choreographies, logically," in *CONCUR*, pp. 47–62, 2014. (Cited on pages 79 and 83.)

[98] F. Montesi and M. Carbone, "Programming services with correlation sets," in *ICSOC*, pp. 125–141, 2011. (Cited on pages 79, 81, 84, and 106.)

[99] Chor, "Programming Language." `http://www.chor-lang.org/`. (Cited on pages 79 and 119.)

[100] S. Carpineti, C. Laneve, and P. Milazzo, "Bopi - A distributed machine for experimenting web services technologies," in *ACSD 2005, 6-9 June 2005, St. Malo, France*, pp. 202–211, 2005. (Cited on page 79.)

[101] R. S. Chin and S. T. Chanson, "Distributed object-based programming systems," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 91–124, 1991. (Cited on pages 80 and 119.)

[102] S. Basu, T. Bultan, and M. Ouederni, "Deciding choreography realizability," in *POPL*, pp. 191–202, 2012. (Cited on page 81.)

[103] T. Bray, “The javascript object notation (json) data interchange format,” 2014. (Cited on page 84.)

[104] L. Bettini, M. Coppo, L. D’Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida, “Global progress in dynamically interleaved multiparty sessions,” in *CONCUR*, vol. 5201 of *LNCS*, pp. 418–433, Springer, 2008. (Cited on page 88.)

[105] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani, “Global progress for dynamically interleaved multiparty sessions,” *MSCS*, vol. 760, pp. 1–65, 2015. (Cited on page 92.)

[106] B. C. Pierce, *Types and Programming Languages*. MA, USA: MIT Press, 2002. (Cited on page 93.)

[107] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, “Choreography and orchestration conformance for system design,” in *COORDINATION’06*, vol. 4038 of *LNCS*, (Heidelberg, Germany), pp. 63–81, Springer-Verlag, 2006. (Cited on page 96.)

[108] P. Deniélou and N. Yoshida, “Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types,” in *ICALP 2013, Proceedings, Part II*, pp. 174–186, Springer, 2013. (Cited on pages 97 and 121.)

[109] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri, “Self-adaptive multiparty sessions,” *Service Oriented Computing and Applications*, pp. 1–20, 2014. (Cited on page 117.)

[110] C. Di Giusto and J. A. Pérez, “Disciplined structured communications with consistent runtime adaptation,” in *SAC*, pp. 1913–1918, ACM, 2013. (Cited on pages 117 and 118.)

[111] S. Capecchi, E. Giachino, and N. Yoshida, “Global Escape in Multiparty Sessions,” in *Proc. of FSTTCS 2010*, vol. 8 of *LIPIcs*, pp. 338–351, Schloss Dagstuhl, 2010. (Cited on pages 118 and 121.)

[112] I. Jureta, S. Faulkner, and P. Thiran, “Dynamic requirements specification for adaptable and open service-oriented systems,” in *ICSOC*, vol. 4749 of *LNCS*, pp. 270–282, Springer, 2007. (Cited on page 118.)

[113] S. Rinderle, A. Wombacher, and M. Reichert, “Evolution of process choreographies in dychor,” in *OTM Conferences (1)*, vol. 4275 of *LNCS*, pp. 273–290, Springer, 2006. (Cited on page 118.)

[114] A. Wombacher, “Alignment of choreography changes in BPEL processes,” in *IEEE SCC*, pp. 1–8, IEEE, 2009. (Cited on page 118.)

[115] M. Bravetti *et al.*, “Towards global and local types for adaptation,” in *SEFM Workshops*, vol. 8368 of *LNCS*, pp. 3–14, Springer, 2013. (Cited on page 119.)

[116] S. Gay and M. Hole, “Subtyping for session types in the pi calculus,” *Acta Informatica*, vol. 42, pp. 191–225, Nov. 2005. (Cited on pages 121 and 189.)

[117] M. Bravetti, M. Carbone, T. T. Hildebrandt, I. Lanese, J. Mauro, J. A. Pérez, and G. Zavattaro, “Towards global and local types for adaptation,” in *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*, pp. 3–14, Springer, 2013. (Cited on page 121.)

[118] R. Pawlak *et al.*, “JAC: an aspect-based distributed dynamic framework,” *Softw., Pract. Exper.*, vol. 34, no. 12, pp. 1119–1148, 2004. (Cited on page 167.)

Part V

Appendix

APPENDIX A

Adaptable Choreographies: Proofs

A.1 Proof of Theorem 1

In order to prove the bound on the complexity of the connectedness check we use the lemma below, showing that the checks to verify the connectedness for a single sequence operator can be performed in linear time on the size of the sets generated by transI and transF.

Lemma 3. *Given S, S' sets of multisets of two elements, checking if $\forall\ s \in S\ .\ \forall s' \in S'\ .\ s \cap s' \neq \emptyset$ can be done in $O(n)$ steps, where n is the maximum of $|S|$ and $|S'|$.*

Proof. Without loss of generality, we can assume that $|S| \leq |S'|$. If $|S| \leq 9$ then the check can be performed in $O(n)$ by comparing all the elements in S with all the elements in S'. If $|S| > 9$ then at least 4 distinct elements appear in the multisets in S since the maximum number of multisets with cardinality 2 obtained by 3 distinct elements is 9. In this case the following cases cover all the possibilities:

- there exist distinct elements a, b, c, d s.t. $\{a, b\}$, $\{a, c\}$, and $\{a, d\}$ belong to S. In this case for the check to succeed all the multisets in S' must contain a, otherwise the intersection of the multiset not containing a with one among the multisets $\{a, b\}$, $\{a, c\}$, and $\{a, d\}$ is empty. Similarly, since $|S'| > 9$, for the check to succeed all the multisets in S must contain a. Hence, if $\{a, b\}$, $\{a, c\}$, and $\{a, d\}$ belong to S then the check succeeds iff a belongs to all the multisets in S and in S'.

- there exist distinct elements a, b, c, d s.t. $\{a, b\}$ and $\{c, d\}$ belong to S. In this case the check succeeds only if S' is a subset of $\{\{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}$. Since $|S'| > 9$ the check can never succeed.

- there exist distinct elements a, b, c s.t. $\{a, a\}$ and $\{b, c\}$ belong to S. In this case the check succeeds only if S' is a subset of $\{\{a, b\}, \{a, c\}\}$. Since $|S'| > 9$ the check can never succeed.

- there exist distinct elements a, b s.t. $\{a, a\}$ and $\{b, b\}$ belong to S. In this case the check succeeds only if S' is a subset of $\{\{a, b\}\}$. Since $|S'| > 9$ the check can never succeed.

Summarising, if $|S| > 9$ the check can succeed iff all the multisets in S and in S' share a common element. The existence of such an element can be verified in time $O(n)$. $\square$

Theorem 1 (Connectedness-check complexity).
The connectedness of a DIOC process $\mathcal{I}$ can be checked in time $O(n^2 \log(n))$, where n is the number of nodes in the abstract syntax tree of $\mathcal{I}$.

Proof. To check the connectedness of $\mathcal{I}$ we first compute the values of the functions transI and transF for each node of the abstract syntax tree (AST). We then check for each sequence operator whether connectedness holds.

The functions transI and transF associate to each node a set of pairs of roles. Assuming an implementation of the data set structure based on balanced trees (with pointers), transI and transF can be computed in constant time for interactions, assignments, $\mathbf{1}$, $\mathbf{0}$, and sequence constructs. For while and scope constructs computing $\mathsf{transF}(\mathcal{I}')$ requires the creation of balanced trees having an element for every role of $\mathcal{I}'$. Since the roles are $O(n)$, $\mathsf{transF}(\mathcal{I}')$ can be computed in $O(n \log(n))$. For parallel and if constructs a union of sets is needed. The union costs $O(n \log(n))$ since each set generated by transI and transF contains at maximum n elements.

Since the AST contains n nodes, the computation of the sets generated by transI and transF can be performed in $O(n^2 \log(n))$.

To check connectedness we have to verify that for each node $\mathcal{I}';\mathcal{I}''$ of the AST $\forall \mathtt{R}_1 \to \mathtt{R}_2 \in \mathsf{transF}(\mathcal{I}'), \forall \mathtt{S}_1 \to \mathtt{S}_2 \in \mathsf{transI}(\mathcal{I}'')$. $\{\mathtt{R}_1, \mathtt{R}_2\} \cap \{\mathtt{S}_1, \mathtt{S}_2\} \neq \emptyset$. Since $\mathsf{transF}(\mathcal{I}')$ and $\mathsf{transI}(\mathcal{I}'')$ have $O(n)$ elements, thanks to Lemma 3, checking if $\mathcal{I}';\mathcal{I}''$ is connected costs $O(n)$. Since in the AST there are less than n sequence operators, checking the connectedness on the whole AST costs $O(n^2)$.

The complexity of checking the connectedness of the entire AST is therefore limited by the cost of computing functions transI and transF and of checking the connectedness. All these activities have a complexity of $O(n^2 \log(n))$. $\square$

A.2 Proof of Theorem 2

Before entering into the details of the proof, we recall the proof strategy that we described in § 4.7, which consists in defining a notion of bisimulation which implies weak trace equivalence. We report below the definition of bisimulation.

Definition 12 (Weak System Bisimulation)**.** A *weak system bisimulation* is a relation $\mathcal{R}$ between DIOC systems and DPOC systems such that if $(\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle, \langle \mathbf{I}', \mathcal{N}\rangle) \in \mathcal{R}$ then:

- if $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle$ then $\langle \mathbf{I}', \mathcal{N}\rangle \xrightarrow{\eta_1}, \dots, \xrightarrow{\eta_k}\xrightarrow{\mu} \langle \mathbf{I}''', \mathcal{N}'''\rangle$ with $\forall\, i \in [1 \dots k], \eta_i \in \{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x); o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(); \tau\}$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle, \langle \mathbf{I}''', \mathcal{N}'''\rangle) \in \mathcal{R}$;
- if $\langle \mathbf{I}', \mathcal{N}\rangle \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}'''\rangle$ with $\eta \in \{o^? : \mathtt{R}_1(v) \to \mathtt{R}_2(x); o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(); \surd; \mathcal{I}; \texttt{no-up}; \mathbf{I}''', \tau\}$ then one of the following two conditions holds:
  - $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\eta} \langle \Sigma'', \mathbf{I}', \mathcal{I}''\rangle$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle, \langle \mathbf{I}''', \mathcal{N}'''\rangle) \in \mathcal{R}$ or
  - $\eta \in \{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \tau\}$ and $(\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle, \langle \mathbf{I}''', \mathcal{N}'''\rangle) \in \mathcal{R}$

In the proof, we provide a relation $\mathcal{R}$ which relates each well-annotated connected DIOC system with its projection and show that it is a bisimulation. Such a relation is not trivial since events that are atomic in the DIOC, (e.g., the evaluation of the guard of a conditional, including the removal of the discarded branch), are not atomic at DPOC level. In the case of conditional, the DIOC transition is mimicked by a conditional performed by the role evaluating the guard, a set of auxiliary communications sending the value of the guard to the other roles, and local conditionals based on the received value. These mismatches are taken care by function upd (Definition 31). This function needs also to remove the auxiliary communications used to synchronise the termination of scopes, which have no counterpart after the DIOC scope has been consumed. However, we have to record the impact of the mentioned auxiliary communications on the possible executions. Thus we define an event structure for DIOC (Definition 25) and one for DPOC (Definition 26) and we show that the two are related (Lemma 5).

Thanks to the existence of a bisimulation relating each well-annotated connected DIOC system with its projection, we can prove that the projection is correct.

In our proof strategy we rely on the uniqueness of indexes that unfortunately it is not preserved by transitions due to while unfolding. Indeed, as an example, consider the DIOC $\mathbf{i}$: `while` b@R $\{\mathbf{j}\colon x$@R $= e\}$. If the condition b evaluates to true, in one step the application of Rule $\lfloor^{\text{DPOC}}|_{\text{WHILE-UNFOLD}}\rceil$ produces the DIOC $\mathbf{j}$: x@R $= e$; $\mathbf{i}$: `while` b@R $\{\mathbf{j}\colon x$@R $= e\}$ where the index $\mathbf{j}$ is used twice.

To solve this problem, instead of using indexes, we rely on *global indexes* built on top of indexes. Global indexes can be used both at the DIOC level and at the DPOC level and their uniqueness is preserved by transitions.

Definition 24 (Global index)**.** *Given an annotated DIOC process $\mathcal{I}$, or an annotated DPOC network $\mathcal{N}$, for each annotated construct with index ι we define its global index ξ as follows:*

- *if the construct is not in the body of a while then $\xi = \iota$;*
- *if the innermost while construct that contains the considered construct has global index ξ' then the considered construct has global index $\xi = \xi' : \iota$.*

Lemma 4 (Distinctness of Global Indexes)**.** *Given a well-annotated DIOC $\mathcal{I}$, a global state Σ, and a set of updates $\mathbf{I}$, if $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\eta_1} \dots \xrightarrow{\eta_n} \langle \Sigma', \mathbf{I}', \mathcal{I}' \rangle$ then all global indexes are distinct.*

Proof. The proof is by induction on the number n of transitions, using a stronger inductive hypothesis: indexes are distinct but, possibly, inside DIOC subterms of the form $\mathcal{I}; \mathbf{i}\colon \texttt{while}\ b@\mathtt{R}\ \{\mathcal{I}'\}$. In this last case, the same index can occur both in $\mathcal{I}$ and in $\mathcal{I}'$, in constructs with different global indexes. The statement of the Lemma follows directly since distinct indexes imply distinct global indexes.

In the base case ($n = 0$), thanks to well annotatedness, indexes are always distinct. The inductive case follows directly by induction for transitions with label $\surd$. Otherwise, we have a case analysis on the only axiom which derives a transition with label different from $\surd$.

The only difficult cases are $\lfloor^{\text{DIOC}} |_{\text{WHILE-UNFOLD}} \rceil$ and $\lfloor^{\text{DIOC}} |_{\text{UP}} \rceil$.

In the case of Rule $\lfloor^{\text{DIOC}} |_{\text{WHILE-UNFOLD}} \rceil$, note that the while is enabled, hence it cannot be part of a term of the form $\mathcal{I}; \mathbf{i}\colon \texttt{while}\ b@\mathtt{R}\ \{\mathcal{I}'\}$. Hence, indexes of the body of the while loop do not occur elsewhere. As a consequence, after the transition no clashes are possible with indexes in the context. Note also that indexes of the body of the loop are duplicated, but the resulting term has the form $\mathcal{I}; \mathbf{i}\colon \texttt{while}\ b@\mathtt{R}\ \{\mathcal{I}'\}$, thus global indexes are distinct by construction.

The case $\lfloor^{\text{DIOC}} |_{\text{UP}} \rceil$ follows thanks to the condition $\mathsf{freshIndexes}(\mathcal{I}')$.

□

Using global indexes we can now define event structures corresponding to the execution of DIOCs and DPOCs. We start by defining DIOC events. Some events correspond to transitions of the DIOC, and we say that they are enabled when the corresponding transition is enabled, executed when the corresponding transition is executed.

Definition 25 (DIOC events)**.** *We use ε to range over events, and we write $[\varepsilon]_{\mathtt{R}}$ to highlight that event ε is performed by role $\mathtt{R}$. An annotated DIOC $\mathcal{I}$ contains the following events:*

Communication events: *a sending event $\xi : \overline{o}@\mathtt{R}_2$ in role $\mathtt{R}_1$ and a receiving event $\xi : o@\mathtt{R}_1$ in role $\mathtt{R}_2$ for each interaction $\mathbf{i}\colon o : \mathtt{R}_1(e) \to \mathtt{R}_2(x)$ with global* *index ξ; we also denote the sending event as f_ξ or $[f_\xi]_{\mathtt{R}_1}$ and the receiving event as t_ξ or $[t_\xi]_{\mathtt{R}_2}$. Sending and receiving events correspond to the transition executing the interaction.*

Assignment events: *an assignment event ε_ξ in role* $\mathtt{R}$ *for each assignment* $\mathbf{i}\colon x@\mathtt{R} = e$ *with global index ξ; the event corresponds to the transition executing the assignment.*

Scope events: *a scope initialisation event $\uparrow_\xi$ and a scope termination event $\downarrow_\xi$ for each scope* $\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}$ *with global index ξ. Both these events belong to all the roles in* $\mathsf{roles}(\mathcal{I})$*. The scope initialisation event corresponds to the transition performing or not performing an update on the given scope. The scope termination event is just an auxiliary event (related to the auxiliary interactions implementing the scope termination).*

If events: *a guard if-event ε_ξ in role* $\mathtt{R}$ *for each construct* $\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}$ *with global index ξ; the guard-if event corresponds to the transition evaluating the guard of the condition.*

While events: *a guard while-event ε_ξ in role* $\mathtt{R}$ *for each construct* $\mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}$ *with global index ξ; the guard-while event corresponds to the transition evaluating the guard of the while loop.*

Function $\mathsf{events}(\mathcal{I})$ *denotes the set of events of the annotated DIOC $\mathcal{I}$. A sending and a receiving event with the same global index ξ are called matching events. We denote with $\overline{\varepsilon}$ an event matching event ε. A communication event is either a sending event or a receiving event. A communication event is unmatched if there is no event matching it.*

As a corollary of Lemma 4 events have distinct names. Note also that, for each while loop, there are events corresponding to the execution of just one loop. If unfolding is performed, new events are created.

Similarly to what we did for DIOC, we can define events for DPOC as follows.

Definition 26 (DPOC events)**.** *An annotated DPOC network $\mathcal{N}$ contains the following events:*

Communication events *a sending event $\xi : \overline{o^?}@\mathtt{R}_2$ in role* $\mathtt{R}_1$ *for each send* $\iota\colon \mathbf{i}.o^? : e\ \mathtt{to}\ \mathtt{R}_2$ *with global index ξ in role* $\mathtt{R}_1$*; and a receiving event $\xi : o^?@\mathtt{R}_1$ in role* $\mathtt{R}_2$ *for each receive* $\iota\colon \mathbf{i}.o^? : x\ \mathtt{from}\ \mathtt{R}_1$ *with global index ξ in role* $\mathtt{R}_2$*; we also denote the sending event as f_ξ or $[f_\xi]_{\mathtt{R}_1}$; and the receiving event as t_ξ or $[t_\xi]_{\mathtt{R}_2}$. Sending and receiving events correspond to the transitions executing the corresponding communication.*

Assignment events *an assignment event ε_ξ in role* $\mathtt{R}$ *for each assignment* $\mathbf{i}\colon x = e$ *with global index ξ; the event corresponds to the transition executing the assignment.*

Scope events *a scope initialisation event $\uparrow_\xi$ and a scope termination event $\downarrow_\xi$ for each* $\mathbf{i}:$ `scope` $@\mathtt{R}$ $\{P\}$ `roles` $\{S\}$ *or* $\mathbf{i}:$ `scope` $@\mathtt{R}$ $\{P\}$ *with global index ξ. Scope events with the same global index coincide, and thus the same event may belong to different roles; the scope initialisation event corresponds to the transition performing or not performing an update on the given scope for the role leading the update. The scope termination event is just an auxiliary event (related to the auxiliary interactions implementing the scope termination).*

If events *a guard if-event ε_ξ in role* $\mathtt{R}$ *for each construct* $\mathbf{i}$: `if` b $\{P\}$ `else` $\{P'\}$ *with global index ξ; the guard-if event corresponds to the transition evaluating the guard of the condition.*

While events *a guard while-event ε_ξ in role* $\mathtt{R}$ *for each construct* $\mathbf{i}$: `while` b $\{P\}$ *with global index ξ; the guard-while event corresponds to the transition evaluating the guard of the while loop.*

Let $\mathsf{events}(\mathcal{N})$ *denote the set of events of the network $\mathcal{N}$. A sending and a receiving event with either the same global index ξ or with global indexes differing only for replacing index $\mathbf{i}_?$ with $\mathbf{i}_\mathtt{T}$ or $\mathbf{i}_\mathtt{F}$ are called matching events. We denote with $\overline{\varepsilon}$ an event matching event ε.*

With a slight abuse of notation, we write $\mathsf{events}(P)$ to denote events originated by constructs in process P, assuming the network $\mathcal{N}$ to be understood. We use the same syntax for events of DIOCs and of DPOCs. Indeed, the two kinds of events are strongly related (cf. Lemma 5).

The relation below defines a causality relation DIOC among events based on the constraints given by the semantics on the execution of the corresponding transitions.

Definition 27 (DIOC causality relation)**.** *Let us consider an annotated DIOC $\mathcal{I}$. A causality relation $\leq_{DIOC} \subseteq$ $\mathsf{events}(\mathcal{I}) \times \mathsf{events}(\mathcal{I})$ is a partial order among events in $\mathcal{I}$. We define $\leq_{DIOC}$ as the minimum partial order satisfying:*

Sequentiality: *let $\mathcal{I}';\mathcal{I}''$ be a subterm of DIOC $\mathcal{I}$. If ε' is an event in $\mathcal{I}'$ and ε'' is an event in $\mathcal{I}''$, then $\varepsilon' \leq_{DIOC} \varepsilon''$.*

Scope: *let* $\mathbf{i}$: `scope` $@\mathtt{R}$ $\{\mathcal{I}'\}$ *be a subterm of DIOC $\mathcal{I}$. If ε' is an event in $\mathcal{I}'$ then $\uparrow_\xi \leq_{DIOC} \varepsilon' \leq_{DIOC} \downarrow_\xi$.*

Synchronisation: *for each interaction the sending event precedes the receiving event.*

If: *let* $\mathbf{i}$: `if` $b@\mathtt{R}$ $\{\mathcal{I}\}$ `else` $\{\mathcal{I}'\}$ *be a subterm of DIOC $\mathcal{I}$, let ε_ξ be the guard if-event in role* $\mathtt{R}$*, then for every event ε in $\mathcal{I}$ and for every event ε' in $\mathcal{I}'$ we have $\varepsilon_\xi \leq_{DIOC} \varepsilon$ and $\varepsilon_\xi \leq_{DIOC} \varepsilon'$.*

While: *let* $\mathbf{i}$: `while` $b@\mathtt{R}$ $\{\mathcal{I}\}$ *be a subterm of DIOC $\mathcal{I}$, let ε_ξ be the guard while-event in role* $\mathtt{R}$*, then for every event ε in $\mathcal{I}'$ we have $\varepsilon_\xi \leq_{DIOC} \varepsilon$.*

Similarly for DPOC:

Definition 28 (DPOC causality relation). *Let us consider an annotated DPOC network $\mathcal{N}$. A causality relation $\leq_{DPOC} \subseteq \mathsf{events}(\mathcal{N}) \times \mathsf{events}(\mathcal{N})$ is a partial order among events in $\mathcal{N}$. We define $\leq_{DPOC}$ as the minimum partial order satisfying:*

Sequentiality: *Let $P'; P''$ be a subterm of DPOC network $\mathcal{N}$. If ε' is an event in P' and ε'' is an event in P'' then $\varepsilon' \leq_{DPOC} \varepsilon''$.*

Scope: *Let* $\mathbf{i}$: `scope` `@R` $\{P\}$ `roles` $\{S\}$ *or* $\mathbf{i}$: `scope` `@R` $\{P\}$ *be a subterm of DPOC $\mathcal{N}$ with global index ξ. If ε' is an event in P then $\uparrow_\xi \leq_{DPOC} \varepsilon' \leq_{DPOC} \downarrow_\xi$.*

Synchronisation: *For each pair of events ε and ε', $\varepsilon \leq \varepsilon'$ implies $\overline{\varepsilon} \leq_{DPOC} \varepsilon'$.*

If: *Let* $\mathbf{i}$: `if` b $\{P\}$ `else` $\{P'\}$ *be a subterm of DPOC network $\mathcal{N}$ with global index ξ, let ε_ξ be the guard if-event, then for every event ε in P and for every event ε' in P' we have $\varepsilon_\xi \leq_{DPOC} \varepsilon$ and $\varepsilon_\xi \leq_{DPOC} \varepsilon'$.*

While: *Let* $\mathbf{i}$: `while` b $\{P\}$ *be a subterm of DPOC network $\mathcal{N}$ with global index ξ, let ε_ξ be the guard while-event, then for every event ε in P we have $\varepsilon_\xi \leq_{DPOC} \varepsilon$.*

Thanks to the definition above, we have that the events and causality relation are preserved by projection.

Lemma 5. *Given a DIOC process $\mathcal{I}$ and for each state Σ the DPOC network $\mathsf{proj}(\mathcal{I}, \Sigma)$ is such that:*

1. $\mathsf{events}(\mathcal{I}) \subseteq \mathsf{events}(\mathsf{proj}(\mathcal{I}, \Sigma))$;
2. $\forall\, \varepsilon_1, \varepsilon_2 \in \mathsf{events}(\mathcal{I}).\varepsilon_1 \leq_{DIOC} \varepsilon_2 \Rightarrow \varepsilon_1 \leq_{DPOC} \varepsilon_2 \vee \varepsilon_1 \leq_{DPOC} \overline{\varepsilon_2}$

Proof. 1. By definition of projection.

2. Let $\varepsilon_1 \leq_{DIOC} \varepsilon_2$. We have a case analysis on the condition used to derive the dependency.

   Sequentiality Consider $\mathcal{I} = \mathcal{I}'; \mathcal{I}''$. If events are in the same role the implication follows from the sequentiality of the $\leq_{DPOC}$.

   Let us show that there exists an event ε'' in an initial interaction of $\mathcal{I}''$ such that either $\varepsilon'' \leq_{DPOC} \varepsilon_2$ or $\varepsilon'' \leq_{DPOC} \overline{\varepsilon_2}$. The proof is by induction on the structure of $\mathcal{I}''$. The only difficult case is sequential composition. Assume $\mathcal{I}'' = \mathcal{I}_1; \mathcal{I}_2$. If $\varepsilon_2 \in \mathsf{events}(\mathcal{I}_1)$ the thesis follows from inductive hypothesis. If $\varepsilon_2 \in \mathsf{events}(\mathcal{I}_2)$ then by induction there exists an event ε_3 in an initial interaction of $\mathcal{I}_2$ such that $\varepsilon_3 \leq_{DPOC} \varepsilon_2$ or $\varepsilon_3 \leq_{DPOC} \overline{\varepsilon_2}$. By synchronisation (Definition 28) we have that $\overline{\varepsilon_3} \leq_{DPOC} \varepsilon_2$ or $\overline{\varepsilon_3} \leq_{DPOC} \overline{\varepsilon_2}$. By connectedness we have that ε_3 or $\overline{\varepsilon_3}$ are in the same role of an event ε_4 in $\mathcal{I}'$. By sequentiality (Definition 28) we have that $\varepsilon_4 \leq_{DPOC} \varepsilon_3$ or $\varepsilon_4 \leq_{DPOC} \overline{\varepsilon_3}$. By synchronisation we have that $\overline{\varepsilon_4} \leq_{DPOC} \varepsilon_3$ or $\overline{\varepsilon_4} \leq_{DPOC} \overline{\varepsilon_3}$. The thesis follows from the inductive hypothesis on ε_4 and by transitivity of $\leq_{DPOC}$.

Let us also show that there exists a final event $\varepsilon''' \in \mathsf{events}(\mathcal{I}')$ such that $\varepsilon_1 \leq_{DPOC} \varepsilon'''$ or $\varepsilon_1 \leq_{DPOC} \overline{\varepsilon'''}$. The proof is by induction on the structure of $\mathcal{I}'$. The only difficult case is sequential composition. Assume $\mathcal{I}' = \mathcal{I}_1;\mathcal{I}_2$. If $\varepsilon_1 \in \mathsf{events}(\mathcal{I}_2)$ the thesis follows from inductive hypothesis. If $\varepsilon_1 \in \mathsf{events}(\mathcal{I}_1)$ then the proof is similar to the one above, finding a final event in $\mathcal{I}_1$ and applying sequentiality, synchronisation, and transitivity.

The thesis follows from the two results above again by sequentiality, synchronisation, and transitivity.

Scope it means that either (*a*) $\varepsilon_1 = \uparrow_\xi$ and ε_2 is an event in the scope or (*b*) $\varepsilon_1 = \uparrow_\xi$ and $\varepsilon_2 = \downarrow_\xi$, or (*c*) ε_1 is an event in the scope and $\varepsilon_2 = \downarrow_\xi$. We consider case (*a*) since case (*c*) is analogous and case (*b*) follows by transitivity. If ε_2 is in the coordinator then the thesis follows easily. Otherwise it follows thanks to the auxiliary synchronisations with a reasoning similar to the one for sequentiality.

Synchronisation it means that ε_1 is a sending event and ε_2 is the corresponding receiving event, namely $\varepsilon_1 = \overline{\varepsilon_2}$. Thus, since $\varepsilon_2 \leq_{DPOC} \varepsilon_2$ then $\overline{\varepsilon_2} \leq_{DPOC} \varepsilon_2$.

If it means that ε_1 is the evaluation of the guard and ε_2 is an event in one of the two branches. Thus, if ε_2 is in the coordinator then the thesis follows easily. Otherwise it follows thanks to the auxiliary synchronisations with a reasoning similar to the one for sequentiality.

While it means that ε_1 is the evaluation of the guard and ε_2 is in the body of the while loop. Thus, if ε_2 is in the coordinator then the thesis follows easily. Otherwise it follows thanks to the auxiliary synchronisations with a reasoning similar to the one for sequentiality.

□

To complete the definition of our event structure we now define a notion of conflict between (DIOC and DPOC) events, relating events which are in different branches of the same conditional.

Definition 29 (Conflicting events). *Given a DIOC process $\mathcal{I}$, two events $\varepsilon, \varepsilon' \in \mathsf{events}(\mathcal{I})$ are conflicting if they belong to different branches of the same conditional, i.e., there exists a subprocess* $\mathbf{i}$: `if` b@R $\{\mathcal{I}'\}$ `else` $\{\mathcal{I}''\}$ *of $\mathcal{I}$ such that $\varepsilon \in \mathsf{events}(\mathcal{I}') \wedge \varepsilon' \in \mathsf{events}(\mathcal{I}'')$ or $\varepsilon' \in \mathsf{events}(\mathcal{I}') \wedge \varepsilon \in \mathsf{events}(\mathcal{I}'')$.*

Similarly, given a DPOC network $\mathcal{N}$, we say that two events $\varepsilon, \varepsilon' \in \mathsf{events}(\mathcal{N})$ are conflicting if they belong to different branches of the same conditional, i.e., there exists a subprocess $\mathbf{i}$: `if` b $\{P\}$ `else` $\{P'\}$ *of $\mathcal{N}$ such that $\varepsilon \in \mathsf{events}(P) \wedge \varepsilon' \in \mathsf{events}(P')$ or $\varepsilon' \in \mathsf{events}(P) \wedge \varepsilon \in \mathsf{events}(P')$.*

Similarly to what we did for DIOCs, we define below well-annotated DPOCs. Well-annotated DPOCs include all DPOCs obtained by projecting a well-annotated DIOC and executing the resulting network. They enjoy various properties useful for our proofs.

Definition 30 (Well-annotated DPOC)**.** *An annotated DPOC network $\mathcal{N}$ is well annotated for its causality relation $\leq_{DPOC}$ if the following conditions hold:*

C1 *for each global index ξ there are at most two communication events on public operations with global index ξ and, in this case, they are matching events;*

C2 *only events which are minimal according to $\leq_{DPOC}$ may correspond to enabled transitions;*

C3 *for each pair of non-conflicting sending events $[f_\xi]_{\mathtt{R}_1}$ and $[f_{\xi'}]_{\mathtt{R}_1}$ on the same operation $\mathbf{i}.o^?$ with the same target $\mathtt{R}_2$ such that $\xi \neq \xi'$ we have $[f_\xi]_{\mathtt{R}_1} \leq_{DPOC} [f_{\xi'}]_{\mathtt{R}_1}$ or $[f_{\xi'}]_{\mathtt{R}_1} \leq_{DPOC} [f_\xi]_{\mathtt{R}_1}$;*

C4 *for each pair of non-conflicting receiving events $[t_\xi]_{\mathtt{R}_2}$ and $[t_{\xi'}]_{\mathtt{R}_2}$ on the same operation $\mathbf{i}.o^?$ with the same sender $\mathtt{R}_1$ such that $\xi \neq \xi'$ we have $[t_\xi]_{\mathtt{R}_2} \leq [t_{\xi'}]_{\mathtt{R}_2}$ or $[t_{\xi'}]_{\mathtt{R}_2} \leq [t_\xi]_{\mathtt{R}_2}$;*

C5 *if ε is an event inside a scope with global index ξ then its matching events $\overline{\varepsilon}$ (if they exist) are inside a scope with the same global index.*

C6 *if two events have the same index but different global indexes then one of them, let us call it ε_1, is inside the body of a while loop with global index ξ_1 and the other, ε_2, is not. Furthermore, $\varepsilon_2 \leq_{DPOC} \varepsilon_{\xi_1}$ where ε_{ξ_1} is the guarding while-event of the while loop with global index ξ_1.*

Since scope update, conditional, and iteration at the DIOC level happen in one step, while they correspond to many steps of the projected DPOC, we introduce a function, denoted upd, that bridges this gap. More precisely, function upd is obtained as the composition of two functions, a function compl that completes the execution of DIOC actions which have already started, and a function clean that eliminates all the auxiliary closing communications of scopes (scope execution introduces in the DPOC auxiliary communications which have no correspondence in the DIOC).

Definition 31 (upd function)**.** *Let $\mathcal{N}$ be an annotated DPOC (we drop indexes if not relevant). The* upd *function is defined as the composition of a function* compl *and a function* clean*. Thus,* $\mathsf{upd}(\mathcal{N}) = \mathsf{clean}(\mathsf{compl}(\mathcal{N}))$*. Network* $\mathsf{compl}(\mathcal{N})$ *is obtained from $\mathcal{N}$ by repeating the following operations while possible.*

1. *Performing the reception of the positive evaluation of the guard of a while loop, by replacing for every* $\mathbf{i}.wb^*_{\mathtt{i}} : \mathtt{true}\ \mathtt{to}\ \mathtt{R}'$ *enabled, all the terms*

   $$\mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}; \mathtt{while}\ x_{\mathtt{i}}\ \{P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}\}$$

   not inside another while construct, with

   $$P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}; \mathtt{while}\ x_{\mathtt{i}}\ \left\{ \begin{array}{c} P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \\ \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R} \end{array} \right\}$$

   and replace $\mathbf{i}.wb^*_{\mathtt{i}} : \mathtt{true}\ \mathtt{to}\ \mathtt{R}'$ *with* $\mathbf{1}$.

2. *Performing the reception of the negative evaluation of the guard of a while loop by replacing, for every* $\mathbf{i}.wb^*_{\mathtt{i}} : \mathtt{false}\ \mathtt{to}\ \mathtt{R}'$ *enabled, all the terms*

   $$\mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}; \mathtt{while}\ x_{\mathtt{i}}\ \{P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}\}$$

   not inside another while construct, with $\mathbf{1}$*, and replace* $\mathbf{i}.wb^*_{\mathtt{i}} : \mathtt{false}\ \mathtt{to}\ \mathtt{R}'$ *with* $\mathbf{1}$.

3. *Performing the unfolding of a while loop by replacing every*

   $$\mathtt{while}\ x_{\mathtt{i}}\ \{P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}\}$$

   enabled not inside another while construct, such that $x_{\mathtt{i}}$ *evaluates to* `true` *in the local state, with*

   $$P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}; \mathtt{while}\ x_{\mathtt{i}}\ \left\{ \begin{array}{c} P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \\ \mathbf{i}.we^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R} \end{array} \right\}$$

4. *Performing the termination of while loop by replacing every*

   $$\mathtt{while}\ x_{\mathtt{i}}\ \{P; \mathbf{i}.we^*_{\mathtt{i}} : \mathsf{ok}\ \mathtt{to}\ \mathtt{R}; \mathbf{i}.wb^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}\}$$

   enabled not inside another while construct, such that $x_{\mathtt{i}}$ *evaluates to* `false` *in the local state, with* $\mathbf{1}$.

5. *Performing the reception of the positive evaluation of the guard of a conditional by replacing, for every* $\mathbf{i}.cnd^*_{\mathtt{i}} : \mathtt{true}\ \mathtt{to}\ \mathtt{R}'$ *enabled, all the terms*

   $$\mathbf{i}.cnd^*_{\mathtt{i}} : x_{\mathtt{i}}\ \mathtt{from}\ \mathtt{R}; \mathtt{if}\ x_{\mathtt{i}}\ \{P'\}\ \mathtt{else}\ \{P''\}$$

   not inside a while construct with P'*, and replace* $\mathbf{i}.cnd^*_{\mathtt{i}} : \mathtt{true}\ \mathtt{to}\ \mathtt{R}'$ *with* $\mathbf{1}$.

6. *Performing the reception of the negative evaluation of the guard of a conditional by replacing, for every* $\mathbf{i}.cnd^*_\mathbf{i}$: `false to R'` *enabled, all the terms*

$$\mathbf{i}.cnd^*_\mathbf{i} : x_\mathbf{i}\ \texttt{from}\ \mathtt{R}; \texttt{if}\ x_\mathbf{i}\ \{P'\}\ \texttt{else}\ \{P''\}$$

   not inside a while construct, with P''*, and replace* $\mathbf{i}.cnd^*_\mathbf{i}$: `false to R'` *with* $\mathbf{1}$.

7. *Performing the selection of the "then" branch by replacing every*

$$\texttt{if}\ x_\mathbf{i}\ \{P'\}\ \texttt{else}\ \{P''\}$$

   enabled, such that $x_\mathbf{i}$ *evaluates to* `true` *in the local state, with* P'.

8. *Performing the selection of the "else" branch by replacing every*

$$\texttt{if}\ x_\mathbf{i}\ \{P'\}\ \texttt{else}\ \{P''\}$$

   enabled, such that $x_\mathbf{i}$ *evaluates to* `false` *in the local state, with* P''.

9. *Performing the communication of the updated code by replacing, for every* $\mathbf{i}.sb^*_\mathbf{i}$: P `to S` *enabled, all the terms*

$$\mathbf{i}\text{: }\texttt{scope}\ @\mathtt{R}\ \{P'\}$$

   in role `S` *not inside a while construct with* P*, and replace* $\mathbf{i}.sb^*_\mathbf{i}$: P `to S` *with* $\mathbf{1}$.

10. *Performing the communication that no update is needed by replacing, for each*
    $\mathbf{i}.sb^*_\mathbf{i}$: `no to S` *enabled, all the terms*

$$\mathbf{i}\text{: }\texttt{scope}\ @\mathtt{R}\ \{P'\}$$

   in role `S` *not inside a while construct with* P'*, and replace* $\mathbf{i}.sb^*_\mathbf{i}$: P `to S` *with* $\mathbf{1}$.

Network $\mathsf{clean}(\mathcal{N})$ *is obtained from* $\mathcal{N}$ *by repeating the following operations while possible:*

- *Removing the auxiliary communications for end of scope and end of while loop synchronisation by replacing each*

$$\begin{array}{ll} \mathbf{i}.se^*_\mathbf{i} : \mathsf{ok}\ \texttt{to}\ \mathtt{R} & \mathbf{i}_\mathsf{C}\text{: }\mathbf{i}.we^*_\mathbf{i} : \mathsf{ok}\ \texttt{to}\ \mathtt{R} \\ \mathbf{i}.se^*_\mathbf{i} : \_\ \texttt{from}\ \mathtt{R} & \mathbf{i}_\mathsf{C}\text{: }\mathbf{i}.we^*_\mathbf{i} : \_\ \texttt{from}\ \mathtt{R} \end{array}$$

  not inside a while construct with $\mathbf{1}$.

Furthermore clean *may apply 0 or more times the following operation:*

- *replace a subterm* $\mathbf{1}; P$ *by* P *or a subterm* $\mathbf{1} \mid P$ *by* P.

Note that function compl does not reduce terms inside a while construct. Assume, for instance, to have an auxiliary send targeting a receive inside the body of a while loop. These two communications should not interact since they have different global indexes. This explains why we exclude terms inside the body of while loops.

We proceed now to prove some of the proprieties of DIOC and DPOC. The first result states that in a well-annotated DPOC only transitions corresponding to events minimal with respect to the causality relation $\leq_{DPOC}$ may be enabled.

Lemma 6. *If $\mathcal{N}$ is a DPOC, $\leq_{DPOC}$ its causality relation and ε is an event corresponding to a transition enabled in $\mathcal{N}$ then ε is minimal with respect to $\leq_{DPOC}$.*

Proof. The proof is by contradiction. Suppose ε is enabled but not minimal, i.e. there exists ε' such that $\varepsilon' \leq_{DPOC} \varepsilon$. If there is more than one such ε' consider the one such that the length of the derivation of $\varepsilon' \leq_{DPOC} \varepsilon$ is minimal. This derivation should have length one, and following Definition 28 it may result from one of the following cases:

- Sequentiality: $\varepsilon' \leq_{DPOC} \varepsilon$ means that $\varepsilon' \in \mathsf{events}(P')$, $\varepsilon \in \mathsf{events}(P'')$, and $P'; P''$ is a subterm of $\mathcal{N}$. Because of the semantics of sequential composition ε cannot be enabled.

- Scope: let $\mathbf{i}$: `scope` `@R` $\{P\}$ `roles` $\{S\}$ or $\mathbf{i}$: `scope` `@R` $\{P\}$ be a subprocess of $\mathcal{N}$ with global index ξ. We have the following cases:

  - $\varepsilon' = \uparrow_\xi$ and $\varepsilon \in \mathsf{events}(P)$, and this implies that ε cannot be enabled since if ε' is enabled then the Rules $\lfloor^{\text{DPOC}}|_{\text{UP}}\rceil$ or $\lfloor^{\text{DPOC}}|_{\text{NOUP}}\rceil$ for starting the execution of the scope have not been applied yet;
  - $\varepsilon' = \uparrow_\xi$ and $\varepsilon = \downarrow_\xi$, or $\varepsilon' \in \mathsf{events}(P)$ and $\varepsilon = \downarrow_\xi$: this is trivial, since $\downarrow_\xi$ is an auxiliary event and no transition corresponds to it;

- If: $\varepsilon' \leq_{DPOC} \varepsilon$ means that ε' is the evaluation of the guard of a subterm $\mathbf{i}$: `if` $x_{\mathtt{i}}$ $\{P'\}$ `else` $\{P''\}$ and $\varepsilon \in \mathsf{events}(P') \cup \mathsf{events}(P'')$. Event ε cannot be enabled because of the semantics of conditionals.

- While: $\varepsilon' \leq_{DPOC} \varepsilon$ means that ε' is the evaluation of the guard of a subterm $\mathbf{i}$: `while` $x_{\mathtt{i}}$ $\{P\}$ and $\varepsilon \in \mathsf{events}(P)$. Event ε cannot be enabled because of the semantics of the while loop.

□

We now prove that all the DPOCs obtained as projection of well-annotated connected DIOCs are well-annotated.

Lemma 7. *Let $\mathcal{I}$ be a well-annotated connected DIOC process and Σ a state. Then the projection $\mathcal{N} = \mathsf{proj}(\mathcal{I}, \Sigma)$ is a well-annotated DPOC network with respect to $\leq_{DPOC}$.*

Proof. We have to prove that $\mathsf{proj}(\mathcal{I}, \Sigma)$ satisfies the conditions of Definition 30 of well-annotated DPOC:

C1 For each global index ξ there are at most two communication events on public operations with global index ξ and, in this case, they are matching events. The condition follows by the definition of the projection function, observing that in well-annotated DIOCs, each interaction has its own index, and different indexes are mapped to different global indexes.

C2 Only events which are minimal according to $\leq_{DPOC}$ may correspond to enabled transitions. This condition follows from Lemma 6.

C3 For each pair of non-conflicting sending events $[f_{\xi}]_{\mathtt{R}}$ and $[f_{\xi'}]_{\mathtt{R}}$ on the same operation $o^{?}$ and with the same target $\mathtt{R}'$ such that $\xi \neq \xi'$ we have $[f_{\xi}]_{\mathtt{R}} \leq_{DPOC} [f_{\xi'}]_{\mathtt{R}}$ or $[f_{\xi'}]_{\mathtt{R}} \leq_{DPOC} [f_{\xi}]_{\mathtt{R}}$. Note that the two events are in the same role $\mathtt{R}$, thus without loss of generality we can assume that there exist two processes P, P' such that $[f_{\xi}]_{\mathtt{R}} \in \mathsf{events}(P)$ and $[f_{\xi'}]_{\mathtt{R}} \in \mathsf{events}(P')$ and there is a subprocess of $\mathcal{N}$ of one of the following forms:

- $P; P'$: the thesis follows by sequentiality (Definition 28);
- $P|P'$: this case can never happen for the reasons below. For events on public operations this follow by the definition of projection, since the prefixes of the names of operations are different. For events on private operations originated by the same construct this follows since all the targets are different. For events on private operations originated by different constructs this follows since the prefixes of the names of the operations are different.
- $\mathtt{if}\ b\ \{P\}\ \mathtt{else}\ \{P'\}$: this case can never happen since the events are non-conflicting (Definition 29).

C4 Similar to the previous case, with receiving events instead of sending events.

C5 If ε is an event inside a scope with global index ξ then its matching events $\overline{\varepsilon}$ (if they exist) are inside a scope with the same global index. This case holds by definition of the projection function.

C6 If two events have the same index but different global indexes then one of them, let us call it ε_1, is inside the body of a while loop with global index ξ_1 and the other, ε_2, is not. Furthermore, $\varepsilon_2 \leq_{DPOC} \varepsilon_{\xi_1}$ where ε_{ξ_1} is the guarding while-event of the while loop with global index ξ_1. By definition of well-annotated DIOC and of projection the only case where there are two events with the same index and different global indexes is for the auxiliary communications in the projection of the while construct, where the conditions hold by construction.

$\square$

The next lemma shows that for every starting set of updates $\mathbf{I}$ the DPOC $\mathcal{N}$ and $\mathsf{upd}(\mathcal{N})$ have the same set of weak traces.

Lemma 8. *Let $\mathcal{N}$ be a DPOC. The following properties hold:*

1. *if* $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N})\rangle \xrightarrow{\eta} \langle \mathbf{I}', \mathcal{N}'\rangle$ *with* $\eta \in \{o^? : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \mathbf{I}', \surd, \mathcal{I},$ $\mathsf{no\text{-}up}, \tau\}$ *then there exists* $\mathcal{N}''$ *s.t.* $\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\eta_1} \dots \xrightarrow{\eta_k}\xrightarrow{\eta} \langle \mathbf{I}', \mathcal{N}''\rangle$ *where* $\eta_i \in \{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \tau\}$ *for each* $i \in \{1, \dots, k\}$ *and* $\mathsf{upd}(\mathcal{N}'') = \mathsf{upd}(\mathcal{N}')$.

2. *if* $\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\eta} \langle \mathbf{I}', \mathcal{N}'\rangle$ *for* $\eta \in \left\{ \begin{array}{c} o^? : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \\ \mathbf{I}', \surd, \mathcal{I}, \mathsf{no\text{-}up}, \tau \end{array} \right\}$, *then one of the following holds:*

   (a) $\mathsf{upd}(\mathcal{N}) = \mathsf{upd}(\mathcal{N}')$ *and* $\eta \in \{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \tau\}$*, or*

   (b) $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N})\rangle \xrightarrow{\eta} \langle \mathbf{I}', \mathcal{N}''\rangle$ *such that* $\mathsf{upd}(\mathcal{N}') = \mathsf{upd}(\mathcal{N}'')$.

Proof.

1. Applying the upd function corresponds to perform weak transitions, namely transitions with labels in $\{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \tau\}$. Some of such transitions may not be enabled yet. Hence, $\mathcal{N}$ may perform the subset of the weak transitions above which are enabled, reducing to some $\mathcal{N}'''$. Then, η is enabled also in $\mathcal{N}'''$ and we have $\langle \mathbf{I}, \mathcal{N}'''\rangle \xrightarrow{\eta} \langle \mathbf{I}', \mathcal{N}''\rangle$. At this point we have that $\mathcal{N}''$ and $\mathcal{N}'$ may differ only for the weak transitions that were not enabled, which can be executed by upd.

2. There are two cases. In the first case the transition with label η is one of the transitions executed by function upd. In this case the condition 2a holds. In the second case, the transition with label η is not one of the transitions executed by function upd. In this case the transition with label η is still enabled

in $\mathsf{upd}(\mathcal{N})$ and can be executed. This leads to a network that differs from $\mathcal{N}'$ only because of transitions executed by the upd function and case 2b holds.

□

We now prove a few properties of transitions with label $\surd$.

Lemma 9. *For each DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ that reduces with a transition labelled $\surd$ then, for each role $\mathtt{R} \in \mathsf{roles}(\mathcal{I})$, the DPOC role $(\pi(\mathcal{I}, \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}}$ can reduce with a transition labelled $\surd$ and vice versa.*

Proof. Note that a DIOC can perform a transition with label $\surd$ only if it is a term obtained using sequential and/or parallel composition starting from $\mathbf{1}$ constructs. The projection has the same shape, hence it can perform the desired transition. The other direction is similar. □

The next lemma shows that if two matching events are enabled in the projection of a DIOC, then the corresponding interaction is enabled in the DIOC.

Lemma 10. *Let $\mathcal{I}$ be a DIOC obtained from a well-annotated connected DIOC via 0 or more transitions and $\mathbf{i}\colon o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ be an interaction in $\mathcal{I}$. If $\mathbf{i}\colon \mathbf{i}.o : e\ \mathtt{to}\ \mathtt{R}_2$ and $\mathbf{i}\colon \mathbf{i}.o : x\ \mathtt{from}\ \mathtt{R}_1$ are matching events and are both enabled in $\mathsf{proj}(\mathcal{I}, \Sigma)$ then $\mathbf{i}\colon o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ is enabled.*

Proof. Note that well-annotatedness in preserved under reduction, hence $\mathcal{I}$ is well-annotated.

If $\mathcal{I}$ is connected for sequence, then the proof is by structural induction on $\mathcal{I}$. The cases for $\mathbf{1}$, $\mathbf{0}$, scopes, conditionals, and while loops are trivial. For parallel composition consider that, since the two events are matching, then they have the same global index. As a consequence, they are from the same component, and the thesis follows by inductive hypothesis. Let us consider sequential composition. Suppose $\mathcal{I} = \mathcal{I}';\mathcal{I}''$. If $\mathbf{i}\colon o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x) \in \mathcal{I}'$ then the thesis follows by inductive hypothesis. Otherwise, by inductive hypothesis $\mathbf{i}\colon o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ is enabled in $\mathcal{I}''$. Thus, $\mathtt{R}_1 \rightarrow \mathtt{R}_2 \in \mathsf{transI}(\mathcal{I}'')$. From connectedness, for each $\mathtt{S}_1 \rightarrow \mathtt{S}_2 \in \mathsf{transF}(\mathcal{I}')$ we have $\{\mathtt{R}_1, \mathtt{R}_2\} \cap \{\mathtt{S}_1, \mathtt{S}_2\} \neq \emptyset$. This is not possible since otherwise at least one of the events $\mathbf{i}\colon \mathbf{i}.o : e\ \mathtt{to}\ \mathtt{R}_2$ and $\mathbf{i}\colon \mathbf{i}.o : x\ \mathtt{from}\ \mathtt{R}_1$ would not be enabled. Thus, the only possibility is $\mathsf{transF}(\mathcal{I}') = \emptyset$. This implies that $\mathcal{I}'$ has a transition with label $\surd$. Thus, $\mathbf{i}\colon o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ is enabled in $\mathcal{I}$.

Let us now consider the case in which $\mathcal{I}$ is not connected for sequence. This can happen only if $\mathcal{I} = \mathcal{I}';\mathcal{I}''$ and $\mathcal{I}'$ contained a scope which has been updated. Assume the scope has coordinator $\mathtt{R}$ and involves roles $\{\mathtt{R}_i\}_{i \in I}$. The only tricky case is when $\mathbf{i}\colon o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ is an initial interaction in $\mathcal{I}''$. Since the term was connected for sequence before the update was performed, we know that for each $\mathtt{R}_i, i \in I$, $\{\mathtt{R}_i, \mathtt{R}\} \cap \{\mathtt{R}_1, \mathtt{R}_2\} \neq \emptyset$. We have two cases: either $\mathtt{R}_1 = \mathtt{R} \vee \mathtt{R}_2 = \mathtt{R}$ or $\{\mathtt{R}_i\}_{i \in I} \subseteq \{\mathtt{R}_1, \mathtt{R}_2\}$.

In the first case, let us assume that $\mathtt{R}_1 = \mathtt{R}$ (the other case is analogous). Since inside the process of $\mathtt{R}_1$ the send primitive $\mathbf{i}: \mathbf{i}.o : e \texttt{ to } \mathtt{R}_2$ is enabled, all the auxiliary communications closing the scope have been performed, and so also the execution of the body of the scope has been completed.

The second case is analogous and follows from the fact that since the send primitive for $\mathtt{R}_1$ and the receive primitive for $\mathtt{R}_2$ are enabled, then it is not possible that either $\mathtt{R}_1$ or $\mathtt{R}_2$ (or both) is sending an auxiliary communication to $\mathtt{R}$ for closing the scope. □

The following result states that weak system bisimilarity implies weak trace equivalence.

Lemma 11. *Let $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ be a DIOC system and $\langle \mathbf{I}', \mathcal{N} \rangle$ a DPOC system. If $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \sim \langle \mathbf{I}', \mathcal{N} \rangle$ then the DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ and the DPOC system $\langle \mathbf{I}', \mathcal{N} \rangle$ are weak trace equivalent.*

Proof. The proof is by coinduction. Take a DIOC trace $\mu_1, \mu_2, \ldots$ of the DIOC system. From bisimilarity, the DPOC system has a sequence of transitions with labels $\eta_1, \ldots, \eta_k, \mu_1$ where $\eta_1, \ldots, \eta_k$ are weak transitions. Hence, the first label in the weak trace is μ_1. After the transition with label μ_1, the DIOC system and the DPOC system are again bisimilar. By coinductive hypothesis, the DPOC system has a weak trace $\mu_2, \ldots$. By composition the DPOC system has a trace $\mu_1, \mu_2, \ldots$ as desired. The opposite direction is similar. □

We can now prove our main theorem (Theorem 2, restated below) for which, given a connected well-annotated DIOC process $\mathcal{I}$ and a state Σ, the DPOC network obtained as its projection has the same behaviours of $\mathcal{I}$.

Theorem 2 (Correctness)**.** *For each initial, connected DIOC process $\mathcal{I}$, each state Σ, each set of updates $\mathbf{I}$, the DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ and the DPOC system $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ are weak trace equivalent.*

Proof. We prove that the relation $\mathcal{R}$ below is a weak system bisimulation. This implies weak trace equivalence by Lemma 11.

$$\mathcal{R} = \left\{ (\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}, \mathcal{N} \rangle) \;\middle|\; \begin{array}{l} \mathsf{upd}(\mathcal{N}) = \mathsf{proj}(\mathcal{I}, \Sigma), \\ \mathsf{events}(\mathcal{I}) \subseteq \mathsf{events}(\mathsf{compl}(\mathcal{N})), \\ \forall\, \varepsilon_1, \varepsilon_2 \in \mathsf{events}(\mathcal{I}) \;. \\ \quad \varepsilon_1 \leq_{DIOC} \varepsilon_2 \Rightarrow \varepsilon_1 \leq_{DPOC} \varepsilon_2 \vee \varepsilon_1 \leq_{DPOC} \overline{\varepsilon_2} \end{array} \right\}$$

where $\mathcal{I}$ is obtained from a well-annotated connected DIOC via 0 or more transitions and $\mathsf{upd}(\mathcal{N})$ is a well-annotated DPOC.

To ensure that proving that the relation above is a bisimulation implies our thesis, let us show that the pair $(\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle, \langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma)\rangle)$ from the theorem statement belongs to $\mathcal{R}$. Note that here $\mathcal{I}$ is well-annotated and connected, and for each such $\mathcal{I}$ we have $\mathsf{upd}(\mathsf{proj}(\mathcal{I}, \Sigma)) = \mathsf{proj}(\mathcal{I}, \Sigma)$. From Lemma 7 $\mathsf{proj}(\mathcal{I}, \Sigma)$ is well-annotated, thus $\mathsf{upd}(\mathsf{proj}(\mathcal{I}, \Sigma))$ is well-annotated. Observe that compl is the identity on $\mathsf{proj}(\mathcal{I}, \Sigma)$, thus from Lemma 5 we have that the conditions $\mathsf{events}(\mathcal{I}) \subseteq \mathsf{events}(\mathsf{compl}(\mathcal{N}))$ and $\forall \varepsilon_1, \varepsilon_2 \in \mathsf{events}(\mathcal{I})\ .\ \varepsilon_1 \leq_{DIOC} \varepsilon_2 \Rightarrow \varepsilon_1 \leq_{DPOC} \varepsilon_2 \vee \varepsilon_1 \leq_{DPOC} \overline{\varepsilon_2}$ are satisfied.

We now prove that $\mathcal{R}$ is a weak system bisimulation. To prove it, we show below that it is enough to consider only the case in which $\mathcal{N}$ (and not $\mathsf{upd}(\mathcal{N})$) is equal to $\mathsf{proj}(\mathcal{I}, \Sigma)$. Furthermore, in this case the transition of $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle$ is matched by the first transition of $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma)\rangle$.

Formally, for each $(\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle, \langle \mathbf{I}, \mathcal{N}\rangle)$ where $\mathcal{N} = \mathsf{proj}(\mathcal{I}, \Sigma)$ we have to prove the following simplified bisimulation clauses.

- if $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle$ then $\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\mu} \langle \mathbf{I}'', \mathcal{N}'''\rangle$ with $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle, \langle \mathbf{I}'', \mathcal{N}'''\rangle) \in \mathcal{R}$;

- if $\langle \mathbf{I}, \mathcal{N}\rangle \xrightarrow{\eta} \langle \mathbf{I}'', \mathcal{N}'''\rangle$ with $\eta \in \{o : \mathtt{R}_1(v) \to \mathtt{R}_2(x); \surd; \mathcal{I}; \mathsf{no\text{-}up}; \mathbf{I}''; \tau\}$ then $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\eta} \langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle, \langle \mathbf{I}'', \mathcal{N}'''\rangle) \in \mathcal{R}$.

In fact, consider a general network $\mathcal{N}_g$ with $\mathsf{upd}(\mathcal{N}_g) = \mathsf{proj}(\mathcal{I}, \Sigma)$. If $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle$, then by hypothesis $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N}_g)\rangle \xrightarrow{\mu} \langle \mathbf{I}'', \mathcal{N}'''\rangle$. From Lemma 8 case 1 there exists $\mathcal{N}''$ s.t. $\langle \mathbf{I}, \mathcal{N}_g\rangle \xrightarrow{\eta_1} \dots \xrightarrow{\eta_k}\xrightarrow{\mu} \langle \mathbf{I}'', \mathcal{N}''\rangle$ where $\eta_i \in \{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \tau\}$ for each $i \in \{1, \dots, k\}$ and $\mathsf{upd}(\mathcal{N}'') = \mathsf{upd}(\mathcal{N}''')$. By hypothesis $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}''\rangle, \langle \mathbf{I}'', \mathcal{N}'''\rangle) \in \mathcal{R}$, hence, by Definition of $\mathcal{R}$, $\mathsf{upd}(\mathcal{N}''') = \mathsf{proj}(\mathcal{I}'', \Sigma'')$, and therefore also $\mathsf{upd}(\mathcal{N}'') = \mathsf{proj}(\mathcal{I}'', \Sigma'')$. The conditions on events hold by hypothesis since function upd has no effect on DPOC events corresponding to DIOC events. Furthermore, only enabled interactions have been executed, hence dependencies between DPOC events corresponding to DIOC events are untouched.

If instead $\langle \mathbf{I}, \mathcal{N}_g\rangle \xrightarrow{\eta} \langle \mathbf{I}'', \mathcal{N}'''\rangle$ with $\eta \in \{o^? : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \surd, \mathcal{I}, \mathsf{no\text{-}up}, \mathbf{I}'', \tau\}$ then thanks to Lemma 8 we have one of the following: (2a) $\mathsf{upd}(\mathcal{N}_g) = \mathsf{upd}(\mathcal{N}''')$ and $\eta \in \{o^* : \mathtt{R}_1(v) \to \mathtt{R}_2(x), o^* : \mathtt{R}_1(X) \to \mathtt{R}_2(), \tau\}$, or (2b) $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N}_g)\rangle \xrightarrow{\eta} \langle \mathbf{I}'', \mathcal{N}''\rangle$ such that $\mathsf{upd}(\mathcal{N}''') = \mathsf{upd}(\mathcal{N}'')$. In case (2b) we have $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N}_g)\rangle \xrightarrow{\eta} \langle \mathbf{I}'', \mathcal{N}''\rangle$. Then, by hypothesis, we have $\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\eta}$

$\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}'', \mathcal{N}'' \rangle) \in \mathcal{R}$. To deduce that $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}'', \mathcal{N}''' \rangle) \in \mathcal{R}$, one can proceed using the same strategy as the case of the challenge from the DIOC above. In case (2a) the step is matched by the DIOC by staying idle, following the second option in the definition of weak system bisimilarity. The proof is similar to the one above.

Thus, we have to prove the two simplified bisimulation clauses above. The proof is by structural induction on the DIOC $\mathcal{I}$. All the subterms of a well-annotated connected DIOC are well-annotated and connected, thus the induction can be performed. We consider both challenges from the DIOC ($\rightarrow$) and from the DPOC ($\leftarrow$). The case for label $\surd$ follows from Lemma 9. The case for labels $\mathbf{I}$ is trivial. Let us consider the other labels $o : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x), \mathcal{I}, \textsf{no-up}$, and τ.

Note that no transition (at the DIOC or at the DPOC level) with one of these labels can change the set of updates $\mathbf{I}$. Thus, in the following, we will not write it. Essentially, we will use DIOC processes and DPOC networks instead of DIOC systems and DPOC systems, respectively. Note that DPOC networks also include the state, while this is not the case for DIOC processes. For DIOC processes, we assume to associate to them the state Σ, and comment on its changes whenever needed.

Case $\mathbf{1}, \mathbf{0}$ trivial.

Case i: $x@\mathtt{R} = e$ the assignment changes the global state in the DIOC, and its projection on the role $\mathtt{R}$ changes the local state of the role in the DPOC in a corresponding way.

Case i: $o : \mathtt{R}_1(e) \rightarrow \mathtt{R}_2(x)$ trivial. Just note that at the DPOC level the interaction gives rise to one send and one receive on the same prefix. Synchronisation between send and receive is performed by Rule $\lfloor^{\text{DPOC}}|_{\text{SYNCH}}\rceil$ that also removes the prefix from the label.

Case $\mathcal{I}; \mathcal{I}'$ from the definition of the projection function we have that $\mathcal{N} = \|_{\mathtt{R} \in \textsf{roles}(\mathcal{I};\mathcal{I}')} (\pi(\mathcal{I}, \mathtt{R}); \pi(\mathcal{I}', \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}}$.

$\rightarrow$ Assume that $\mathcal{I}; \mathcal{I}' \xrightarrow{\mu} \mathcal{I}''$ with $\mu \in \{o : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x); \mathcal{I}; \textsf{no-up}, \tau\}$. There are two possibilities: either (*i*) $\mathcal{I} \xrightarrow{\mu} \mathcal{I}'''$ and $\mathcal{I}'' = \mathcal{I}'''; \mathcal{I}'$ or (*ii*) $\mathcal{I}$ has a transition with label $\surd$ and $\mathcal{I}' \xrightarrow{\mu} \mathcal{I}''$.

In case (*i*) by inductive hypothesis

$$\|_{\mathtt{R} \in \textsf{roles}(\mathcal{I})} (\pi(\mathcal{I}, \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}} \xrightarrow{\mu} \mathcal{N}''' \text{ and } \textsf{upd}(\mathcal{N}''') = \|_{\mathtt{R} \in \textsf{roles}(\mathcal{I})} (\pi(\mathcal{I}''', \mathtt{R}), \Sigma'_{\mathtt{R}})_{\mathtt{R}}$$

Thus

$$\|_{\mathtt{R} \in \textsf{roles}(\mathcal{I})} (\pi(\mathcal{I}, \mathtt{R}); \pi(\mathcal{I}', \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}} \xrightarrow{\mu} \mathcal{N} \quad \text{and}$$

$$\mathsf{upd}(\mathcal{N}) = \|_{\mathtt{R} \in \mathsf{roles}(\mathcal{I})} (\pi(\mathcal{I}''', \mathtt{R}); \pi(\mathcal{I}', \mathtt{R}), \Sigma'_{\mathtt{R}})_{\mathtt{R}}$$

If $\mathsf{roles}(\mathcal{I}') \subseteq \mathsf{roles}(\mathcal{I})$ then the thesis follows. If $\mathsf{roles}(\mathcal{I}') \not\subseteq \mathsf{roles}(\mathcal{I})$ then at the DPOC level the processes in the roles in $\mathsf{roles}(\mathcal{I}') \setminus \mathsf{roles}(\mathcal{I})$ are not affected by the transition. Note however that the projection of $\mathcal{I}$ on these roles is a term composed only by $\mathbf{1}$s, and the ones corresponding to parts of $\mathcal{I}$ that have been consumed can be removed by the clean part of function upd.

In case (*ii*), $\mathcal{I}$ has a transition with label $\surd$ and $\mathcal{I}' \xrightarrow{\mu} \mathcal{I}''$. By inductive hypothesis $\mathsf{proj}(\mathcal{I}', \Sigma) \xrightarrow{\mu} \mathcal{N}''$ and $\mathsf{upd}(\mathcal{N}'') = \mathsf{proj}(\mathcal{I}'', \Sigma')$. The thesis follows since, thanks to Lemma 9, $\mathsf{proj}(\mathcal{I}; \mathcal{I}', \Sigma) \xrightarrow{\mu} \mathcal{N}$ and $\mathsf{upd}(\mathcal{N}) = \mathsf{proj}(\mathcal{I}'', \Sigma')$, possibly using the clean part of function upd to remove the $\mathbf{1}$s which are no more needed.

Note that, in both the cases, conditions on events follow by inductive hypothesis.

$\leftarrow$ Assume that

$$\mathcal{N} = \|_{\mathtt{R} \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} (\pi(\mathcal{I}, \mathtt{R}); \pi(\mathcal{I}', \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}} \xrightarrow{\eta} \|_{\mathtt{R} \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} (P_{\mathtt{R}}, \Sigma'_{\mathtt{R}})_{\mathtt{R}}$$

with $\eta \in \{o : \mathtt{R}_1(v) \to \mathtt{R}_2(x), \mathcal{I}, \mathsf{no\text{-}up}, \tau\}$. We have a case analysis on η.

If $\eta = o : \mathtt{R}_1(v) \to \mathtt{R}_2(x)$ then

$$(\pi(\mathcal{I}; \mathcal{I}', \mathtt{R}_1), \Sigma_{\mathtt{R}_1})_{\mathtt{R}_1} \xrightarrow{\overline{\mathbf{i}.o}\langle v\rangle @\mathtt{R}_2 : \mathtt{R}_1} (P_{\mathtt{R}_1}, \Sigma_{\mathtt{R}_1})_{\mathtt{R}_1} \text{ and}$$

$$(\pi(\mathcal{I}; \mathcal{I}', \mathtt{R}_2), \Sigma_{\mathtt{R}_2})_{\mathtt{R}_2} \xrightarrow{\mathbf{i}.o(x \leftarrow v) @\mathtt{R}_1 : \mathtt{R}_2} (P_{\mathtt{R}_2}, \Sigma_{\mathtt{R}_2})_{\mathtt{R}_2}$$

The two events have the same global index since they have the same index $\mathbf{i}$ (otherwise they could not synchronise) and they are both outside of any while loop (since they are enabled), hence the global index coincides with the index. Thus, they are either both from $\mathcal{I}$ or both from $\mathcal{I}'$.

In the first case we have also

$$\|_{\mathtt{R} \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} (\pi(\mathcal{I}, \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}} \xrightarrow{o:\mathtt{R}_1(v) \to \mathtt{R}_2(x)} \|_{\mathtt{R} \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} (P''_{\mathtt{R}}, \Sigma_{\mathtt{R}})_{\mathtt{R}}$$

with $P_{\mathtt{R}} = P''_{\mathtt{R}}; \pi(\mathcal{I}', \mathtt{R})$. Thus, by inductive hypothesis, $\mathcal{I} \xrightarrow{o:\mathtt{R}_1(v) \to \mathtt{R}_2(x)} \mathcal{I}''$ and $\mathsf{upd}(\|_{\mathtt{R} \in \mathsf{roles}_{\mathcal{I};\mathcal{I}'}} (P''_{\mathtt{R}}, \Sigma_{\mathtt{R}})_{\mathtt{R}}) = \mathsf{proj}(\mathcal{I}'', \Sigma)$. Hence, we have that
$\mathcal{I}; \mathcal{I}' \xrightarrow{o:\mathtt{R}_1(v) \to \mathtt{R}_2(x)} \mathcal{I}''; \mathcal{I}'$ and the
$\mathsf{upd}(\|_{\mathtt{R} \in \mathsf{roles}_{\mathcal{I};\mathcal{I}'}} (P''_{\mathtt{R}}; \pi(\mathcal{I}', \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}}) = \mathsf{proj}(\mathcal{I}''; \mathcal{I}', \Sigma)$. The thesis follows.

In the second case, we need to show that the interaction $o : \mathtt{R}_1(v) \to \mathtt{R}_2(x)$ is enabled. Assume that this is not the case. This means that there is a DIOC event ε corresponding to some construct in $\mathcal{I}$. Because of the definition of $\mathcal{R}$ ε is also a DPOC event and $\varepsilon \leq_{DPOC} \xi : \overline{o}@\mathtt{R}_2 \vee \varepsilon \leq_{DPOC} \xi : o@\mathtt{R}_1$. Hence, at least one of the two events is not minimal and the corresponding transition cannot be enabled, against our hypothesis. Therefore the interaction $o : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x)$ is enabled. Thus, $\mathcal{I}$ has a transition with label $\surd$ and $\mathcal{I}' \xrightarrow{o:\mathtt{R}_1(v)\rightarrow\mathtt{R}_2(x)} \mathcal{I}''$. Thanks to Lemma 9 then both $(\pi(\mathcal{I},\mathtt{R}_1),\Sigma_{\mathtt{R}_1})_{\mathtt{R}_1}$ and $(\pi(\mathcal{I},\mathtt{R}_2),\Sigma_{\mathtt{R}_2})_{\mathtt{R}_2}$ have a transition with label $\surd$. Thus, we have

$$(\pi(\mathcal{I}',\mathtt{R}_1),\Sigma_{\mathtt{R}_1})_{\mathtt{R}_1} \xrightarrow{\overline{\mathbf{i}.o}\langle v\rangle @\mathtt{R}_2:\mathtt{R}_1} (P_{\mathtt{R}_1},\Sigma_{\mathtt{R}_1})_{\mathtt{R}_1}$$

$$(\pi(\mathcal{I}',\mathtt{R}_2),\Sigma_{\mathtt{R}_2})_{\mathtt{R}_2} \xrightarrow{\mathbf{i}.o(x\leftarrow v)@\mathtt{R}_1:\mathtt{R}_2} (P_{\mathtt{R}_2},\Sigma_{\mathtt{R}_2})_{\mathtt{R}_2} \text{ and thus}$$

$$\mathsf{proj}(\mathcal{I}',\Sigma) \xrightarrow{o:\mathtt{R}_1(v)\rightarrow\mathtt{R}_2(x)} \|_{\mathtt{R}\in\mathsf{roles}(\mathcal{I}')} (P_{\mathtt{R}},\Sigma_{\mathtt{R}})_{\mathtt{R}}$$

The thesis follows by inductive hypothesis.

For the other cases of η, all the roles but one are unchanged. The proof of these cases is similar to the one for interaction, but simpler.

Note that in all the above cases, conditions on events follow by inductive hypothesis.

Case $\mathcal{I}|\mathcal{I}'$ from the definition of the projection function we have

$$\mathcal{N} = \|_{\mathtt{R}\in\mathsf{roles}(\mathcal{I};\mathcal{I}')} (\pi(\mathcal{I},\mathtt{R}) \mid \pi(\mathcal{I}',\mathtt{R}),\Sigma_{\mathtt{R}})_{\mathtt{R}}$$

$\rightarrow$ We have a case analysis on the rule used to derive the transition. If the transition is derived using Rule $\lfloor^{\text{DIOC}}|_{\text{PARALLEL}}\rceil$ and $\mathcal{I}|\mathcal{I}'$ can perform a transition with label μ then one of its two components can perform a transition with the same label μ and the thesis follows by inductive hypothesis. Additional roles not occurring in the term performing the transition are dealt with by the clean part of function upd. If instead the transition is derived using Rule $\lfloor^{\text{DIOC}}|_{\text{PAR-END}}\rceil$ then the thesis follows from Lemma 9.

$\leftarrow$ We have a case analysis on the label η of the transition. If $\eta = o^? : \mathtt{R}_1(v) \rightarrow \mathtt{R}_2(x)$ then a send and a receive on the same operation are enabled. The two events have the same global index since they have the same index $\mathbf{i}$ (otherwise they could not synchronise) and they are both outside of any while loop (since they are enabled), hence the global index coincides with the index. Thus, they are either both from $\mathcal{I}$ or both from $\mathcal{I}'$. The thesis follows by inductive hypothesis. For the other cases of η, only the process of one role changes. The thesis follows by inductive hypothesis. In all the cases, roles not occurring in the term performing the transition are dealt with by function upd.

Case i: $\mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}$ from the definition of projection

$$\mathcal{N} = \left(\|_{\mathtt{S}\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\smallsetminus\{\mathtt{R}\}} \begin{pmatrix} \mathbf{i}_?\colon \mathbf{i}.cnd^*_\mathbf{i} : x_\mathbf{i}\ \mathtt{from}\ \mathtt{R}; \\ \mathbf{i}\colon \mathtt{if}\ x_\mathbf{i}\ \{\pi(\mathcal{I},\mathtt{S})\}\ \mathtt{else}\ \{\pi(\mathcal{I}',\mathtt{S})\}, \Sigma_\mathtt{S} \end{pmatrix}_\mathtt{S} \right) \parallel$$
$$\Bigg(\mathbf{i}\colon \mathtt{if}\ b \left\{ \left(\prod_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\smallsetminus\{\mathtt{R}\}} \mathbf{i}_\mathtt{T}\colon \mathbf{i}.cnd^*_\mathbf{i} : \mathtt{true}\ \mathtt{to}\ \mathtt{R}' \right) ; \pi(\mathcal{I},\mathtt{R}) \right\}$$
$$\mathtt{else} \left\{ \left(\prod_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\smallsetminus\{\mathtt{R}\}} \mathbf{i}_\mathtt{F}\colon \mathbf{i}.cnd^*_\mathbf{i} : \mathtt{false}\ \mathtt{to}\ \mathtt{R}' \right) ; \pi(\mathcal{I}',\mathtt{R}) \right\}, \Sigma_\mathtt{R} \Bigg)_\mathtt{R}$$

Let us consider the case when the guard is true (the other one is analogous).

$\rightarrow$ The only possible transition from the DIOC is $\mathbf{i}\colon \mathtt{if}\ b@\mathtt{R}\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\} \xrightarrow{\tau} \mathcal{I}$. The DPOC can match this transition by reducing to

$$\mathcal{N}' = \left(\|_{\mathtt{S}\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\smallsetminus\{\mathtt{R}\}} \begin{pmatrix} \mathbf{i}_?\colon \mathbf{i}.cnd^*_\mathbf{i} : x_\mathbf{i}\ \mathtt{from}\ \mathtt{R}; \\ \mathbf{i}\colon \mathtt{if}\ x_\mathbf{i}\ \{\pi(\mathcal{I},\mathtt{S})\}\ \mathtt{else}\ \{\pi(\mathcal{I}',\mathtt{S})\} \end{pmatrix}, \Sigma_\mathtt{S} \Bigg)_\mathtt{S} \right) \parallel$$
$$\left(\left(\prod_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\smallsetminus\{\mathtt{R}\}} \mathbf{i}_\mathtt{T}\colon \mathbf{i}.cnd^*_\mathbf{i} : \mathit{true}\ \mathtt{to}\ \mathtt{R}' \right) ; \pi(\mathcal{I},\mathtt{R}), \Sigma_\mathtt{R} \right)_\mathtt{R}$$

By applying function upd we get

$$\mathsf{upd}(\mathcal{N}') = \left(\|_{\mathtt{S}\in\mathsf{roles}(\mathcal{I},\mathcal{I}')\smallsetminus\{\mathtt{R}\}} (\pi(\mathcal{I},\mathtt{S}), \Sigma_\mathtt{S})_\mathtt{S} \right) \parallel (\pi(\mathcal{I},\mathtt{R}), \Sigma_\mathtt{R})_\mathtt{R}$$

Concerning events, at the DIOC level events corresponding to the guard and to the discarded branch are removed. The same holds at the DPOC level, thus conditions on the remaining events are inherited. This concludes the proof.

$\leftarrow$ The only possible transition from the DPOC is the evaluation of the guard from the coordinator. This reduces $\mathcal{N}$ to $\mathcal{N}'$ above and the thesis follows from the same reasoning.

Case i: $\texttt{while}\ b@\texttt{R}\ \{\mathcal{I}\}$ from the definition of projection

$$\mathcal{N} = \left(\|_{\texttt{S}\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \left(\begin{array}{l} \mathbf{i}_?\colon \mathbf{i}.wb_{\mathbf{i}}^* : x_{\mathbf{i}}\ \texttt{from R}; \mathbf{i}\colon \texttt{while}\ x_{\mathbf{i}}\ \{\pi(\mathcal{I},\texttt{S}); \\ \mathbf{i}_{\texttt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \texttt{ok to R}; \mathbf{i}_?\colon \mathbf{i}.wb_{\mathbf{i}}^* : x_{\mathbf{i}}\ \texttt{from R}\} \end{array}, \Sigma_{\texttt{S}} \right)_{\texttt{S}} \right) \|$$

$$\left(\begin{array}{l} \mathbf{i}\colon \texttt{while}\ b \left\{ \begin{array}{l} \left(\prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{T}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{true}\ \texttt{to R}' \right); \pi(\mathcal{I},\texttt{R}); \\ \prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \_\ \texttt{from R}' \end{array} \right\}; \\ \prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{F}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{false}\ \texttt{to R}', \Sigma_{\texttt{R}} \end{array} \right)_{\texttt{R}}$$

$\rightarrow$ Let us consider the case when the guard is true. The only possible transition from the DIOC is $\mathbf{i}\colon \texttt{while}\ b@\texttt{R}\ \{\mathcal{I}\} \xrightarrow{\tau} \mathcal{I}; \mathbf{i}\colon \texttt{while}\ b@\texttt{R}\ \{\mathcal{I}\}$. The DPOC can match this transition by reducing to

$$\mathcal{N}' = \|_{\texttt{S}\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \left(\begin{array}{l} \mathbf{i}_?\colon \mathbf{i}.wb_{\mathbf{i}}^* : x_{\mathbf{i}}\ \texttt{from R}; \mathbf{i}\colon \texttt{while}\ x_{\mathbf{i}}\ \{\pi(\mathcal{I},\texttt{S}); \\ \mathbf{i}_{\texttt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \texttt{ok to R}; \mathbf{i}_?\colon \mathbf{i}.wb_{\mathbf{i}}^* : x_{\mathbf{i}}\ \texttt{from R}\} \end{array}, \Sigma_{\texttt{S}} \right)_{\texttt{S}} \|$$

$$\left(\begin{array}{l} \left(\prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{T}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{true}\ \texttt{to R}' \right); \pi(\mathcal{I},\texttt{R}); \\ \left(\prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \_\ \texttt{from R}' \right); \\ \mathbf{i}\colon \texttt{while}\ b \left\{ \begin{array}{l} \left(\prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{T}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{true}\ \texttt{to R}' \right); \pi(\mathcal{I},\texttt{R}); \\ \prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \_\ \texttt{from R}' \end{array} \right\}; \\ \prod_{\texttt{R}'\in \mathsf{roles}(\mathcal{I})\smallsetminus\{\texttt{R}\}} \mathbf{i}_{\texttt{F}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{false}\ \texttt{to R}', \Sigma_{\texttt{R}} \end{array} \right)_{\texttt{R}}$$

By applying function upd we get

$$\mathsf{upd}(\mathcal{N}') = \left(\|_{\mathtt{S}\in\mathsf{roles}(\mathcal{I})\smallsetminus\{\mathtt{R}\}} \left(\begin{array}{l} \pi(\mathcal{I},\mathtt{S}); \mathbf{i}_?\colon \mathbf{i}.wb_{\mathbf{i}}^* : x_{\mathbf{i}} \ \mathtt{from}\ \mathtt{R};\\ \mathbf{i}\colon \mathtt{while}\ x_{\mathbf{i}}\ \{\pi(\mathcal{I},\mathtt{S});\\ \quad \mathbf{i}_{\mathtt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \mathsf{ok}\ \mathtt{to}\ \mathtt{R};\\ \quad \mathbf{i}_?\colon \mathbf{i}.wb_{\mathbf{i}}^* : x_{\mathbf{i}}\ \mathtt{from}\ \mathtt{R}\} \end{array}, \Sigma_{\mathtt{S}} \right)_{\mathtt{S}} \right) \|$$

$$\left(\begin{array}{l} \pi(\mathcal{I},\mathtt{R});\\ \mathbf{i}\colon \mathtt{while}\ b \left\{ \begin{array}{l} \left(\prod_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{T}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{true}\ \mathtt{to}\ \mathtt{R}' \right); \pi(\mathcal{I},\mathtt{R});\\ \prod_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{C}}\colon \mathbf{i}.we_{\mathbf{i}}^* : \_\ \mathtt{from}\ \mathtt{R}' \end{array} \right\};\\ \prod_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{\mathtt{R}\}} \mathbf{i}_{\mathtt{F}}\colon \mathbf{i}.wb_{\mathbf{i}}^* : \mathit{false}\ \mathtt{to}\ \mathtt{R}' \end{array}, \Sigma_{\mathtt{R}} \right)_{\mathtt{R}}$$

exactly the projection of $\mathcal{I}; \mathbf{i}\colon \mathtt{while}\ b@\mathtt{R}\ \{\mathcal{I}\}$.

As far as events are concerned, in $\mathsf{compl}(\mathcal{N}')$ we have all the needed events since, in particular, we have already done the unfolding of the while in all the roles. Concerning the ordering, at the DIOC level, we have two kinds of causal dependencies: (1) events in the unfolded process precede the guard event; (2) the guard event precedes the events in the body. The first kind of causal dependency is matched at the DPOC level thanks to the auxiliary synchronisations that close the unfolded body (which are not removed by compl) using synchronisation and sequentiality. The second kind of causal dependency is matched thanks to the auxiliary synchronisations that start the following iteration using synchronisation, sequentiality and while.

The case when the guard evaluates to `false` is simpler.

$\leftarrow$ The only possible transition from the DPOC is the evaluation of the guard from the coordinator. This reduces $\mathcal{N}$ to $\mathcal{N}'$ above and the thesis follows from the same reasoning.

Case $\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\mathcal{I}\}$ from the definition of the projection

$$\mathcal{N} = \left(\|_{\mathtt{R}'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{\mathtt{R}\}} (\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\pi(\mathcal{I},\mathtt{R}')\}, \Sigma_{\mathtt{R}'})_{\mathtt{R}'} \right) \| \\ (\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\pi(\mathcal{I},\mathtt{R})\}\ \mathtt{roles}\ \{\mathsf{roles}(\mathcal{I})\}, \Sigma_{\mathtt{R}})_{\mathtt{R}}$$

$\rightarrow$ Let us consider the case when the scope is updated. At the DIOC level all the possible transitions have label of the form $\mathcal{I}'$ and are obtained by applying Rule $\lfloor^{\text{DIOC}}|_{\text{UP}}\rceil$. Correspondingly, at the DPOC level one applies Rule $\lfloor^{\text{DPOC}}|_{\text{LEAD-UP}}\rceil$

to the coordinator of the update, obtaining

$$\mathcal{N}' = \left(\|_{\mathtt{R}' \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{\mathtt{R}\}} \left(\mathbf{i}\colon \mathtt{scope}\ @\mathtt{R}\ \{\pi(\mathcal{I}, \mathtt{R}')\}, \Sigma_{\mathtt{R}'}\right)_{\mathtt{R}'}\right) \parallel \left(\begin{array}{l} \left(\prod_{\mathtt{R}' \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{\mathtt{R}\}} \mathbf{i}.sb_{\mathbf{i}}^{*} : \pi(\mathcal{I}', \mathtt{R}')\ \mathtt{to}\ \mathtt{R}'\right); \\ \pi(\mathcal{I}', \mathtt{R}); \\ \prod_{\mathtt{R}' \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{\mathtt{R}\}} \mathbf{i}.se_{\mathbf{i}}^{*} : \_\ \mathtt{from}\ \mathtt{R}' \end{array}, \Sigma_{\mathtt{R}}\right)_{\mathtt{R}}$$

By applying the upd function we get:

$$\mathsf{upd}(\mathcal{N}') = \left(\|_{\mathtt{R}' \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{\mathtt{R}\}} (\pi(\mathcal{I}', \mathtt{R}'), \Sigma_{\mathtt{R}'})_{\mathtt{R}'}\right) \parallel (\pi(\mathcal{I}', \mathtt{R}), \Sigma_{\mathtt{R}})_{\mathtt{R}}$$

This is exactly the projection of the DIOC obtained after applying the rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{UP}}\rceil$. The conditions on events are inherited. Observe that the closing event of the scope is replaced by events corresponding to the auxiliary interactions closing the scope. This allows us to preserve the causality dependencies also when the scope is inserted in a context.

The case of Rule $\lfloor^{\mathrm{DIOC}}|_{\mathrm{NoUP}}\rceil$ is simpler.

$\Leftarrow$ The only possible transitions from the DPOC are the ones of the coordinator of the update checking whether to apply an update or not. This reduces $\mathcal{N}$ to $\mathcal{N}'$ above and the thesis follows from the same reasoning.

$\square$

APPENDIX B

Adaptable Choreographies: Test Code

B.1 Code used for validation

B.1.1 Pipe and fork-join code

In this section, we provide the code of the AIOCJ programs used in Section 5.4. For simplicity, we show the programs fixing the number of tasks $n = 5$, and showing just two adaptation rules per scenario.

Pipe scenario. In the pipe scenario each task computes the increment function, passing the output of its computation as input to the next task.

```
 1  include startTimer, endTimer from "socket://localhost:8000"
 2  preamble{ starter: a }
 3
 4  aioc {
 5    { x@a = 0 | x@b = 0 };
 6    _r@a = startTimer( n );
 7    x@a = 1 + x; pass: a( x ) -> b( x );
 8    x@b = 1 + x; pass: b( x ) -> a( x );
 9    x@a = 1 + x; pass: a( x ) -> b( x );
10    x@b = 1 + x; pass: b( x ) -> a( x );
11    x@a = 1 + x; pass: a( x ) -> b( x );
12    _r@a = stopTimer( n )
13  }
```

Listing B.1: Pipe scenario without scopes.

Listing B.1 shows the 5 tasks that increment variable x. After each increment the variable is sent from role a to role b or vice versa. Lines 6 and 12 are calls to the Timer service (Line 1), used to log the times of execution.

```
 1  include startTimer, endTimer from "socket://localhost:8000"
 2  preamble{ starter: a }
 3
 4  aioc {
 5    { x@a = 0 | x@b = 0 };
 6    _r@a = startTimer( n );
 7    scope @a {x@a = 1 + x; pass: a( x ) -> b( x )} prop {N.x = 1};
 8    scope @b {x@b = 1 + x; pass: b( x ) -> a( x )} prop {N.x = 2};
 9    scope @a {x@a = 1 + x; pass: a( x ) -> b( x )} prop {N.x = 3};
10    scope @b {x@b = 1 + x; pass: b( x ) -> a( x )} prop {N.x = 4};
11    scope @a {x@a = 1 + x; pass: a( x ) -> b( x )} prop {N.x = 5};
12    _r@a = stopTimer( n )
13  }
```

Listing B.2: Pipe scenario with scopes.

Listing B.2 shows the 5 scopes enclosing the tasks of Listing B.1. Notably, each scope has a unique property `N.x` assigned incrementally. This property is used to identify uniquely each scope.

Rules of the pipe scenario. Listing B.3 shows two rules used in the pipe scenario. The adapted behaviour increments the variable `x` by 2. In particular, the rule at Lines 1-4 applies to the scope at Line 7 in Listing B.2, while the subsequent rule (Lines 5-8) applies to the scope at Line 8.

```
1  rule {
2    on { N.x == 1 }
3    do { x@a = 2 + x; pass: a( x ) -> b( x ) }
4  }
5  rule {
6    on { N.x == 2 }
7    do { x@b = 2 + x; pass: b( x ) -> a( x ) }
8  }
```

Listing B.3: Rules for pipe scenario.

Fork-join scenario. In the fork-join scenario tasks run in parallel. Each task takes one character of a message and replaces it with the next character in the alphabet. The program makes use of an external service to retrieve the character of the message, get the next letter, and store the new character in the message.

```
include start, end from "socket://localhost:8000"
include getNthChar, getNext, setNthChar from "socket://localhost:8001"

preamble{ starter: a }

aioc {
  _r@a = start( n );
  {
     {  l0@a = getNthChar( 0 ); l0@a = getNext( l0 ); _r@a = setNthChar( 0, l0 ) }
   | {  l1@b = getNthChar( 1 ); l1@b = getNext( l1 ); _r@b = setNthChar( 1, l1 ) }
   | {  l2@a = getNthChar( 2 ); l2@a = getNext( l2 ); _r@a = setNthChar( 2, l2 ) }
   | {  l3@b = getNthChar( 3 ); l3@b = getNext( l3 ); _r@b = setNthChar( 3, l3 ) }
   | {  l4@a = getNthChar( 4 ); l4@a = getNext( l4 ); _r@a = setNthChar( 4, l4 ) }
  };
  _r@a = end( n )
}
```

Listing B.4: Fork-join scenario without scopes.

Listing B.4 shows the 5 tasks of the fork-join scenario. Each task uses a dedicated variable (`l0, l1, ..., ln`) to ensure thread-safety of the computation.

```
include start, end from "socket://localhost:8000"
include getNthChar, getNext, setNthChar from "socket://localhost:8001"

preamble{ starter: a }

aioc {
  _r@a = startTimer( n ); {
    scope @a {
      l0@a = getNthChar( 0 ); l0@a = getNext( l0 ); _r@a = setNthChar( 0, l0 ) }
      prop { N.char = 0 } |
    scope @b {
      l1@b = getNthChar( 1 ); l1@b = getNext( l1 ); _r@b = setNthChar( 1, l1 ) }
      prop { N.char = 1 } |
    scope @a {
      l2@a = getNthChar( 2 ); l2@a = getNext( l2 ); _r@a = setNthChar( 2, l2 ) }
      prop { N.char = 2 } |
    scope @b {
      l3@b = getNthChar( 3 ); l3@b = getNext( l3 ); _r@b = setNthChar( 3, l3 ) }
      prop { N.char = 3 } |
    scope @a {
      l4@a = getNthChar( 4 ); l4@a = getNext( l4 ); _r@a = setNthChar( 4, l4 ) }
      prop { N.char = 4 }
  };
  _r@a = stopTimer( n )
}
```

Listing B.5: Fork-join scenario with scopes.

Likewise the pipe scenario, the scopes in Listing B.5 enclose the tasks of Listing B.4. Also in this scenario each scope has a unique property N.char that identifies it uniquely.

Rules of the fork-join scenario. The adapted behaviour introduces the new function GetDoubleNext that gets a character and returns the character two positions after it in the alphabet. Listing B.6 shows two rules used in the fork scenario. The rule at Lines 1-6 applies to the scope at Lines 8-10 of Listing B.5, while the rule at Lines 7-12 applies to the scope at Lines 11-13.

```
1  rule {
2  include getNthChar, getDoubleNext, setNthChar from "socket://localhost:8001"
3    on { N.char == 0 }
4    do {  l0@b = getNthChar( 0 ); l0@b = getDoubleNext( l0 );
5      _r@b = setNthChar( 0, l0 ) }
6  }
7  rule {
8  include getNthChar, getDoubleNext, setNthChar from "socket://localhost:8001"
9    on { N.char == 1 }
10   do {  l1@b = getNthChar( 1 ); l1@b = getDoubleNext( l1 );
11     _r@b = setNthChar( 1, l1 ) }
12 }
```

Listing B.6: Rules of pipe scenario.

B.1.2 AIOCJ programs used for benchmarking primitives

```
include start, end from "socket://localhost:8000"
preamble { starter: a }
aioc {
  _r@a = start( "assignment" );
  x@a = 1;
  _r@a = end( "assignment" );

  _r@a = start( "interaction" );
  pass: a( x ) -> b( x );
  _r@a = end( "interaction" );

  _r@a = start( "if statement" );
  if ( x == 1 )@a { skip };
  _r@a = end( "if statement" )
}
```

Listing B.7: Code for benchmarking assignment, interaction, and the if statement.

```
include start, end
  from "socket://localhost:8000"

```

```
preamble { starter: a }

aioc {
  _r@a = start( "scope" );
  scope @a{ skip }
  prop { N.scope_name = "applicable"};
  _r@a = end( "scope" )
}
```

Listing B.8: Code for benchmarking the scope primitive.

Listings B.7 and B.8 show the code used to measure the performances of primitives of AIOCJ. We tested the performances of the scope (Listing B.8) under 7 contexts:

1. without adaptation servers;
2. with 1 adaptation server, but without rules;
3. with 1 adaptation server and 1 matching rule;
4. with 1 adaptation server, 50 rules but none matching;
5. with 1 adaptation server, 50 rules, and 1 matching;
6. with 1 adaptation server, 100 rules but none matching;
7. with 1 adaptation server, 100 rules, and 1 matching;

```
1  rule {
2    on { N.scope_name == "applicable" }
3    do { skip }
4  }
5  rule {
6    on { N.scope_name == "non-applicable" }
7    do { skip }
8  }
```

Listing B.9: Rules used for benchmarking scopes.

Listing B.9 shows the two kinds of rules used for testing performances of Listing B.8. The first one (Lines 1-4) is the matching rule used in contexts 3, 5, and 7. The second one (Lines 5-8) is a non matching rule present, in different copies, in contexts 4, 5, 6, and 7.

APPENDIX C

Adaptable Choreographies: Models of Adaptation

To model in AIOCJ the advanced examples below, we introduce the keyword `roles` for AIOC `scope`s. Consider the code below:

```
scope @Role1 {
  /* code */
} roles { Role2 }
```

the keyword `roles` attached to the `scope` means that `Role2` passively participates to the `scope`, i.e., it has no interactions. At runtime, this feature enables the application of updates that comprise `Role2` among its roles. In this way, scopes comprise passive roles that can play an important part in the updated code, e.g., a `Logger` that is activated by an update.

This feature does not detach the implementation of AIOCJ from the theoretical model of Dynamic Choreographies, yet it allows to model advanced behaviours like the ones below.

C.1 A distributed adaptive document system

Here we provide an account of how we can model aspect orientation in AIOCs.

The example proposed here is taken from [118] and gives a pragmatic assessment on the expressiveness of the AIOC language with respect to distributed dynamic aspect-oriented systems.

In the scenario proposed in [118], the authors describe the classic server-client interaction in which the server serves documents to Clients. The server implements three “methods” to interact with documents, namely:

- Search, which allows a Client to search a certain string on a document;
- Read, which returns the content of a document;
- Write, which modifies the content of a document.

```
include search, read, write from "socket://localhost:8002"

preamble {
  starter: Client
}

aioc {
  continue@Client = true;

  while( continue )@Client{
    r@Client = getInput( "Select operation: (S)earch, " +
      "(R)ead, (W)rite, or (E)xit" );

    if( r == "S" )@Client{
      s@Client = getInput( "Insert search expression" );
      // for logging purposes
      scope @Client{ skip } prop { N.scopename = "log" }
      roles { Logger };
      // adapt the search protocol
      scope @Logger{
        search: Client( s ) -> Server( s );
        res@Server = search( s );
        response: Server( res ) -> Client( res )
      } prop{ N.scopename = "search" }
      roles { Balancer, Rserver };
      _r@Client = show( res )
    };

    if( r == "R" )@Client{
      s@Client = getInput( "Insert the page to read" );
      // for logging purposes
      scope @Client{ skip } prop { N.scopename = "log" }
      roles { Logger };
      // adapt the read protocol
      scope @Logger{
        read: Client( s ) -> Server( s );
        res@Server = read( s );
        response: Server( res ) -> Client( res )
      } prop { N.scopename = "read" }
      roles { Balancer, Rserver };
      _r@Client = show( res )
    };

    if( r == "W" )@Client{
      s@Client = getInput( "Insert the page and the " +
        "content of the modification (page, content)" );
      scope @Client{ skip } prop { N.scopename = "log" }
      roles { Logger };
      scope @Logger{
        write: Client( s ) -> Server( s );
        res@Server = write( s );
        response: Server( res ) -> Client( res )
      } prop{ N.scopename = "write" }
      roles { Balancer, Rserver };
      _r@Client = show( res )
    };

    if( r == "E" )@Client{
      continue@Client = false
    };

    if( r != "S" and r != "E" and
      r != "R" and r != "W" )@Client{
      r@Client = show( "Insert command not valid" )
    }
  }
}

rule {
  include log, getLoadBalancing from "socket://localhost:8000"
  on { N.scopename == "log" }
  do {
    log: Client( r ) -> Logger( log );
    _r@Logger = log( log );
    if( log == "S" )@Logger {
      search_balance@Logger = getLoadBalancing( log )
    };
    if( log == "R" )@Logger {
      read_balance@Logger = getLoadBalancing( log )
    }
  }
}

rule {
  include getServer from "socket://localhost:8001"
  include search from "socket://localhost:8002"
  include rSearch from "socket://localhost:8003"
  on { N.scopename == "search" and search_balance == true }
  do {
    ser@Balancer = getServer( "S" );
    if( ser )@Balancer{
      search: Client( s ) -> Server( s );
      res@Server = search( s );
      response: Server( res ) -> Client( res )
    } else {
      search: Client( s ) -> Rserver( s );
      res@Rserver = rSearch( s );
      response: Rserver( res ) -> Client( res )
    }
  }
}

rule {
  include getServer from "socket://localhost:8001"
  include read from "socket://localhost:8002"
  include rRead from "socket://localhost:8003"
  on { N.scopename == "read" and read_balance == true }
  do {
    ser@Balancer = getServer( "R" );
    if( ser )@Balancer{
      read: Client( s ) -> Server( s );
      res@Server = read( s );
      response: Server( res ) -> Client( res )
    } else {
      read: Client( s ) -> Rserver( s );
      res@Rserver = rRead( s );
      response: Rserver( res ) -> Client( res )
    }
  }
}


rule {
  include getServer from "socket://localhost:8001"
  include write from "socket://localhost:8002"
  include rWrite from "socket://localhost:8003"
  on {
    N.scopename == "write" and
    ( read_balance == true or search_balance == true )
  }
  do {
    write: Client( s ) -> Server( s );
    res@Server = write( s );
    // propagates changes to Rserver
    write: Server( s ) -> Rserver( s );
    res@Rserver = rWrite( s );
    ok: Rserver() -> Server();
    res@Server = "[Synchronised] " + res;
    response: Server( res ) -> Client( res )
  }
}
```

C.1.1 Pointcuts

First, we model one of the basic concepts of AOP which is pointcut. A pointcut describes a set of join points. Each time the execution of a program reaches one of the join points relative to a pointcut, a piece of code associated with the pointcut is executed. We implement pointcuts as empty scopes defined before each operation. The empty scopes make use of the feature `roles{...}` that lets us to specify additional roles in the scope. Additional roles do not take an active part in the scope they are added in, but their behaviour relative to that scope can be changed by an adaptation rule. In the example, if the adaptation rule `N.scope_name = log` applies, we allow a Logger to count the number of interactions relative to search, read, and write operations.

```
scope @Client{ skip }
prop { N.scopename = "log" }
roles { Logger };
```

```
rule {
 include log, getLoadBalancing
  from "socket://localhost:8000"
 on { N.scopename == "log" }
 do {
  log: Client( r ) -> Logger( log );
  _r@Logger = log( log );
  if( log == "S" )@Logger {
   search_balance@Logger =
    getLoadBalancing( log )
  };
  if( log == "R" )@Logger {
   read_balance@Logger =
    getLoadBalancing( log )
  }
 }
}
```

C.1.2 Dynamic wrappers

Dynamic wrappers, as defined in JAC, are software entities that allow the modification or the enhancement of the semantics of some base objects. They provide the ability to add code before, after, or around existing methods.

In the proposed example, the authors suggest to apply distributed aspects to avoid the overload of a single server. Namely, they introduce aspects that manage distribution of documents, load-balancing, and coherence among copies of the same document.

As for pointcuts, we make use of the feature `roles{...}` to include in the interaction of each operation a Balancer and an additional server (Rserver). Then, in the adaptation rules the Balancer routes the operations between the two servers to balance the load. Furthermore, in order to preserve coherence among documents, each time a write operation is executed, the server that applied the modifications notifies the other to synchronise the state of the copy of the document.

C.2 ContextChat

In this section we model context orientation in the AIOC language.

The scenario proposed here is taken from [73] in which the authors introduce context-oriented programming within the actor model of the Erlang language. Below we provide a pragmatic assessment on the expressiveness of the AIOC language with respect to distributed dynamic context-oriented systems.

Context-oriented evolve from aspect-oriented programming by introducing mechanisms to enforce the coherent application of aspects.

In [73] the authors propose the example of ContextChat to highlight the features of context-oriented programming applied to the Erlang actor model. In particular the authors introduce a fixed stack of adaptation procedures. The slots that compose said stack describe the behaviour of adaptable procedures (also called variations):

- activatable slots contain one variation which can be independently activated;
- switch slots contain several mutually exclusive variations;
- free slots contain an undefined single variation which can be acquired and assigned from other programs.

In ContextChat connected clients exchange messages in real time. The system implements advanced features as context variants:

- offline reception: if an user goes offline, other users can still send messages to him. The system saves offline messages and shows them to the recipient when s/he returns online. Offline and online statuses are described in a switch slot, therefore mutually exclusive;
- backup feature: the user can activate (activatable slot) a backup mode that saves all messages (sent and received) on a remote server;
- tracing feature: the system can activate (activatable slot) a tracing mode that collects information on client communication to handle network communication in a more efficient way (e.g., using internal messages in place of network ones);
- text effects: users can submit (free slot) customisable filters to the messages (e.g., for text emphasising, emoticons, etc.).

Modelling contexts with scopes

Let us analyse how the different behaviours described by activatable, switch, and free slots map to AIOCJ scopes.

Activatable and free slots coalesce in the same type of scopes since in AIOCJ the source of variations are rules that can be add and removed at runtime (i.e., the set of running Adaptation Servers, cf. § 5.1). Activation of such variations is based on variables belonging to the environment (prefixed with E.), to the scope, which are non-functional (prefixed with N.), and from the status of the coordinator of the adaptation.

Switch slots require a choice in the design of the adaptation with respect to the basic and the altered mutually exclusive behaviours.

Referring to the offline/online statuses of ContextChat, we assume that "online" is the basic behaviour which alternates with the "offline" status following the preferences of the User. Hence, the resulting AIOC contains the basic behaviour (online) and, depending on the availability of Adaptation Servers and the User's status, mutually exclusive behaviours will overwrite the basic one. Since after adaptation scope boundaries are removed, each time the basic behaviour adapts only the matching rule overwrites its behaviour, excluding any other applicable variant.

To keep the example simple and since activatable slots directly map to scopes, we implement the example of ContextChat with the online/offline variants and the free slot for text emphasising.

We report below the full code of the AIOC and the adaptation rules.

```
preamble { starter: User1 }

aioc {
 { name@User2 = "User2"     | name@User1 = "User1"
  | continue@User1 = true    | continue@User2 = true
  | status@User1 = "online"    | status@User2 = "online"
 };
 {
  while( continue == true )@User1{
   scope @User1{
   msg@User1 = getInput( name + ": insert a message" )
   } prop { N.scope_name = "get_msg" };
   scope @User1{
    send: User1( msg ) -> User2( msg );
    scope @User2{
     _r@User2 = show( name + ": " + msg )
    } prop { N.scope_name = "display_msg" }
   |
   c@User1 = getInput( name + ": do you want to continue [y/n]? ");
   if( c == "n" )@User1{
    continue@User1 = false | continue@User2 = false
   }
  } prop { N.scope_name = "status_switch_user1" }
  }
  |
  while( continue == true )@User2{
   scope @User2 {
    msg@User2 = getInput( name + ": insert a message" )
   } prop { N.scope_name = "get_msg" };
   scope @User2{
    send: User2( msg ) -> User1( msg );
    scope @User1{
     _r@User1 = show( name + ": " + msg )
   } prop { N.scope_name = "display_msg" }
   |
   c@User2 = getInput( name + ": do you want to continue [y/n]? ");
   if( c == "n" )@User2{
    continue@User1 = false | continue@User2 = false
    }
   } prop { N.scope_name = "status_switch_user2" }
  }
 }
}
```

```
// the rule applies on any scope with
// property scope_name equal to "get_msg"
rule {
 on { N.scope_name == "get_msg" }
 do {
  msg@User1 = getInput( name + ": insert a message or " +
   "write 'exit' to go offline" );
  if( msg == "exit" )@User1{ status@User1 = "offline" }
 }
}

// the rule applies when the User is offline. It acts as a switch
// and, when the User returns online, displays the stored messages
rule {
 on { status == "offline" and N.scope_name == "status_switch_user1" }
 do {
  _r@User1 = show( name + ": click ok to go online" );
  status@User1 = "online";
  if( stored_msg_count > 0 )@User1 {
   _r@User1 = show( name + " received: " + stored_msg );
   stored_msg@User1 = ""; stored_msg_count@User1 = 0
  }
 }
}

// the rule applies when the User is offline.
// It stores the message for future display
rule {
 on { status == "offline" and N.scope_name == "display_msg" }
 do {
  if( stored_msg_count > 0 )@User1{
   stored_msg@User1 = stored_msg + "; " + msg
  } else { stored_msg@User1 = msg };
  stored_msg_count@User1 = stored_msg_count + 1
 }
}

// the rule applies when the User is online. It adds text effects to
// the displayed message we apply text effect only on online display
rule {
 on { status == "online" and N.scope_name == "display_msg" and
    N.emph_effect != "applied" }
 do {
  msg@User1 = "*" + msg + "*";
  scope @User1{
   _r@User1 = show( msg )
  } prop { N.scope_name = "display_msg", N.emph_effect = "applied" }
 }
}
```

```
// Rules below mirror the ones above but apply only on User2
rule {
 on { N.scope_name == "get_msg" }
 do {
  msg@User2 = getInput( name + ": insert a message or " +
   "write 'exit' to go offline" );
  if( msg == "exit" )@User2{
   status@User2 = "offline"
  }
 }
}

rule {
 on { status == "offline" and
    N.scope_name == "status_switch_user2" }
 do {
  _r@User2 = show( name + ": click ok to go online" );
  status@User2 = "online";
  if( stored_msg_count > 0 )@User2 {
   _r@User2 = show( name + " received: " + stored_msg );
   stored_msg@User2 = "";
   stored_msg_count@User2 = 0
  }
 }
}

rule {
 on { status == "offline" and N.scope_name == "display_msg" }
 do {
  if( stored_msg_count > 0 )@User2{
   stored_msg@User2 = stored_msg + "; " + msg
  } else {
   stored_msg@User2 = msg
  };
  stored_msg_count@User2 = stored_msg_count + 1
 }
}

rule {
 on { status == "online" and N.scope_name == "display_msg"
    and N.emph_effect != "applied" }
 do {
  msg@User2 = "*" + msg + "*";
  scope @User2{
   _r@User2 = show( msg )
  } prop { N.scope_name = "display_msg",
      N.emph_effect = "applied" }
 }
}
```

C.2.1 Online/Offline switch

Since Online status is the basic behaviour, the choreography implements it as its default. However, the programmer can specify that the procedure for acquire a User's message can be changed (below). In this case the rule always applies and augments the functionality of the interaction with the User which can either send a message or go offline by writing "exit".

```
scope @User1{
  msg@User1 = getInput( name + ": insert a message" )
} prop { N.scope_name = "get_msg" }
```

```
// rule applies on any scope with property
// scope_name equal to "get_msg"
rule {
  on { N.scope_name == "get_msg" }
  do { msg@User1 = getInput( name +
    ": write a message or 'exit' to go offline" );
    if( msg == "exit" )@User1{
      status@User1 = "offline"
 }}}
```

Note that the scope, led and participated only by `User1`, indicates as `N.scope_name = "get_msg"` its only non-functional property. `User2` shows an equivalent scope (with the non-functional property `N.scope_name = "get_msg"`), yet the adaptation mechanisms of AIOCJ apply the corresponding rule to the right scope as it checks applicability of rules on the participants to the scope. Since in this case the leaders are also the only participants to the scope, no further properties are needed to ensure coherent application of adaptation.

Subsequently, the scope with `N.scope_name = "status_switch_user1"` overwrites the sending procedure. The rules procedure waits for the user to return online and displays the messages received in the meanwhile. Notably the rule applies only if the status variable of the leaders is set to "offline".

```
scope @User1{
 send: User1( msg ) -> User2( msg );
{
 scope @User2{
  _r@User2 = show( name + ": " + msg )
 } prop { N.scope_name = "display_msg" }
 |
 c@User1 = getInput( name + ": do you want to continue [y/n]? ");
 if( c == "n" )@User1{
   continue@User1 = false | continue@User2 = false
  }
 }
}
prop { N.scope_name = "status_switch_user1" }
```

```
rule {
  on { status == "offline" and
  N.scope_name == "status_switch_user1" }
  do { _r@User1 = show( name + ": click ok to go online" );
    status@User1 = "online";
    if( stored_msg_count > 0 )@User1 {
      _r@User1 = show( name + " received: " + stored_msg );
      stored_msg@User1 = "";
      stored_msg_count@User1 = 0
    }
  }
}
```

Finally, if the user is online and sends a message, the addressee (User2, in case of User1) enters the scope with N.scope_name = "display_msg". The scope serves a twofold purpose: in case the addressee is offline, the overwriting rule (which is led and participated by User2) stores the message into a storage variable for future visualisation, if otherwise User2 is online the text-effect rule applies. Notably text-effects can stack as a the adaptation code of the rule bears a nested scope with the same properties of the adapted scope, beside the one N.emph_effect = "applied" which prevents the infinite application of the same rule.

```
rule {
 on { status == "offline" and N.scope_name == "display_msg" }
 do {
  if( stored_msg_count > 0 )@User1{
   stored_msg@User1 = stored_msg + "; " + msg
  } else {
   stored_msg@User1 = msg
  };
  stored_msg_count@User1 = stored_msg_count + 1
 }
}

rule {
 on { status == "online" and N.scope_name == "display_msg" and
      N.emph_effect != "applied" }
 do {

  msg@User1 = "*" + msg + "*";
  scope @User1{
   _r@User1 = show( msg )
  } prop { N.scope_name = "display_msg",
             N.emph_effect = "applied" }
 }
}
```

APPENDIX D

Applied Choreographies: Additional Material

D.1 Applied Choreographies

$$\frac{\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset}{\eta; \eta' \simeq_{\mathsf{C}} \eta'; \eta} \lfloor^{\text{CS}}|_{\text{EtaEta}}\rceil \qquad \frac{\mathsf{p} \notin \mathsf{pn}(\eta)}{\begin{array}{c}\mathsf{if}\ \mathsf{p}.e\ \{\eta; C_1\}\ \mathsf{else}\ \{\eta; C_2\} \\ \simeq_{\mathsf{C}}\ \eta; \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}\end{array}} \lfloor^{\text{CS}}|_{\text{EtaCnd}}\rceil$$

$$\frac{\mathsf{q} \notin \mathsf{pn}(\eta)}{\begin{array}{c}k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{q}[\mathtt{B}].\{o_i(x_i); \eta; C_i\}_{i \in I} \\ \simeq_{\mathsf{C}}\ \eta; k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I}\end{array}} \lfloor^{\text{CS}}|_{\text{EtaRcv}}\rceil$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c}k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{p}[\mathtt{B}].\{o_i(x_i); k' : \mathtt{C} \mathbin{\texttt{->}} \mathsf{q}[\mathtt{D}].\{o'_{ij}(x'_{ij}); C_{ij}\}_{j \in J}\}_{i \in I} \\ \simeq_{\mathsf{C}} k' : \mathtt{C} \mathbin{\texttt{->}} \mathsf{q}[\mathtt{D}].\{o'_j(x'_j); k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{p}[\mathtt{B}].\{o_{ij}(x_{ij}); C_{ij}\}_{i \in I}\}_{j \in J}\end{array}} \lfloor^{\text{CS}}|_{\text{RcvRcv}}\rceil$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c}\mathsf{if}\ \mathsf{p}.e\ \{\mathsf{if}\ \mathsf{q}.e'\ \{C_1\}\ \mathsf{else}\ \{C_2\}\}\ \mathsf{else}\ \{\mathsf{if}\ \mathsf{q}.e'\ \{C'_1\}\ \mathsf{else}\ \{C'_2\}\} \\ \simeq_{\mathsf{C}} \mathsf{if}\ \mathsf{q}.e'\ \{\mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C'_1\}\}\ \mathsf{else}\ \{\mathsf{if}\ \mathsf{p}.e\ \{C_2\}\ \mathsf{else}\ \{C\prime_2\}\}\end{array}} \lfloor^{\text{CS}}|_{\text{CndCnd}}\rceil$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c}k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{p}[\mathtt{B}].\{o_i(x_i); \mathsf{if}\ \mathsf{q}.e\ \{C_{i1}\}\ \mathsf{else}\ \{C_{i2}\}\}_{i \in I} \quad \simeq_{\mathsf{C}} \\ \mathsf{if}\ \mathsf{q}.e\ \{k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{p}[\mathtt{B}].\{o_i(x_i); C_{i1}\}_{i \in I}\}\ \mathsf{else}\ \{k : \mathtt{A} \mathbin{\texttt{->}} \mathsf{p}[\mathtt{B}].\{o_i(x_i); C_{i2}\}_{i \in I}\}\end{array}} \lfloor^{\text{CS}}|_{\text{RcvCnd}}\rceil$$

Figure D.1: Choreography Calculus, swap relation $\simeq_{\mathsf{C}}$

$$\frac{\Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \mathsf{init}\big(\,\widetilde{\mathsf{r}[\mathtt{C}]}, k, G\,\big) \vdash C \quad \widetilde{\mathsf{r}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash \textbf{start}\; k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C} \; \lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$$

$$\frac{\Gamma, \mathsf{p} : k[\mathtt{A}], k[\mathtt{A}] : [\![G]\!]_{\mathtt{A}} \vdash C \quad \Gamma \vdash \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle}{\Gamma \vdash \textbf{req}\; k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C} \; \lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil$$

$$\frac{\tilde{l} \subseteq \tilde{l}' \quad \Gamma, \tilde{l}' : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle, \mathsf{init}\big(\,\widetilde{\mathsf{q}[\mathtt{C}]}, k, G\,\big) \vdash C}{\Gamma, \tilde{l}' : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \vdash \textbf{acc}\; k : \widetilde{l.\mathsf{q}[\mathtt{C}]}; C} \; \lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil$$

$$\frac{\Gamma \vdash \mathsf{p}.e : \textbf{bool} \quad \Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash \mathsf{if}\; \mathsf{p}.e\; \{C_1\}\; \mathsf{else}\; \{C_2\}} \; \lfloor^{\mathrm{T}}|_{\textsc{Cond}}\rceil$$

$$\frac{j \in I \quad \Gamma \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \quad \Gamma \vdash \mathsf{p}.e : U_j \quad \Gamma, \mathsf{q}.x : U_j, k[\mathtt{A}] : T_j, k[\mathtt{B}] : T'_j \vdash C}{\Gamma, k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); T_i\}_{i \in I}, k[\mathtt{B}] : ?\mathtt{A}.\{o_i(U_i); T'_i\}_{i \in I} \vdash k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathsf{q}[\mathtt{B}].o_j(x); C} \; \lfloor^{\mathrm{T}}|_{\textsc{Com}}\rceil$$

$$\frac{j \in I \quad \Gamma \vdash \mathsf{p} : k[\mathtt{A}] \quad \Gamma \vdash \mathsf{p}.e : U_j \quad \Gamma, k[\mathtt{A}] : T_j \vdash C}{\Gamma, k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); T_i\}_{i \in I} \vdash k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o_j; C} \; \lfloor^{\mathrm{T}}|_{\textsc{Send}}\rceil$$

$$\frac{\Gamma \vdash \mathsf{q} : k[\mathtt{B}] \quad \forall j \in I.\, \Gamma, \mathsf{q}.x_j : U_j, k[\mathtt{B}] : T_j \vdash C_j}{\Gamma, k[\mathtt{B}] : ?\mathtt{A}.\{o_i(U_i); T_i\}_{i \in I} \vdash k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].\{o_j(x_j); C_j\}_{j \in I \cup J}} \; \lfloor^{\mathrm{T}}|_{\textsc{Recv}}\rceil$$

$$\frac{\Gamma, X : \Gamma' \vdash C \quad \Gamma', X : \Gamma' \vdash C' \quad \Gamma'|_{\mathsf{locs}} \subseteq \Gamma \quad \tilde{\mathsf{p}} = \mathsf{pn}(C')}{\Gamma \vdash \mathsf{def}\; X\langle\tilde{\mathsf{p}}\rangle = C'\; \mathsf{in}\; C} \; \lfloor^{\mathrm{T}}|_{\textsc{Def}}\rceil$$

$$\frac{\Gamma_i \vdash C_i \quad \Gamma_2 \vdash C_2}{\Gamma_1 \circ \Gamma_2 \vdash C_1 \mid C_2} \; \lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil \qquad \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash \mathbf{0}} \; \lfloor^{\mathrm{T}}|_{\textsc{End}}\rceil \qquad \frac{\mathsf{end}(\Gamma) \quad \Gamma'' \subseteq \Gamma'}{\Gamma, \Gamma'', X\langle\tilde{\mathsf{p}}\rangle : \Gamma' \vdash X\langle\tilde{\mathsf{p}}\rangle} \; \lfloor^{\mathrm{T}}|_{\textsc{Call}}\rceil$$

Figure D.2: Choreography Calculus - Typing Rules

D.2 Typing

In addition to the description in § 6.3 we comment the additional rules in Figure D.2.

In Rule $\lfloor^{\mathrm{T}}|_{\textsc{End}}\rceil$ end holds if the protocols for all sessions have terminated (i.e., all local typings have type end). In Rule $\lfloor^{\mathrm{T}}|_{\textsc{Def}}\rceil$ the condition $\Gamma'|_{\mathsf{locs}} \subseteq \Gamma$ checks that the body of the recursive procedure does not introduce unexpected services ($\Gamma'|_{\mathsf{locs}}$ returns all service typings in Γ').

$$\frac{\{\mathtt{A},\mathtt{B}\} \cap \{\mathtt{C},\mathtt{D}\} = \emptyset}{\begin{array}{c}\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); \mathtt{C} \texttt{ -> } \mathtt{D}.\{o'_j(U'_j); G_{ij}\}_{j\in J}\}_{i\in I} \\ \simeq_{\mathsf{G}} \mathtt{C} \texttt{ -> } \mathtt{D}.\{o'_j(U'_j); \mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_{ij}\}_{i\in I}\}_{j\in J}\end{array}} \lfloor^{\mathrm{GS}}|_{\textsc{ComCom}}\rceil$$

$$\frac{\{\mathtt{A},\mathtt{B}\} \cap \{\mathtt{D}\} = \emptyset}{\begin{array}{c}\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); \mathtt{C} \rightsquigarrow \mathtt{D}.o(U); G_i\} \\ \simeq_{\mathsf{G}} \mathtt{C} \rightsquigarrow \mathtt{D}.o(U); \mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\}\end{array}} \lfloor^{\mathrm{GS}}|_{\textsc{ComRecv}}\rceil$$

$$\frac{\mathtt{B} \neq \mathtt{D}}{\begin{array}{c}\mathtt{A} \rightsquigarrow \mathtt{B}.o(U); \mathtt{C} \rightsquigarrow \mathtt{D}.o'(U); G \\ \simeq_{\mathsf{G}} \mathtt{C} \rightsquigarrow \mathtt{D}.o'(U); \mathtt{A} \rightsquigarrow \mathtt{B}.o(U); G\end{array}} \lfloor^{\mathrm{GS}}|_{\textsc{RecvRecv}}\rceil$$

Figure D.3: Global types, Swap Relation $\simeq_{\mathsf{G}}$.

$$[\![\mathtt{C} \texttt{ -> } \mathtt{D}.\{o_i(U_i); G_i\}]\!]^{\mathtt{A}}_{\mathtt{B}} = \begin{cases} \mathsf{end} & \text{if } \mathtt{C} = \mathtt{A} \ \wedge \ \mathtt{D} = \mathtt{B} \\ \bigsqcup_i [\![G_i]\!]^{\mathtt{A}}_{\mathtt{B}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{C} \rightsquigarrow \mathtt{D}.o(U); G]\!]^{\mathtt{A}}_{\mathtt{B}} = \begin{cases} ?\mathtt{A}.o(U); [\![G]\!]^{\mathtt{A}}_{\mathtt{B}} & \text{if } \mathtt{C} = \mathtt{A} \wedge \ \mathtt{D} = \mathtt{B} \\ [\![G]\!]^{\mathtt{A}}_{\mathtt{B}} & \text{otherwise} \end{cases}$$

$$[\![\mathbf{t}]\!]^{\mathtt{A}}_{\mathtt{B}} = \mathsf{end} \qquad [\![\mathsf{end}]\!]^{\mathtt{A}}_{\mathtt{B}} = \mathsf{end} \qquad [\![\mathsf{rec}\ \mathbf{t}; G]\!]^{\mathtt{A}}_{\mathtt{B}} = \mathsf{end}$$

Figure D.4: Choreography Calculus - Buffer Type Projection

D.3 Endpoint Projection

We report the complete definition of choreography annotation, process projection, merging function, and grouping function, respectively in Figure D.5, Figure D.6, Figure D.7, and Figure D.8.

$$@C = \emptyset @C$$

$$d@C = \begin{cases} \eta; d@C' & \text{if } C = \eta; C' \\ d@C_1 \mid d@C_2 & \text{if } C = C_1|C_2 \\ k : \mathtt{A} \to \mathsf{q}[\mathtt{B}].\{o_i(x_i); d@C_i)\}_{i\in I} & \text{if } C = k : \mathtt{A} \to \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i\in I} \\ \mathsf{def}\ X\langle\tilde{\mathsf{p}}\rangle = d'@C'' \ \mathsf{in}\ d'@C' & \text{if } \begin{cases} C = \mathsf{def}\ X = C'' \ \mathsf{in}\ C' \wedge \\ \tilde{\mathsf{p}} = \mathsf{fp}(C'') \ \wedge\ d' = d \cup (X, X\langle\tilde{\mathsf{p}}\rangle) \end{cases} \\ \mathbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]}; d@C' & \text{if } C = \mathbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]}; C' \\ \mathsf{if}\ \mathsf{p}.e\ \{d@C_1\}\ \mathsf{else}\ \{d@C_2\} & \text{if } C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \\ X\langle\tilde{\mathsf{p}}\rangle & \text{if } C = X \ \wedge\ (X, X\langle\tilde{\mathsf{p}}\rangle) \in d \\ \mathbf{0} & \text{if } C = \mathbf{0} \end{cases}$$

Figure D.5: Choreography Calculus — Annotation operator

$$
\begin{aligned}
\left\llbracket \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]} \right\rrbracket_{\mathsf{r}} &= \begin{cases} \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \textbf{acc}\ k[g] : l.\mathsf{r}[\mathtt{C}]; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } l.\mathsf{r}[\mathtt{C}] \in \widetilde{l.\mathsf{q}[\mathtt{B}]} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\left\llbracket \textbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]} \right\rrbracket_{\mathsf{r}} &= \begin{cases} \textbf{acc}\ k : l.\mathsf{r}[\mathtt{C}]; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } l.\mathsf{r}[\mathtt{C}] \in \widetilde{l.\mathsf{q}[\mathtt{B}]} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\llbracket \eta; C \rrbracket_{\mathsf{r}} &= \begin{cases} \eta; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \eta \in \{(req), (send)\} \ \wedge\ \{\mathsf{r}\} \in \mathsf{fn}(\eta) \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\llbracket k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathsf{q}[\mathtt{B}].o(x); C \rrbracket_{\mathsf{r}} &= \begin{cases} k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].o(x); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\llbracket k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \rrbracket_{\mathsf{r}} &= \begin{cases} k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].\{o_i(x_i); \llbracket C_i \rrbracket_{\mathsf{r}}\}_{i \in I} & \text{if } \mathsf{r} = \mathsf{q} \\ \bigsqcup_{i \in I} \llbracket C_i \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\llbracket \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \rrbracket_{\mathsf{r}} &= \begin{cases} \mathsf{if}\ \mathsf{p}.e\ \{\llbracket C_1 \rrbracket_{\mathsf{r}}\}\ \mathsf{else}\ \{\llbracket C_2 \rrbracket_{\mathsf{r}}\} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C_1 \rrbracket_{\mathsf{r}} \sqcup \llbracket C_2 \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\llbracket \mathsf{def}\ X\langle \tilde{\mathsf{p}} \rangle = C'\ \mathsf{in}\ C \rrbracket_{\mathsf{r}} &= \begin{cases} \mathsf{def}\ X_{\mathsf{r}} = \llbracket C' \rrbracket_{\mathsf{r}}\ \mathsf{in}\ \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} \in \tilde{\mathsf{p}} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \\
\llbracket X\langle \tilde{\mathsf{p}} \rangle \rrbracket_{\mathsf{r}} &= \begin{cases} X_{\mathsf{r}} & \text{if } \mathsf{r} \in \tilde{\mathsf{p}} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
\llbracket C_1 \mid C_2 \rrbracket_{\mathsf{r}} &= \llbracket C_1 \rrbracket_{\mathsf{r}} \mid \llbracket C_2 \rrbracket_{\mathsf{r}} \\
\llbracket \mathbf{0} \rrbracket_{\mathsf{r}} &= \mathbf{0}
\end{aligned}
$$

Figure D.6: Choreography Calculus, process projection

$$\textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C_1 \ \sqcup\ \textbf{req}\ k : \mathsf{q}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C_2 = \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; (C_1 \sqcup C_2)$$

$$\textbf{acc}\ k : l.\mathsf{p}[\mathtt{A}]; C_1 \ \sqcup\ \textbf{acc}\ k : l.\mathsf{q}[\mathtt{A}]; C_2 = \textbf{acc}\ k : l.\mathsf{p}[\mathtt{A}]; (C_1 \sqcup C_2)$$

$$k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; C_1 \ \sqcup\ k : \mathsf{q}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; C_2 = k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; (C_1 \sqcup C_2)$$

$$k : \mathtt{A} \texttt{->} \mathsf{p}[\mathtt{B}].\, \{o_i(x_i); C_i\}_{i \in I} \ \sqcup\ k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].\, \left\{o_j(x_j); C'_j\right\}_{j \in J} = \\ k : \mathtt{A} \texttt{->} \mathsf{p}[\mathtt{B}]. \left\{ \begin{array}{ll} & \{\ o_i(x_i); C_i\ \}_{i \in I/J} \\ \cup & \{\ o_i(x_i); C'_i\ \}_{i \in J/I} \\ \cup & \{o_i(x_i); C_i \sqcup C'_i\}_{i \in I \cap J} \end{array} \right\}$$

$$\mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C'_1\} \ \sqcup\ \mathsf{if}\ \mathsf{q}.e\ \{C_2\}\ \mathsf{else}\ \{C'_2\} = \mathsf{if}\ \mathsf{p}.e\ \{C_1 \sqcup C_2\}\ \mathsf{else}\ \{C'_1 \sqcup C'_2\}$$

$$\mathsf{def}\ X = C'_1\ \mathsf{in}\ C_1 \ \sqcup\ \mathsf{def}\ X = C'_2\ \mathsf{in}\ C_2 = \mathsf{def}\ X = C'_1 \sqcup C'_2\ \mathsf{in}\ C_1 \sqcup C_2$$

$$X \ \sqcup\ X = X \qquad\qquad \mathbf{0} \ \sqcup\ \mathbf{0} = \mathbf{0}$$

Figure D.7: Merging Function

$$\begin{array}{lcl}
\left\lfloor \textbf{start}\ k : \mathsf{p}[\mathtt{D}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C \right\rfloor_l & = & \left\lfloor \textbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]}; C \right\rfloor_l \\
\left\lfloor \textbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]}; C \right\rfloor_l & = & \begin{cases} \{\mathsf{r}\} \cup \lfloor C \rfloor_l & \text{if } l.\mathsf{r}[\mathtt{A}] \in \widetilde{l.\mathsf{q}[\mathtt{B}]} \\ \lfloor C \rfloor_l & \text{otherwise} \end{cases} \\
\lfloor \eta; C \rfloor_l & = & \lfloor C \rfloor_l \quad \text{if } \eta \neq (start) \\
\lfloor \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \rfloor_l & = & \lfloor C_1 \rfloor_l \cup \lfloor C_2 \rfloor_l \\
\lfloor \mathsf{def}\ X = C'\ \mathsf{in}\ C \rfloor_l & = & \lfloor C' \rfloor_l \cup \lfloor C \rfloor_l \\
\lfloor X \rfloor_l & = & \emptyset \\
\lfloor \mathbf{0} \rfloor_l & = & \emptyset \\
\lfloor C_1 \mid C_2 \rfloor_l & = & \lfloor C_1 \rfloor_l \cup \lfloor C_2 \rfloor_l
\end{array}$$

Figure D.8: Service Grouping

D.4 Dynamic Correlation Calculus

$$\mathsf{def}\ X = B' \ \mathsf{in}\ B[X] \cdot t \cdot M \equiv \mathsf{def}\ X = B' \ \mathsf{in}\ B[B'] \cdot t \cdot M$$
$$P \equiv P \mid \mathbf{0} \cdot t \cdot \emptyset \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$
$$P \mid P' \equiv P' \mid P \qquad S \mid S' \equiv S' \mid S$$
$$(S_1 \mid S_2) \mid S_3 \equiv S_1 \mid (S_2 \mid S_3)$$

Figure D.9: Correlation Calculus, structural congruence

APPENDIX E

Applied Choreographies: Proofs

E.1 Proofs of Subject Reduction and Session Fidelity

We define the semantics of annotated ACs by marking transitions with the name of the session whose term has reduced. We annotate other reductions as τ. We range over annotated labels with

$$\beta ::= k : \mathtt{A} \texttt{ -> } \mathtt{B}.o \mid k : \mathtt{A} \ \rightsquigarrow\ \mathtt{B}.o(x) \mid \tau$$

We report the annotated semantics of AC in Figure E.1.

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o \quad D,\eta \blacktriangleright D'}{D,\ \eta; C \quad \xrightarrow{k\colon \mathtt{A} \texttt{ -> } \mathtt{B}.o} \quad D',\ C} \ \lfloor^{\mathrm{C}}|_{\textsc{Send}}\rceil$$

$$\frac{j \in I \quad D, k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o_j(x_j) \blacktriangleright D'}{D,\ k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i\in I} \quad \xrightarrow{k\colon \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(x_j)} \quad D',\ C_j} \ \lfloor^{\mathrm{C}}|_{\textsc{Recv}}\rceil$$

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x) \quad D, k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o \blacktriangleright D'}{D,\ \eta; C \quad \xrightarrow{k\colon \mathtt{A} \texttt{ -> } \mathtt{B}.o} \quad D',\ k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C} \ \lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil$$

$$\frac{\#\tilde{\mathsf{r}} \quad \#k' \quad \mathsf{p} \in D(l) \quad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l.\mathsf{r}[\mathtt{B}]} \quad D,\delta \blacktriangleright D'}{D,\ \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C \quad \xrightarrow{\tau} \quad D',\ C[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]} \ \lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$$

$$\frac{\begin{array}{c} i \in \{1,\ldots,n\} \qquad \#k' \qquad \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{\widetilde{l_i.\mathtt{B}_i}\} \qquad \#\tilde{\mathsf{r}} \qquad \{\tilde{\mathsf{r}}\} = \bigcup_i \{\tilde{\mathsf{r}}_i\} \\ \mathsf{p} \in D(l) \qquad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l_1.\mathsf{r}_1[\mathtt{B}_1]}, \ldots, \widetilde{l_n.\mathsf{r}_n[\mathtt{B}_n]} \qquad D,\delta \blacktriangleright D' \end{array}}{\begin{array}{c} D, \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C \mid \prod_i \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big) \xrightarrow{\tau} \\ D',\ C[k'/k] \mid \prod_i \big(\ C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\ \big) \mid \prod_i \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big) \end{array}} \ \lfloor^{\mathrm{C}}|_{\textsc{PStart}}\rceil$$

$$\frac{i = 1 \text{ if } \mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{ true}, i = 2 \text{ otherwise}}{D,\ \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \quad \xrightarrow{\tau} \quad D,\ C_i} \ \lfloor^{\mathrm{C}}|_{\textsc{Cond}}\rceil$$

$$\frac{D, C_1 \xrightarrow{\beta} D', C_1'}{D, \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \quad \xrightarrow{\beta} \quad D', \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'} \ \lfloor^{\mathrm{C}}|_{\textsc{Ctx}}\rceil$$

$$\frac{D, C_1 \quad \xrightarrow{\beta} \quad D', C_1'}{D, C_1 \mid C_2 \quad \xrightarrow{\beta} \quad D', C_1' \mid C_2} \ \lfloor^{\mathrm{C}}|_{\textsc{Par}}\rceil$$

$$\frac{\mathcal{R} \in \{\equiv, \simeq_{\mathsf{C}}\} \quad C_1 \mathrel{\mathcal{R}} C_1' \quad D, C_1' \xrightarrow{\beta} D', C_2' \quad C_2' \mathrel{\mathcal{R}} C_2}{D, C_1 \quad \xrightarrow{\beta} \quad D', C_2} \ \lfloor^{\mathrm{C}}|_{\textsc{Eq}}\rceil$$

Figure E.1: Choreography calculus, annotated semantics.

We give a formal definition to $k \notin \Gamma$, which means that Γ has no local typing and buffer types for session k, formally

$$k \notin \Gamma \iff \nexists\, \mathtt{A}, \mathtt{B} \text{ s.t. } k[\mathtt{A}] : T \in \Gamma \ \lor b[k]^{\mathtt{B}}_{\mathtt{A}} : T' \in \Gamma$$

E.1.1 Local and Typing Environment Subtyping

We define a subtyping relation on local types following [116, 96, 14]. We write the subtyping relation as $T \prec T'$, which intuitively indicates that T is more constrained than T' in its behaviour. Note that, like in [96, 14], the input type is covariant and the output type is contravariant for this relation.

Definition 32 (Local Subtyping). *We define the subtyping relation between local types as $T' \prec T$, which is the smallest relation over closed and unfolded local types, satisfying the rules below*

$$\frac{T'' \prec T' \quad (T \approx T'' \lor T \simeq_{\mathsf{T}} T'')}{T \prec T'} \ \lfloor^{SubT}|_{\text{EQ}}\rceil$$

$$\frac{J \subseteq I \quad \forall\, i \in J \mid T_i \prec T'_i \ \land \ U_i \prec U'_i}{!\mathtt{A}.\{o_i(U_i); T_i\}_{i\in I} \ \prec \ !\mathtt{A}.\{o_i(U'_i); T'_i\}_{i\in J}} \ \lfloor^{SubT}|_{\text{SEND}}\rceil$$

$$\frac{I \subseteq J \quad \forall\, i \in I \mid T_i \prec T'_i \ \land \ U_i \prec U'_i}{?\mathtt{A}.\{o_i(U_i); T_i\}_{i\in I} \ \prec \ ?\mathtt{A}.\{o_i(U'_i); T'_i\}_{i\in J}} \ \lfloor^{SubT}|_{\text{RECV}}\rceil$$

$$\frac{}{S \prec S} \ \lfloor^{SubT}|_{\text{VAL}}\rceil \qquad \frac{S \prec S' \quad \forall\, i, U_i \prec U'_i}{S\{\underline{\mathbf{x}}_i : U_i\}_i \prec S'\{\underline{\mathbf{x}}_i : U'_i\}_i} \ \lfloor^{SubT}|_{\text{PATH}}\rceil \qquad \frac{\mathsf{end} \approx T}{\mathsf{end} \prec T} \ \lfloor^{SubT}|_{\text{END}}\rceil$$

Above, $T \approx T'$ is the standard tree isomorphism on recursive types whilst $T \simeq_{\mathsf{T}} T'$ is the swap relation for local types. The swap relation for local types is the smallest relation between local types that satisfies the Rule below. Let $\mathsf{role}(\alpha) = \mathtt{A}$ if either $\alpha = !\mathtt{A}$ or $\alpha = ?\mathtt{A}$.

$$\frac{\mathsf{role}(\alpha) \neq \mathsf{role}(\alpha')}{\alpha.\left\{o_i(U_i); \alpha'.\left\{o'_j(U'_j); T_{ij}\right\}_j\right\}_i \ \simeq_{\mathsf{T}} \ \alpha'.\left\{o'_j(U'_j); \alpha.\left\{o_i(U_i); T_{ij}\right\}_i\right\}_j} \ \lfloor^{\text{T}}|_{\text{SWAP}}\rceil$$

We also define a subtyping relation between Typing Environments. Intuitively $\Gamma \prec \Gamma'$ means that Γ and Γ' are identical Typing Environments up to some local types that are more constrained in Γ — i.e., subtypes of a correspondent local type — than in Γ'.

Definition 33 (Typing Environment Subtyping). *Let Γ and Γ' be two typing environments then*

$$\Gamma' \prec \Gamma \iff \begin{cases} \Gamma = \Gamma_e, \Gamma_t \text{ s.t. } \nexists\, k[\mathtt{A}] : T \in \Gamma_e \\ \Gamma' = \Gamma_e, \Gamma'_t \\ \mathsf{dom}(\Gamma_t) \cap \mathsf{dom}(\Gamma'_t) = \emptyset \\ \forall\, k[\mathtt{A}] : T \in \Gamma, T \prec \Gamma'_t(k[\mathtt{A}]) \end{cases}$$

In Lemma 12 we prove that if $\Gamma' \prec \Gamma$ and Γ types a running choreography D, C also Γ' types that choreography.

Lemma 12 (Subsumption). *Let $\Gamma' \prec \Gamma$ and $\Gamma \vdash D, C$ for some D, C then $\Gamma' \vdash D, C$.*

Proof. Immediate by Definition 32 and Rules $\lfloor^{\mathrm{T}}|_{\mathrm{RECV}}\rceil$ and $\lfloor^{\mathrm{T}}|_{\mathrm{COM}}\rceil$. □

We annotate the reductions of global types with labels

$$\gamma ::= \mathtt{A} \texttt{ -> } \mathtt{B}.o \mid \mathtt{A} \rightsquigarrow \mathtt{B}.o$$

and report below the correspondent annotated semantics.

$$\frac{j \in I}{\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\} \xrightarrow{\mathtt{A} \texttt{ -> } \mathtt{B}.o_j} \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(U_j); G_j} \lfloor^{\mathrm{G}}|_{\mathrm{SEND}}\rceil$$

$$\frac{}{\mathtt{A} \rightsquigarrow \mathtt{B}.o(U); G \xrightarrow{\mathtt{A} \rightsquigarrow \mathtt{B}.o} G} \lfloor^{\mathrm{G}}|_{\mathrm{RECV}}\rceil \qquad \frac{G[\mathsf{rec}\ \mathbf{t}; G/\mathbf{t}] \xrightarrow{\gamma} G'}{\mathsf{rec}\ \mathbf{t}; G \xrightarrow{\gamma} G'} \lfloor^{\mathrm{G}}|_{\mathrm{REC}}\rceil$$

$$\frac{G_1 \simeq_{\mathsf{G}} G_2 \quad G_2 \xrightarrow{\gamma} G'_2 \quad G'_2 \simeq_{\mathsf{G}} G'_1}{G_1 \xrightarrow{\gamma} G'_1} \lfloor^{\mathrm{G}}|_{\mathrm{SWAP}}\rceil$$

In Lemma 13 we account for the fact that any output reduction at the level of global types can constrain the projected local types of the roles not involved in the reduction. Indeed, referring to Rule $\lfloor^{\mathrm{G}}|_{\mathrm{SEND}}\rceil$, the output operation chooses one of the available continuations G_j, $j \in I$ and discards all the others. Therefore the local types of the other roles not involved in the reduction can be constrained by the removal of the other branches.

Lemma 13 (Projection Subtyping). *Let $T = [\![G]\!]_{\mathtt{C}}$, $T' = [\![G']\!]_{\mathtt{C}}$, and $\{\mathtt{A}, \mathtt{B}, \mathtt{C}\} \in \mathsf{roles}(G)$, $\mathtt{C} \notin \{\mathtt{A}, \mathtt{B}\}$, then $G \xrightarrow{\mathtt{A} \texttt{ -> } \mathtt{B}.o} G'$ implies $T' \prec T$.*

Proof. Easy by induction on the derivation of $G \xrightarrow{\gamma} G'$. □

In Lemma 14 we prove that the typing of choreographies is invariant wrt buffer types.

Lemma 14 (Buffer types invariance). *Let* $\Gamma = \Gamma', \Gamma_b$ *where* Γ_b *contains only buffer typings. If* $\Gamma' \vdash C$ *then* $\Gamma \vdash C$.

Proof. Trivial as buffer typings do not affect the typing of choreographies. □

Below we restate the definition of *Deployment Judgements* enriched with pointers of the kind (DX.Y) for a clearer referencing in the proofs.

Definition 18 (Deployment Judgements) $\Gamma \vdash D \iff$

(D18.1) $\forall\, l \in D,\ \forall\, \{\mathsf{p}, \mathsf{q}\} \subseteq D(l),\ \ \mathsf{dom}(D(\mathsf{p}).\mathsf{que}) \cap \mathsf{dom}(D(\mathsf{q}).\mathsf{que}) = \emptyset$

(D18.2) $\forall\, \mathsf{p}.x : U \in \Gamma,\ \vdash x(\ D(\mathsf{p}).\mathsf{st}\ ) : U$

(D18.3) $\forall\, \{\mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}]\} \subseteq \Gamma,\ \ \underline{\mathbf{k}}(\ D(\mathsf{p}).\mathsf{st}\ ) = \underline{\mathbf{k}}(\ D(\mathsf{q}).\mathsf{st}\ )$

(D18.4) $\forall\, \mathsf{p} : k[\mathtt{A}] \in \Gamma, \Gamma \vdash \mathsf{p}@l\ \wedge\ \mathsf{p} \in D(l)\ \wedge\ \underline{\mathbf{k.A.l}}(\ D(\mathsf{p}).\mathsf{st}\ ) = l$

(D18.5) $\forall\, \mathsf{p} : k[\mathtt{A}] \in \Gamma \wedge \forall\, b[k]^{\mathtt{B}}_{\mathtt{A}} : T \in \Gamma,\ \ \mathsf{bte}(\mathtt{B}, D(\mathsf{p}).\mathsf{que}(\underline{\mathbf{k.B.A}}(D(\mathsf{p}).\mathsf{st}))) = T$

We define a reduction relation for typing environments of the form $\Gamma \to \Gamma'$ where $\to$ is the smallest closed under the rules below. Note that the annotation labels are a subset of the labels used to annotate the semantics of AC, ranged over by β.

$$\frac{k \notin \Gamma \quad \Gamma_k \subseteq [\![G]\!]_k \quad \{k[\mathtt{A}] : T, k[\mathtt{B}] : T'\} \in \Gamma_k \quad j \in I \quad G \xrightarrow{\mathtt{A\ ->\ B}.o_j} G'}{\Gamma, \Gamma_k \xrightarrow{k:\ \mathtt{A\ ->\ B}.o_j} \Gamma, \{k[\mathtt{C}] :\ [\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \{b[k]^{\mathtt{D}}_{\mathtt{C}} :\ [\![G']\!]^{\mathtt{D}}_{\mathtt{C}} \mid b[k]^{\mathtt{D}}_{\mathtt{C}} \in \Gamma_k\}}\ \lfloor^{\Gamma}|_{\textsc{Send}}\rceil$$

$$\frac{k \notin \Gamma \quad \Gamma_k \subseteq [\![G]\!]_k \quad \{k[\mathtt{A}] : T, k[\mathtt{B}] : T'\} \in \Gamma_k \quad \Gamma \vdash \mathsf{q} : k[\mathtt{B}] \quad G \xrightarrow{\mathtt{A} \rightsquigarrow \mathtt{B}.o_j} G'}{\Gamma, \Gamma_k \xrightarrow{k:\mathtt{A} \rightsquigarrow \mathtt{B}.o_j(x)} \Gamma, \{k[\mathtt{C}] :\ [\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \{b[k]^{\mathtt{D}}_{\mathtt{C}} :\ [\![G']\!]^{\mathtt{D}}_{\mathtt{C}} \mid b[k]^{\mathtt{D}}_{\mathtt{C}} \in \Gamma_k\}, \mathsf{q}.x : U_j}\ \lfloor^{\Gamma}|_{\textsc{Recv}}\rceil$$

We define the correspondence operator on label $G_{\mathsf{act}}(\beta)$ between β and γ

$$G_{\mathsf{act}}(\beta) = \begin{cases} \mathtt{A\ ->\ B}.o & \text{if } \beta = k : \mathtt{A\ ->\ B}.o \\ \mathtt{A} \rightsquigarrow \mathtt{B}.o & \text{if } \beta = k : \mathtt{A} \rightsquigarrow \mathtt{B}.o(x). \end{cases}$$

In Lemma 15 we prove that if a typing environment Γ includes local types that are projection of a global type G, then if the global type can reduce also the typing environment can reduce. The reduction preserves the correspondence between the reduced global type and the reduced local types in Γ.

Lemma 15 (Type-Environment Fidelity)**.** *Let* $\Gamma = \Gamma_*, [\![G]\!]_k$ *for some* Γ_*, $k \notin \Gamma_*$, *and* $G \xrightarrow{G_{\mathsf{act}}(\beta)} G'$ *then* $\Gamma \xrightarrow{\beta} \Gamma'$ *and for some* Γ'_*, $k \notin \Gamma'_*$, $\Gamma' = \Gamma'_*, [\![G']\!]_k$.

Proof. Direct by cases on the derivation of Γ. □

We also report Lemmas 16 and 17 that prove that typing is invariant wrt, respectively, structural equivalence and swapping.

Lemma 16 (Subject Congruence)**.** $\Gamma \vdash D, C$ *and* $C \equiv C'$ *imply* $\Gamma \vdash D, C'$ *(up to* α*-renaming)*

Proof. By induction on the rules that define $\equiv$. □

Lemma 17 (Subject Swap)**.** $\Gamma \vdash D, C$ *and* $C \simeq_{\mathsf{C}} C'$ *imply* $\Gamma \vdash D, C'$

Proof. By induction on the derivation of $C \simeq_{\mathsf{C}} C'$. □

Finally, we prove Theorem 3 by proving the stronger result below.

Theorem 2 (Typing Soundness)**.** *Let* D, C *be an annotated AC and* (T2.1) $\Gamma \vdash D, C$ *for some* Γ*:*

if (T2.2) $\beta \neq \tau$ *and* $D, C \xrightarrow{\beta} D', C'$ *then* (T2.3) $\Gamma \xrightarrow{\beta} \Gamma'$ *and* (T2.4) $\Gamma' \vdash D', C'$*;*

if (T2.5) $D, C \xrightarrow{\tau} D', C'$ *then, for some* Γ'*, (T2.6)* $\Gamma' \vdash D', C'$.

Proof. Proof by induction on the derivation of $D, C \xrightarrow{\beta} D', C'$.

Case $\lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$
The case is:

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o_j \quad D, \eta \blacktriangleright D'}{D,\ \eta; C \quad \xrightarrow{k\colon \mathtt{A} \texttt{ -> } \mathtt{B}.o_j} \quad D',\ C} \ \lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$$

Where (T2.2) has $C' = C$ and $D' = D\big[\mathsf{q} \mapsto (\ D(\mathsf{q}).\mathsf{st}, D(\mathsf{q}).\mathsf{que}[t_c \mapsto \tilde{m} :: (o_j, t_m)]\ )\big]$ by Rule $\lfloor^{\mathsf{D}}|_{\text{SEND}}\rceil$.

To prove (T2.3) we must prove Rule $\lfloor^{\Gamma}|_{\text{SEND}}\rceil$ to be applicable.

From (T2.1) we know that there exists a global type G for session k such that $\mathsf{pco}(\Gamma)$ holds. We can partition $\Gamma = \Gamma_*, \Gamma_k$ such that $\Gamma_* = \Gamma / [\![G]\!]_k$ and $\Gamma_k = \Gamma / \Gamma_*$.

From (T2.1) we can write the derivation (with $\Gamma = \Gamma_1, k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{A}}\}_{i \in I}$, $j \in I$)

$$\dfrac{\mathsf{pco}(\Gamma) \quad \Gamma \vdash D \quad \dfrac{j \in I \quad \Gamma_1 \vdash \mathsf{p}\colon k[\mathtt{A}] \quad \Gamma_1 \vdash \mathsf{p}.e\colon U_j \quad \Gamma_1, k[\mathtt{A}]\colon [\![G_j]\!]_{\mathtt{A}} \vdash C}{\Gamma_1, k[\mathtt{A}]\colon !\mathtt{B}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{A}}\}_{i\in I} \vdash k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o_j; C}\lfloor^{\mathtt{T}}|_{\text{SEND}}\rceil}{\Gamma \vdash D, k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o_j; C}\lfloor^{\mathtt{T}}|_{\text{DC}}\rceil$$

Since $\Gamma \vdash k[\mathtt{A}]\colon !\mathtt{B}.\{o_i(U_i); T_i\}_{i\in I}$, G must be swap-equivalent to type $G_* = \mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\}_{i\in I}$, where $\forall\ i \in I$, $[\![G_i]\!]_{\mathtt{B}} = T_i$. G_* reduces with Rule $\lfloor^{\mathtt{G}}|_{\text{SEND}}\rceil$

$$\dfrac{j \in I}{\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\} \xrightarrow{\mathtt{A} \texttt{ -> } \mathtt{B}.o_j} \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(U_j); G_j}\lfloor^{\mathtt{G}}|_{\text{SEND}}\rceil$$

and we can apply rule $\lfloor^{\Gamma}|_{\text{SEND}}\rceil$ where $G'_* = \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(U_j); G_j$ and $G' \simeq_{\mathtt{G}} G'_*$

$$\dfrac{\dfrac{G \simeq_{\mathtt{G}} G_* \quad G_* \xrightarrow{\mathtt{A} \texttt{ -> } \mathtt{B}.o_j} G'_* \quad G'_* \simeq_{\mathtt{G}} G'}{G \xrightarrow{\mathtt{A} \texttt{ -> } \mathtt{B}.o_j} G'}\lfloor^{\mathtt{G}}|_{\text{SWAP}}\rceil \quad k \notin \Gamma_* \qquad \Gamma_k \subseteq [\![G]\!]_k \qquad \{k[\mathtt{A}]\colon T, k[\mathtt{B}]\colon T'\} \in \Gamma_k \qquad j \in I}{\Gamma_*, \Gamma_k \xrightarrow{k\colon\, \mathtt{A} \texttt{ -> } \mathtt{B}.o_j} \Gamma_*, \{k[\mathtt{C}]\colon [\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \{b[k]^{\mathtt{C}}_{\mathtt{D}}\colon [\![G']\!]^{\mathtt{D}}_{\mathtt{C}} \mid b[k]^{\mathtt{D}}_{\mathtt{C}} \in \Gamma_k\}}\lfloor^{\Gamma}|_{\text{SEND}}\rceil$$

Hence (T2.3) holds and $\Gamma' = \Gamma_*, \{k[\mathtt{C}]\colon [\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \{b[k]^{\mathtt{D}}_{\mathtt{C}}\colon [\![G']\!]^{\mathtt{D}}_{\mathtt{C}} \mid b[k]^{\mathtt{D}}_{\mathtt{C}} \in \Gamma_k\}$.

We now prove (T2.4) by proving that Rule $\lfloor^{\mathtt{T}}|_{\text{DC}}\rceil$ applies to $\Gamma' \vdash D', C'$.

$$\dfrac{\mathsf{pco}(\Gamma') \quad \Gamma' \vdash C' \quad \Gamma' \vdash D'}{\Gamma' \vdash D', C'}\lfloor^{\mathtt{T}}|_{\text{DC}}\rceil$$

Hence we need to prove ① $\mathsf{pco}(\Gamma')$, ② $\Gamma' \vdash C'$. and ③ $\Gamma' \vdash D'$

Proof of ①. For all sessions $k' \in \Gamma_*$, $\mathsf{pco}(\Gamma')$ holds as $\mathsf{pco}(\Gamma)$ holds by (T2.1). For session k, $\mathsf{pco}(\Gamma')$ holds by construction.

□

Proof of ②. From the derivation on $\Gamma \vdash D, k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o_j; C$ we know that $\Gamma_1, k[\mathtt{A}]\colon [\![G_j]\!]_{\mathtt{A}} \vdash C$. Let $\Gamma'' = \Gamma_1, k[\mathtt{A}]\colon [\![G_j]\!]_{\mathtt{A}}$ and $\Gamma'_k = \Gamma_1/\Gamma_* = \Gamma_k/\{k[\mathtt{A}]\colon [\![G]\!]_{\mathtt{A}}\}$. We can write $\Gamma'' = \Gamma_*, \Gamma'_k, k[\mathtt{A}]\colon [\![G_j]\!]_{\mathtt{A}}$. Notably the application of Rule $\lfloor^{\mathtt{T}}|_{\text{SEND}}\rceil$

does not modify the buffer types in Γ and therefore $\Gamma''(b[k]^{\mathtt{A}}_{\mathtt{B}}) \neq \Gamma'(b[k]^{\mathtt{A}}_{\mathtt{B}})$, however from Lemma 14 we know that we can omit to consider buffer types as they are irrelevant for the typing of choreographies. For all sessions $k' \neq k$ in Γ'' their local typings are the same in Γ'. For session k, the typing $\Gamma''(k[\mathtt{A}]) = \Gamma'(k[\mathtt{A}]) = [\![G_j]\!]_{\mathtt{A}}$. From Lemma 13, for all other $k[\mathtt{C}] \in \Gamma'', \mathtt{C} \neq \mathtt{A}$ it holds that $\Gamma''(k[\mathtt{C}]) = [\![G]\!]_{\mathtt{C}}$, $\Gamma'(k[\mathtt{C}]) = [\![G']\!]_{\mathtt{C}}$, and $[\![G']\!]_{\mathtt{C}} \prec [\![G]\!]_{\mathtt{C}}$. Therefore $\Gamma' \prec \Gamma''$ and ② holds by Lemma 12.

$\square$

Proof of ③. To prove $\Gamma' \vdash D'$ we need to prove that the conditions of Definition 18 hold. (D18.1–D18.4) hold by the application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$, by construction of Γ', and by (T2.1). (D18.5) holds for all sessions $k' \neq k$ by application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$ and the construction of Γ'. The same holds true for session k and any process $\mathsf{q} : k[\mathtt{C}] \in \Gamma' \mid \mathtt{C} \neq \mathtt{B}$.

Finally, let $\mathsf{q} : k[\mathtt{B}] \in \Gamma$, from $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ we know that

$$D'(\mathsf{q}).\mathsf{que}(\underline{\mathbf{k.A.B}}(D'(\mathsf{q}).\mathsf{st})) = \tilde{m} :: (o_j, t_m)$$

G does not contain any partial reception from $\mathtt{A}$ to $\mathtt{B}$, hence $\tilde{m}$ must be empty

$$D'(\mathsf{q}).\mathsf{que}(\underline{\mathbf{k.A.B}}(D'(\mathsf{q}).\mathsf{st})) = (o_j, t_m)$$

From the definition of type projection we have that $\Gamma'(\ b[k]^{\mathtt{A}}_{\mathtt{B}}\ ) = ?\mathtt{A}.o_j(U_j)$.

From $\lfloor^{\mathrm{T}}|_{\mathrm{SEND}}\rceil$ we know that $\mathsf{p}.e \vdash U_j$ and from $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ that $t_m = eval(e, D(\mathsf{p}).\mathsf{st})$, thus t_m has type U_j and $\mathsf{bte}(\ \mathtt{A}, (o_j, U_j)\ ) = ?\mathtt{A}.o_j(U_j)$. $\square$

Case $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$
The case is:

$$\frac{j \in I \quad D, k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x_j) \blacktriangleright D'}{D,\ k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \xrightarrow{k:\mathtt{A} \rightsquigarrow \mathtt{B}.o_j(x_j)} D',\ C_j} \lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$$

(T2.2) has $C' = C_j$ and $D' = D\big[\mathsf{q} \mapsto (\ D(\mathsf{q}).\mathsf{st} \lhd (\ x, t_m\ ), D(\mathsf{q}).\mathsf{que}[t_c \mapsto \tilde{m}]\ )\big]$ by application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$.

To prove (T2.3) we must prove that Rule $\lfloor^{\Gamma}|_{\mathrm{SEND}}\rceil$ is applicable.

Since D, C reduces with $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$, the conditions of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ must hold, i.e., $t_c = \underline{\mathbf{k.A.B}}(D(\mathsf{q}).\mathsf{st}) \ \wedge\ D(\mathsf{q}).\mathsf{que}(t_c) = (o_j, t_m) ::\ \tilde{m}$. Since (T2.1) holds $\mathsf{pco}(\Gamma)$ and $\Gamma \vdash D$ hold and therefore we know that, by (D18.5), $\Gamma(b[k]^{\mathtt{A}}_{\mathtt{B}}) = \mathsf{bte}(\mathtt{A},\ (o_j, t_m) ::\ \tilde{m})$.

Let $\vdash t_m : U_j$, then $\mathsf{bte}(\mathtt{A},\ (o_j, t_m) ::\ \tilde{m}) = ?\mathtt{A}.o_j(U_j); T$ where $T = \mathsf{bte}(\mathtt{A},\ \tilde{m})$ by definition and $\Gamma(b[k]^{\mathtt{A}}_{\mathtt{B}}) = ?\mathtt{A}.o_j(U_j); T$. Since $\mathsf{pco}(\Gamma)$ holds, there exists a global type G for session k such that G is swap-equivalent to a type $G_* = \mathtt{A} \leadsto \mathtt{B}.o_j(U_j); G_j$ where $[\![G_j]\!]^{\mathtt{A}}_{\mathtt{B}} = T$.

G_* reduces with Rule $\lfloor^{\mathrm{G}}|_{\textsc{Recv}}\rceil$

$$\frac{}{\mathtt{A} \leadsto \mathtt{B}.o(U); G \xrightarrow{\mathtt{A} \leadsto \mathtt{B}.o(U)} G}\ \lfloor^{\mathrm{G}}|_{\textsc{Recv}}\rceil$$

and we can apply Rule $\lfloor^{\Gamma}|_{\textsc{Recv}}\rceil$ with $G'_* = G_j$ and $G' \simeq_{\mathtt{G}} G'_*$

We partition $\Gamma = \Gamma_*, \Gamma_k$ such that $\Gamma_* = \Gamma / [\![G]\!]_k$ and $\Gamma_k = \Gamma/\Gamma_*$.

$$\cfrac{\cfrac{G \simeq_{\mathtt{G}} G_* \quad G_* \xrightarrow{\mathtt{A} \leadsto \mathtt{B}.o_j} G'_* \quad G'_* \simeq_{\mathtt{G}} G'}{G \xrightarrow{\mathtt{A} \leadsto \mathtt{B}.o_j} G'}\ \lfloor^{\mathrm{G}}|_{\textsc{Swap}}\rceil \\ k \notin \Gamma_* \qquad \Gamma_k \subseteq [\![G]\!]_k \qquad \{k[\mathtt{A}]: T, k[\mathtt{B}]: T'\} \in \Gamma_k \qquad \Gamma_* \vdash \mathsf{q}: k[\mathtt{B}]}{\Gamma_*, \Gamma_k \xrightarrow{k:\mathtt{A} \leadsto \mathtt{B}.o_j(x)} \Gamma_*, \{k[\mathtt{C}]: [\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \{b[k]^{\mathtt{D}}_{\mathtt{C}}: [\![G']\!]^{\mathtt{D}}_{\mathtt{C}} \mid b[k]^{\mathtt{D}}_{\mathtt{C}} \in \Gamma_k\}, \mathsf{q}.x: U_j}\ \lfloor^{\Gamma}|_{\textsc{Recv}}\rceil$$

Hence (T2.3) holds and $\Gamma' = \Gamma_*, \{[\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \mathsf{q}.x: U_j$.

(T2.4) holds if we can apply Rule $\lfloor^{\mathrm{T}}|_{\textsc{DC}}\rceil$ on $\Gamma' \vdash D', C'$

$$\frac{\mathsf{pco}(\Gamma') \quad \Gamma' \vdash C' \quad \Gamma' \vdash D'}{\Gamma' \vdash D', C'}\ \lfloor^{\mathrm{T}}|_{\textsc{DC}}\rceil$$

and we need to prove ① $\mathsf{pco}(\Gamma')$, ② $\Gamma' \vdash C'$. and ③ $\Gamma' \vdash D'$

The proof of ① for this case is similar to that of ① for case $\lfloor^{\mathrm{C}}|_{\textsc{Send}}\rceil$.

Proof of ②. From (T2.1), partitioning $\Gamma = \Gamma_1, k[\mathtt{B}]: ?\mathtt{A}.o_j(U_j); [\![G_j]\!]_{\mathtt{B}}$ and since $j \in I$ from Rule $\lfloor^{\mathrm{C}}|_{\textsc{Recv}}\rceil$, we can write the derivation

$$\cfrac{\mathsf{pco}(\Gamma) \quad \Gamma \vdash D \quad \cfrac{j \in I \quad \Gamma_1 \vdash \mathsf{q}: k[\mathtt{B}] \quad \Gamma_1, \mathsf{q}.x_j: U_j, k[\mathtt{B}]: [\![G_j]\!]_{\mathtt{B}} \vdash C_j}{\Gamma_1, k[\mathtt{B}]: ?\mathtt{A}.o_j(U_j); [\![G_j]\!]_{\mathtt{B}} \vdash k : \mathtt{A}\ \texttt{->}\ \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i\in I}}\ \lfloor^{\mathrm{T}}|_{\textsc{Recv}}\rceil}{\Gamma \vdash D, k : \mathtt{A}\ \texttt{->}\ \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i\in I}}\ \lfloor^{\mathrm{T}}|_{\textsc{DC}}\rceil$$

hence we know that $\Gamma_1, \mathsf{q}.x_j: U_j, k[\mathtt{B}]: [\![G_j]\!]_{\mathtt{B}} \vdash C_j$.

Let $\Gamma'' = \Gamma_1, \mathsf{q}.x_j: U_j, k[\mathtt{B}]: [\![G_j]\!]_{\mathtt{B}} \vdash C_j$ and $\Gamma'_k = \Gamma_1/\Gamma_* = \Gamma_k/\{k[\mathtt{B}]: [\![G]\!]_{\mathtt{B}}\}$. We can write $\Gamma'' = \Gamma_*, \Gamma'_k, k[\mathtt{B}]: [\![G_j]\!]_{\mathtt{B}}$. Similarly to the proof of ② for case

$\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ $\Gamma''(b[k]^{\mathtt{A}}_{\mathtt{B}}) \neq \Gamma'(b[k]^{\mathtt{A}}_{\mathtt{B}})$, but we omit to consider buffer types as they are irrelevant for the typing of choreographies by Lemma 14. For all sessions in Γ'', their local typings are the same as in Γ'. We consider in particular k on which we applied the reduction for this case for which it holds

$$\forall\, k[\mathtt{C}] \in \Gamma'',\ \Gamma''(k[\mathtt{C}]) = \Gamma'(k[\mathtt{C}]) = [\![G']\!]_{\mathtt{C}}$$

□

Proof of ③. To prove $\Gamma' \vdash D'$ we prove the conditions in Definition 18. (D18.1) holds from the application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$, (T2.1), and the construction of Γ'. (D18.2) holds for all $\mathsf{p}.x$ from the application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$, (T2.1), and the construction of Γ', except for $\mathsf{q}.x_j$ which is not defined in Γ. However the condition holds by construction of $\Gamma' = \Gamma_1, \mathsf{q}.x_j : U_j, k[\mathtt{B}] : [\![G_j]\!]_{\mathtt{B}}$. (D18.3–D18.4) hold by application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ and (T2.1). (D18.5) holds for all sessions $k' \neq k$ by the application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ and the construction of Γ'. The same holds true for session k and any process $\mathsf{p} : k[\mathtt{C}] \in \Gamma \mid \mathtt{C} \neq \mathtt{B}$.

For $\mathsf{q} : k[\mathtt{B}]$ and role $\mathtt{A}$ we know from the application of $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ that

$$D'(\mathsf{q}).\mathsf{que}(\underline{\mathbf{k.A.B}}(D'(\mathsf{q}).\mathsf{st})) = \tilde{m}$$

Since we took G such that $[\![G]\!]^{\mathtt{A}}_{\mathtt{B}} = ?\mathtt{A}.o_j(U_j); T$, where $T = \mathsf{bte}(\mathtt{A}, \tilde{m})$ then $[\![G']\!]^{\mathtt{A}}_{\mathtt{B}} = T$.

□

Case $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$
The case is:

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x) \quad D, k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o_j \blacktriangleright D'}{D,\ \eta; C \xrightarrow{k:\ \mathtt{A} \text{ -> } \mathtt{B}.o_j} D',\ k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x); C}\ \lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$$

Where (T2.2) has $C' = k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].o(x); C$ and
$D' = D\big[\mathsf{q} \mapsto (\ D(\mathsf{q}).\mathsf{st}, D(\mathsf{q}).\mathsf{que}[t_c \mapsto \tilde{m} :: (o_j, t_m)]\ )\big]$ by application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$.

To prove (T2.3) we must prove Rule $\lfloor^{\Gamma}|_{\mathrm{SEND}}\rceil$ to be applicable.

Similarly to the proof of case $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ we know there exists a global type G for session k such that $\mathsf{pco}(\Gamma)$ holds. Likewise, we partition $\Gamma = \Gamma_*, \Gamma_k$ wrt $[\![G]\!]_k$.

Partitioning $\Gamma = \Gamma_1, k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{A}}\}_{i \in I}, k[\mathtt{B}] : ?\mathtt{A}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{B}}\}_{i \in I}$ we can write the derivation

$$\dfrac{\mathsf{pco}(\Gamma) \quad \Gamma \vdash D \qquad \dfrac{\begin{array}{c} j \in I \qquad \Gamma_1 \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \qquad \Gamma_1 \vdash \mathsf{p}.e : U_j \\ \Gamma_1, \mathsf{q}.x : U_j, k[\mathtt{A}] : [\![G_j]\!]_{\mathtt{A}}, k[\mathtt{B}] : [\![G_j]\!]_{\mathtt{B}} \vdash C \end{array}}{\begin{array}{r} \Gamma_1, k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{A}}\}_i, k[\mathtt{B}] : ?\mathtt{A}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{B}}\}_i \\ \vdash k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x); C \end{array}} \lfloor{}^{\mathrm{T}}|_{\mathrm{COM}}\rceil}{\Gamma \vdash D, k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x); C} \lfloor{}^{\mathrm{T}}|_{\mathrm{DC}}\rceil$$

Since $\Gamma \vdash k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{A}}\}_i$ and $\Gamma \vdash k[\mathtt{B}] : ?\mathtt{A}.\{o_i(U_i); [\![G_i]\!]_{\mathtt{B}}\}_i$, G must be a swap-equivalent to type $G_* = \mathtt{A} \text{ -> } \mathtt{B}.\{o_i(U_i); G_i\}_i$, which reduces along Rule $\lfloor{}^{\mathrm{G}}|_{\mathrm{SEND}}\rceil$

$$\dfrac{j \in I}{\mathtt{A} \text{ -> } \mathtt{B}.\{o_i(U_i); G_i\} \quad \to \quad \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(U_j); G_j} \lfloor{}^{\mathrm{G}}|_{\mathrm{SEND}}\rceil$$

where $G'_* = \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(U_j); G_j$ and $G'_* \simeq_{\mathsf{G}} G'$. Therefore we can apply Rule $\lfloor{}^{\Gamma}|_{\mathrm{SEND}}\rceil$

$$\dfrac{\dfrac{G \simeq_{\mathsf{G}} G_* \quad G_* \xrightarrow{\mathtt{A} \text{ -> } \mathtt{B}.o_j} G'_* \quad G'_* \simeq_{\mathsf{G}} G'}{G \xrightarrow{\mathtt{A} \text{ -> } \mathtt{B}.o_j} G'} \lfloor{}^{\mathrm{G}}|_{\mathrm{SWAP}}\rceil \\ k \notin \Gamma_* \qquad \Gamma_k \subseteq [\![G]\!]_k \qquad \{k[\mathtt{A}] : T, k[\mathtt{B}] : T'\} \in \Gamma_k \qquad j \in I}{\Gamma_*, \Gamma_k \xrightarrow{k \colon \mathtt{A} \text{ -> } \mathtt{B}.o_j} \Gamma_*, \{k[\mathtt{C}] : [\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}, \{b[k]^{\mathtt{D}}_{\mathtt{C}} : [\![G']\!]^{\mathtt{D}}_{\mathtt{C}} \mid b[k]^{\mathtt{D}}_{\mathtt{C}} \in \Gamma_k\}} \lfloor{}^{\Gamma}|_{\mathrm{SEND}}\rceil$$

We prove that for $\Gamma' = \Gamma_*, \{[\![G']\!]_{\mathtt{C}} \mid k[\mathtt{C}] \in \Gamma_k\}$ (T2.4) holds, i.e., that Rule $\lfloor{}^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ is applicable on $\Gamma' \vdash D', C'$ and therefore that ① $\mathsf{pco}(\Gamma')$, ② $\Gamma' \vdash C'$ and ③ $\Gamma' \vdash D'$ hold.

The proof of ① and ③ for this case is similar to those of, respectively, ① and ③ for case $\lfloor{}^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$.

Proof of ②. Let $\Gamma'' = \Gamma_1, \mathsf{q}.x : U_j, k[\mathtt{A}] : [\![G_j]\!]_{\mathtt{A}}, k[\mathtt{B}] : [\![G_j]\!]_{\mathtt{B}}$, we know from the derivation on $\Gamma \vdash D, k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x); C$ that $\Gamma'' \vdash C$.

We need to prove that $\Gamma' \vdash C'$ and therefore that Rule $\lfloor{}^{\mathrm{T}}|_{\mathrm{RECV}}\rceil$ applies. $\Gamma'(k[\mathtt{B}]) = [\![G']\!]_{\mathtt{B}} = ?\mathtt{A}.o_j(U_j); [\![G_j]\!]_{\mathtt{B}}$ by construction and thus we can write $\Gamma' = \Gamma'_1, k[\mathtt{B}] : ?\mathtt{A}.o_j(U_j); [\![G_j]\!]_{\mathtt{B}}$ and apply Rule $\lfloor{}^{\mathrm{T}}|_{\mathrm{RECV}}\rceil$

$$\dfrac{\Gamma'_1 \vdash \mathsf{q} : k[\mathtt{B}] \quad \Gamma'_1, \mathsf{q}.x_j : U_j, k[\mathtt{B}] : [\![G_j]\!]_{\mathtt{B}} \vdash C}{\Gamma'_1, k[\mathtt{B}] : ?\mathtt{A}.o_j(U_j); [\![G_j]\!]_{\mathtt{B}} \vdash k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathsf{q}[\mathtt{B}].o_j(x); C} \lfloor{}^{\mathrm{T}}|_{\mathrm{RECV}}\rceil$$

Again from the derivation of $\Gamma \vdash D, k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathsf{q}[\mathtt{B}].o_j(x); C$ we know that

$$\Gamma_1, \mathsf{q}.x \colon U_j, k[\mathtt{A}] \colon [\![G_j]\!]_{\mathtt{A}}, k[\mathtt{B}] \colon [\![G_j]\!]_{\mathtt{B}} \vdash C$$

Similarly to the proof of ② of case $\lfloor^{\mathrm{C}}|_{\textsc{Send}}\rceil$, $\Gamma' \vdash C'$ since 1) we can omit to consider buffer types as of Lemma 14, 2) for all session $k' \neq k$ its local typings remain unchanged from Γ to Γ', and 3) for session k, $\Gamma''(k[\mathtt{A}]) = \Gamma'(k[\mathtt{A}]) = [\![G_j]\!]_{\mathtt{A}}$ and for all other local typings for $\mathtt{C} \in \tilde{\mathtt{G}}/\{\mathtt{A}\}$, $\Gamma''(k[\mathtt{C}]) = [\![G]\!]_{\mathtt{C}}$, $\Gamma'(k[\mathtt{C}]) = [\![G']\!]_{\mathtt{C}}$, and $[\![G']\!]_{\mathtt{C}} \prec [\![G]\!]_{\mathtt{C}}$. Hence, $\Gamma' \prec \Gamma''$ and ② holds by Lemma 12.

□

Case $\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$
The case is:

$$\frac{\#\tilde{\mathsf{r}} \quad \#k' \quad \mathsf{p} \in D(l) \quad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l.\mathsf{r}[\mathtt{B}]} \quad D, \delta \blacktriangleright D'}{D,\ \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C \quad \xrightarrow{\tau} \quad D',\ C[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]} \ \lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$$

Where (T2.5) has $C' = C[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]$. D' is defined non-deterministically but abides the requirements defined in Rule $\lfloor^{\mathrm{D}}|_{\textsc{Start}}\rceil$. Let $\widetilde{\mathsf{s}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{r}[\mathtt{B}]}$. Since (T2.1) holds, we can apply Rule $\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$. We partition $\Gamma = \Gamma_1, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle$

$$\frac{\Gamma_1, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \mathsf{init}\big(\, \widetilde{\mathsf{s}'[\mathtt{C}]}, k, G \,\big) \vdash C \quad \widetilde{\mathsf{s}'[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \quad \tilde{\mathsf{q}} \notin \Gamma_1}{\Gamma_1, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C} \ \lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$$

We take $\Gamma' = \Gamma, \mathsf{init}\big(\, \widetilde{\mathsf{s}[\mathtt{C}]}, k', G \,\big)$ and we prove the case by proving that we can apply Rule $\lfloor^{\mathrm{T}}|_{\textsc{DC}}\rceil$ on $\Gamma' \vdash D', C'$, i.e, that the following hold: ① $\mathsf{pco}(\Gamma')$, ② $\Gamma' \vdash C'$ and ③ $\Gamma' \vdash D'$.

Proof of ①. ① holds for all session $k'' \in \Gamma', k'' \neq k'$ by (T2.1). For session k' ① holds by construction. □

Proof of ②. By (T2.1) we could apply $\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$ where $\Gamma, \mathsf{init}\big(\, \widetilde{\mathsf{s}'[\mathtt{C}]}, k, G \,\big) \vdash C$. We apply α-renaming on k and $\tilde{\mathsf{q}}$ along what done in Rule $\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$ and we obtain that $\Gamma, \mathsf{init}\big(\, \widetilde{\mathsf{s}'[\mathtt{C}]}, k, G \,\big)[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}] \vdash C[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]$.

Since $\Gamma, \mathsf{init}\big(\, \widetilde{\mathsf{s}'[\mathtt{C}]}, k, G \,\big)[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}] = \Gamma, \mathsf{init}\big(\, \widetilde{\mathsf{s}[\mathtt{C}]}, k', G \,\big)$ then ② holds. □

Proof of ③. To prove ③ we prove the conditions in Definition 18. (D18.1–D18.5) hold by the application of Rule $\lfloor^{\text{D}}|_{\textsc{Start}}\rceil$ and the construction of Γ'. $\square$

Case $\lfloor^{\text{C}}|_{\textsc{PStart}}\rceil$
The case is:

$$\dfrac{\begin{array}{c} i \in \{1, \dots, n\} \qquad \#k' \qquad \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{\widetilde{l_i.\mathtt{B}_i}\} \qquad \#\tilde{\mathsf{r}} \qquad \{\tilde{\mathsf{r}}\} = \bigcup_i \{\tilde{\mathsf{r}}_i\} \\ \mathsf{p} \in D(l) \qquad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l_1.\mathsf{r}_1[\mathtt{B}_1]}, \dots, \widetilde{l_n.\mathsf{r}_n[\mathtt{B}_n]} \qquad D, \delta \blacktriangleright D' \end{array}}{\begin{array}{l} D, \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C \mid \prod_i \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big) \xrightarrow{\tau} \\ \qquad D',\ C[k'/k] \mid \prod_i \big(\, C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\,\big) \mid \prod_i \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big) \end{array}} \ \lfloor^{\text{C}}|_{\textsc{PStart}}\rceil$$

Where (T2.5) has $C' = C[k'/k] \mid \prod_i \big(\, C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\,\big) \mid \prod_i \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)$.
D' is defined non-deterministically but abides the requirements defined in Rule $\lfloor^{\text{D}}|_{\textsc{Start}}\rceil$.
We partition Γ such that:

- $\Gamma = \Gamma_r \circ \Gamma_a$
- $\Gamma_r \vdash \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle$
- $\Gamma_a \vdash \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle$
- $\Gamma_a = \Gamma_1, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\widetilde{\mathtt{B}}_1\rangle \circ \cdots \circ \Gamma_n, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\widetilde{\mathtt{B}}_n\rangle$
- $\Gamma_a^i = \Gamma_i, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\widetilde{\mathtt{B}}_i\rangle \circ \cdots \circ \Gamma_n, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\widetilde{\mathtt{B}}_n\rangle$

and we can write the derivation

$$\dfrac{\mathsf{pco}(\Gamma) \quad \Gamma \vdash D \qquad \dfrac{\dfrac{\Gamma_r, \mathsf{p} \colon k[\mathtt{A}], k[\mathtt{A}] \colon\ [\![G]\!]_{\mathtt{A}} \vdash C \quad \Gamma_r \vdash \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle}{\Gamma_r \vdash \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C} \lfloor^{\text{T}}|_{\textsc{Req}}\rceil \qquad \Delta_1}{\Gamma \vdash \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C \mid \prod_{i \in I} \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)} \lfloor^{\text{T}}|_{\textsc{Par}}\rceil}{\Gamma \vdash D, \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C \mid \prod_{i \in I} \big(\textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)} \lfloor^{\text{T}}|_{\textsc{DC}}\rceil$$

$$\Delta_i = \left\{ \dfrac{\dfrac{\tilde{l}_i \subseteq \tilde{l} \quad \Gamma_i, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle, \mathsf{init}\big(\, \widetilde{\mathsf{q}_i[\mathtt{B}_i]}, k, G\,\big) \vdash C_i}{\Gamma_i, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\widetilde{\mathtt{B}_i}\rangle \vdash \textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i} \lfloor^{\text{T}}|_{\textsc{Acc}}\rceil \qquad \Delta_{i+1}}{\Gamma_a^i \vdash \textbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i \mid \prod_{j \in I/\{1,\cdots,i\}} \big(\textbf{acc}\ k : \widetilde{l_j.\mathsf{q}_j[\mathtt{B}_j]}; C_j\big)} \lfloor^{\text{T}}|_{\textsc{Par}}\rceil \right.$$

Let $\widetilde{\mathsf{s}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{r}_1[\mathtt{B}_1]}, \cdots, \widetilde{\mathsf{r}_n[\mathtt{B}_n]}$.

To prove (T2.6) we take

$$\Gamma' = \Gamma, \mathsf{init}\big(\ \widetilde{\mathsf{s}[\mathtt{C}]}, k', G\ \big) = \Gamma_r \circ \Gamma_a \circ \mathsf{init}\big(\ \widetilde{\mathsf{s}[\mathtt{C}]}, k', G\ \big)$$

By the definition of the merging operator $\circ$ we can partition $\mathsf{init}\big(\ \widetilde{\mathsf{s}[\mathtt{C}]}, k', G\ \big)$ such that

$$\Gamma' = \Gamma'_r \circ \Gamma'_a \circ \Gamma_a$$

Where

- $\Gamma'_r = \Gamma_r, \mathsf{init}\big(\ \mathsf{p}[\mathtt{A}], k', G\ \big)$
- $\Gamma'_a = \Gamma'_1 \circ \cdots \circ \Gamma'_n$
- $\Gamma'_i = \Gamma_i, \tilde{l}\colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle, \mathsf{init}\big(\ \widetilde{\mathsf{r}_i[\mathtt{B}_i]}, k', G\ \big)$

To prove (T2.6) we must prove we can apply Rule $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ on $\Gamma' \vdash D', C'$.

$$\dfrac{\begin{array}{l}\textcircled{1}\ \mathsf{pco}(\Gamma') \\ \textcircled{3}\ \Gamma' \vdash D'\end{array} \quad \textcircled{2} \left\{ \dfrac{\textcircled{2a}\ \Gamma'_r \vdash C[k'/k] \qquad \dfrac{\begin{array}{c}\textcircled{2b}\ \Gamma'_a \vdash \prod_i \big(\ C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\ \big) \\ \textcircled{2c}\ \Gamma_a \vdash \prod_i \big(\mathbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)\end{array}}{\begin{array}{l}\Gamma'_a \circ \Gamma_a \\ \qquad \vdash \prod_i \big(\ C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\ \big) \mid \prod_i \big(\mathbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)\end{array}} \lfloor^{\mathrm{T}}|_{\mathrm{PAR}}\rceil}{\Gamma' \vdash C[k'/k] \mid \prod_i \big(\ C_i[k'/k][\tilde{\mathsf{r}}_i/\tilde{\mathsf{q}}_i]\ \big) \mid \prod_i \big(\mathbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)} \lfloor^{\mathrm{T}}|_{\mathrm{PAR}}\rceil \right.}{\Gamma' \vdash D', C'} \lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$$

$\textcircled{1}$ holds by by construction.

Proof of $\textcircled{2}$. $\textcircled{2}$ holds as

- $\textcircled{2a}$ holds by α-renaming $(\Gamma_r, \mathsf{p}\colon k[\mathtt{A}], k[\mathtt{A}]\colon [\![G]\!]_{\mathtt{A}})[k'/k] \vdash C[k'/k]$ and by omitting to consider buffer types as of Lemma 14;
- similarly to $\textcircled{2a}$, $\textcircled{2b}$ holds by α-renaming on the derivation of

  $$(\Gamma_i, \tilde{l}\colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle, \mathsf{init}\big(\ \widetilde{\mathsf{q}_i[\mathtt{B}_i]}, k, G\ \big))[k'/k][\mathsf{r}_i/\mathsf{q}_i] \vdash C_i[k'/k][\mathsf{r}_i/\mathsf{q}_i]$$

  and by Lemma 14;
- $\textcircled{2c}$ holds by (T2.1).

□

Proof of ③. The proof of ③ of this case is similar to the proof of ③ for Case $\lfloor^{\text{C}}|_{\text{START}}\rceil$. □

Case $\lfloor^{\text{C}}|_{\text{COND}}\rceil$
The case is:

$$\frac{i = 1 \text{ if } \mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{ true}, i = 2 \text{ otherwise}}{D,\ \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \quad \xrightarrow{\tau} \quad D,\ C_i}\ \lfloor^{\text{C}}|_{\text{COND}}\rceil$$

In (T2.5) $D' = D$ and we have two cases for $C' = C_1$ or $C' = C_2$.

From (T2.1) we can write

$$\frac{\Gamma \vdash \mathsf{p}.e\colon \mathbf{bool} \quad \Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}}\ \lfloor^{\text{T}}|_{\text{COND}}\rceil$$

The proof of (T2.6) follows directly from the premises of the typing derivation as $\Gamma \vdash D = D'$ and in both cases that $C' = C_1$ or $C' = C_2$ it holds that $\Gamma \vdash C'$ from the premises of $\lfloor^{\text{T}}|_{\text{COND}}\rceil$.

Case $\lfloor^{\text{C}}|_{\text{CTX}}\rceil$
The case is:

$$\frac{D, C_1 \xrightarrow{\beta} D', C_1'}{D, \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \quad \xrightarrow{\beta} \quad D', \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'}\ \lfloor^{\text{C}}|_{\text{CTX}}\rceil$$

From (T2.1) we know that, $\Gamma = \Gamma_1, X : \Gamma_x$

$$\frac{\mathsf{pco}(\Gamma) \quad \dfrac{\Gamma_1, X : \Gamma_x \vdash C_1 \quad \Gamma_x, X : \Gamma_x \vdash C_2 \quad \Gamma_x|_{\mathsf{locs}} \subseteq \Gamma}{\Gamma \vdash \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1}\ \lfloor^{\text{T}}|_{\text{DEF}}\rceil \quad \Gamma \vdash D}{\Gamma \vdash D, \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1}\ \lfloor^{\text{T}}|_{\text{DC}}\rceil$$

The proof is divided in two cases on the type of β.

Case $\beta \neq \tau$
D, C_1 reduces on some session k. By the induction hypothesis since $\Gamma \vdash D, C_1$

we can find Γ' such that (T2.3) holds. We prove (T2.4) by proving that we can apply $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ on $\Gamma' \vdash D', \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'$ and therefore that ① $\mathsf{pco}(\Gamma')$ holds, ② $\Gamma' \vdash \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1$ and ③ $\Gamma' \vdash D'$.

① holds by the construction of Γ' and ③ holds by the induction hypothesis. To prove ② we have to prove that $\Gamma' \vdash X : C_2$ and $\Gamma_x|_{\mathsf{locs}} \subseteq \Gamma'$.

From the induction hypothesis we have that $\Gamma \xrightarrow{\beta} \Gamma'$ and $\Gamma' \vdash D', C_1'$. By construction of Γ' it holds that $\Gamma' = \Gamma'_*, \Gamma'_k$ where $\Gamma' \cap \Gamma = \Gamma_*$ such that $k \notin \Gamma_*$ and $\Gamma = \Gamma_*, \Gamma_k$ where $\Gamma_k \subseteq [\![G]\!]_k$ for some G. Therefore it holds that $\Gamma_* \vdash X : \Gamma_x$ and thus that $\Gamma' \vdash X : \Gamma_x$. The same applies to $\Gamma_x|_{\mathsf{locs}} \subseteq \Gamma_*$ which proves $\Gamma_x|_{\mathsf{locs}} \subseteq \Gamma'$.

Case $\beta = \tau$
from the induction hypothesis for any considered derivation we have $\Gamma \subseteq \Gamma'$. We prove (T2.6) by proving that we can apply $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ on $\Gamma' \vdash D', \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'$. ①, ②, and ③ hold by construction of Γ'.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$
The case is:

$$\frac{D, C_1 \xrightarrow{\beta} D', C_1'}{D, C_1 \mid C_2 \xrightarrow{\beta} D', C_1' \mid C_2}\ \lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$$

From (T2.1) we have the derivation below, with Γ partitioned as $\Gamma = \Gamma_1 \circ \Gamma_2$

$$\frac{\mathsf{pco}(\Gamma) \qquad \dfrac{\Gamma_1 \vdash C_1 \quad \Gamma_2 \vdash C_2}{\Gamma \vdash C_1 \mid C_2}\ \lfloor^{\mathrm{T}}|_{\mathrm{PAR}}\rceil \qquad \Gamma \vdash D}{\Gamma \vdash D, C_1 \mid C_2}\ \lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$$

The proof is divided in two cases on the type of β.

Case $\beta \neq \tau$
D, C_1 reduces on some session k. By the induction hypothesis and since $\Gamma_1 \vdash D, C_1$ we can find Γ_1' such that $\Gamma_1 \xrightarrow{\beta} \Gamma_1'$ and $\Gamma_1' \vdash D', C_1'$. Then we take $\Gamma' = \Gamma_1' \circ \Gamma_2$ which proves (T2.3) to hold. We prove (T2.4) by proving that we can apply $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ on $\Gamma' \vdash D', C_1' \mid C_2$ and therefore that ① $\mathsf{pco}(\Gamma')$, ② $\Gamma' \vdash C_1' \mid C_2$ and ③ $\Gamma' \vdash D'$ hold. ①, ②, and ③ hold by construction and the induction hypothesis.

Case $\beta = \tau$
from the induction hypothesis, for any derivation we have that $\Gamma_1' \vdash D', C_1'$ and $\Gamma_1 \subseteq \Gamma_1'$. Also in this case we take $\Gamma' = \Gamma_1' \circ \Gamma_2$ and prove (T2.6) by proving that we can apply $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ on $\Gamma' \vdash D', C_1' \mid C_2$. ①, ②, and ③ hold by construction of Γ' and the induction hypothesis.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$
The case is:

$$\frac{\mathcal{R} \in \{\equiv, \simeq_{\mathsf{C}}\} \quad C_1 \,\mathcal{R}\, C_1' \quad D, C_1' \xrightarrow{\beta} D', C_2' \quad C_2' \,\mathcal{R}\, C_2}{D, C_1 \quad \xrightarrow{\beta} \quad D', C_2} \lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$$

The proof is divided into two subcases on the type of $\mathcal{R}$.

Case $\mathcal{R} \ = \equiv$
The case is proven by induction hypothesis and Lemma 16.

Case $\mathcal{R} \ = \simeq_{\mathsf{C}}$
The case is proven by induction hypothesis and Lemma 17.

□

The proof of Theorem 4 follows directly from the proof of Theorem 2 and Lemma 15.

E.2 Proof of Deadlock Freedom

We report below the statement of Theorem 5 enriched with pointers for clearer referencing the in the proof.
Theorem 5 (Deadlock-freedom) (D5.1) $\Gamma \vdash D, C$ and (D5.2) $\mathsf{co}(\Gamma)$ imply that either (D5.3) $C \equiv \mathbf{0}$ or (D5.4) there exist D' and C' such that $D, C \rightarrow D', C'$.
Like in [13, 14], our choreographies enjoy deadlock-freedom provided that they *i*) do not contain free variable names and *ii*) are *well-sorted*, i.e., have no undefined procedure calls. Notably, well-sortedness is guaranteed by the type system.

Proof. Proof by induction on the structure of C.

Case $C \equiv \mathbf{0}$
trivial.

Case $C = k : \mathtt{p}[\mathtt{A}].e$ -> $\mathtt{B}.o; C_1$
from (D5.1) and (D5.2) we know that the requirements of $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$ hold and we can find D' such that $D, k : \mathtt{p}[\mathtt{A}].e$ -> $\mathtt{B}.o \blacktriangleright D'$. We can apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ for which $C' = C_1$.

Case $C = k : \mathtt{p}[\mathtt{A}].e$ -> $\mathtt{q}[\mathtt{B}].o(x); C_1$
since (D5.1) holds both receiver and sender are typed by Γ. Similarly to the previous case, we apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$ for which $C' = k : \mathtt{A}$ -> $\mathtt{q}[\mathtt{B}].o(x)$.

Case $C = k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I}$
from (D5.1) and (D5.2) we know that the requirements of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ hold and $D(\mathsf{q}).\mathsf{que}(\underline{\mathbf{k.A.B}}(D(\mathsf{q}).\mathsf{st})) = (o_j, t_m) :: \tilde{m}$ for some $j \in I$. We can find D' such that $D, k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o_j(x_j) \blacktriangleright D'$ and apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ for which $C' = C_j$.

Case $C = \mathbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C_1$
from (D5.1) and (D5.2) $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$ applies and we can find D' such that $D, \mathbf{start}\ k' : l.\mathsf{p}[\mathtt{A}], \widetilde{l.\mathsf{r}[\mathtt{B}]} \blacktriangleright D'$ for some k', $\tilde{\mathsf{r}}$ fresh. We can apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{START}}\rceil$ for which $C' = C_1[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]$.

Case $C = \mathbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C \mid \prod_{i=1}^{n} \big(\mathbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)$
similarly to the previous case the requirements of $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$ hold and we can find D' such that $D, \mathbf{start}\ k' : l.\mathsf{p}[\mathtt{A}], \widetilde{l_1.\mathsf{r}_1[\mathtt{B}_1]}, \ldots, \widetilde{l_n.\mathsf{r}_n[\mathtt{B}_n]} \blacktriangleright D'$ for some k' and $\tilde{\mathsf{r}}_1, \cdots, \tilde{\mathsf{r}}_n$ fresh. We can apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$ for which
$C' = C[k'/k] \mid \prod_{i=1}^{n} C_i[k'/k][\tilde{\mathsf{r}}_1/\tilde{\mathsf{q}}_1] \mid \prod_{i=1}^{n} \big(\mathbf{acc}\ k : \widetilde{l_i.\mathsf{q}_i[\mathtt{B}_i]}; C_i\big)$.

Case $C = C_1 \mid C_2$
we can apply the induction hypothesis and Rule $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$ such that $D, C_1 \to D_1, C_1'$ and in (D5.4) $D' = D_1$ and $C' = C_1' \mid C_2$.

Case $C = \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1$
applies the induction hypothesis and Rule $\lfloor^{\mathrm{C}}|_{\mathrm{CTX}}\rceil$ for which $D, C_1 \to D_1, C_1$, $D' = D_1$ and $C' = \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'$.

Case $\mathsf{def}\ X = C_2\ \mathsf{in}\ X; C_1$
applies Rule $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$ for $\mathsf{def}\ X = C_2\ \mathsf{in}\ X; C_1 \equiv \mathsf{def}\ X = C_2\ \mathsf{in}\ C_2; C_1$ and by the induction hypothesis $D, C_2 \to D_2, C_2'$ and therefore $D' = D_2$ and $C' = \mathsf{def}\ X = C_2\ \mathsf{in}\ C_2'; C_1$.

Case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}$
from (D5.1) we know that $\Gamma \vdash \mathsf{p}.e\colon \mathbf{bool}$ and therefore we can apply Rule $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$ and, according to the evaluation of e we have $C' = C_1$ or $C' = C_2$.

$\square$

E.3 Proof of Endpoint Projection

To prove our result on the Endpoint Projection we first define the minimal typing system $\vdash_{\mathsf{min}}$ for AC.

E.3.1 Minimal Typing

We take the definition of Local and Typing Environment Subtyping from Section E.1.1 and we define the subtyping for global types $G \prec G'$.

Definition 34 (Global and Local Subtyping)**.** *$G \prec G'$ is the smallest relation over closed and unfolded global types satisfying the rules below*

$$\frac{I \subseteq J \quad \forall\, i \in I,\ G_i \prec G'_i\ \wedge U_i \prec U'_i}{\mathtt{A} \texttt{ -> } \mathtt{B}.\{o_i(U_i); G_i\}_{i\in I} \prec \mathtt{A} \texttt{ -> } \mathtt{B}.\{o_j(U'_j); G'_j\}_{j\in J}} \ \lfloor^{SubG}|_{\text{COM}}\rceil$$

$$\frac{U \prec U' \quad G \prec G'}{\mathtt{A} \ \rightsquigarrow\ \mathtt{B}.o(U); G \prec \mathtt{A} \ \rightsquigarrow\ \mathtt{B}.o(U'); G'} \ \lfloor^{SubG}|_{\text{RECV}}\rceil$$

$$\frac{G'' \prec G' \quad (G'' \approx G \ \vee\ G'' \simeq_{\mathsf{G}} G)}{G \prec G'} \ \lfloor^{SubG}|_{\text{EQ}}\rceil \qquad \frac{\mathsf{end} \approx G}{\mathsf{end} \prec G} \ \lfloor^{SubG}|_{\text{END}}\rceil$$

We extend the subtyping relations for local and global types to set inclusion and point-wise to *i*) the typing of services (i.e., of kind $\tilde{l}\colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$) and *ii*) the typing of sessions, respectively. Given two types G and G', we denote their least upper bound wrt $\prec$ with $G \triangledown G'$ (the same for local types and typing environments). Our definition of subtyping for global types follows [14].

We define the minimal typing system $\vdash_{\mathsf{min}}$ on this notion of subtyping. The minimal typing uses the minimal global and local types for typing sessions and services such that the projection of the choreography is still typable. We report the rules for minimal typing in Figure E.2.

E.3.2 Typing Projection

Like in [14] our EPP projects recursive definitions of the same procedure from the point of view of different processes, therefore the same definition has different types according to the process on which it has been projected. Similarly to [14], to deal with this discrepancy we define the projection of environments. Note that we give the definition of $[\![\Gamma]\!]$ over annotated ACs, i.e., ACs where any procedure X is annotated, e.g., $X\langle\tilde{\mathsf{p}}\rangle$, where $\tilde{\mathsf{p}}$ are the process identifiers of the active processes in the body of X. Formally, given a choreography C it is always possible to have its annotated version with operator $@C$, defined following the rules in Figure D.5.

Definition 35 (Typing Projection)**.** *Let $\Gamma = \Gamma', \Gamma_{\mathsf{def}}$ where $\nexists\ X\colon \Gamma^* \in \Gamma'$ and $\Gamma_{\mathsf{def}} = \Gamma/\Gamma'$ (Γ_{def} contains only typings of recursive procedures). The projection of Γ, written $[\![\Gamma]\!]$, is defined as:*

$$[\![\Gamma]\!] = \Gamma', \left\{ X\langle\mathsf{p}\rangle\colon\ [\![\Gamma_x]\!]_{\mathsf{p}} \mid X\langle\tilde{\mathsf{r}}\rangle\colon \Gamma_x \ \in\ \Gamma \ \wedge\ \mathsf{p} \in \{\tilde{\mathsf{r}}\} \right\}$$

$$\frac{\Gamma, \mathsf{init}\big(\,\widetilde{\mathsf{r}[\mathtt{C}]}, k, G\,\big) \vdash_{\mathsf{min}} C \quad \widetilde{\mathsf{r}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash_{\mathsf{min}} \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{START1}}\rceil$$

$$\frac{\Gamma, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \mathsf{init}\big(\,\widetilde{\mathsf{r}[\mathtt{C}]}, k, G'\,\big) \vdash_{\mathsf{min}} C \quad \widetilde{\mathsf{r}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, \tilde{l} \colon G \triangledown G'\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash_{\mathsf{min}} \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{START2}}\rceil$$

$$\frac{\Gamma, \mathsf{p} : k[\mathtt{A}], k[\mathtt{A}] : [\![G]\!]_{\mathtt{A}} \vdash_{\mathsf{min}} C \quad \Gamma \vdash \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle}{\Gamma \vdash_{\mathsf{min}} \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{REQ}}\rceil$$

$$\frac{\tilde{l} \subseteq \tilde{l}' \quad \Gamma, \tilde{l}' \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle, \Gamma' \vdash_{\mathsf{min}} C \quad \Gamma' \prec \mathsf{init}\big(\,\widetilde{\mathsf{q}[\mathtt{C}]}, k, G\,\big)}{\Gamma, \tilde{l}' \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \vdash_{\mathsf{min}} \textbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{C}]}; C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{ACC}}\rceil$$

$$\frac{\Gamma_1 \triangledown \Gamma_2 \vdash \mathsf{p}.e \colon \textbf{bool} \quad \Gamma_1 \vdash_{\mathsf{min}} C_1 \quad \Gamma_2 \vdash_{\mathsf{min}} C_2}{\Gamma_1 \triangledown \Gamma_2 \vdash_{\mathsf{min}} \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}} \ \lfloor^{\mathrm{Min}}|_{\mathrm{COND}}\rceil$$

$$\frac{\Gamma \vdash \mathsf{p} \colon k[\mathtt{A}], \mathsf{q} \colon k[\mathtt{B}] \quad \Gamma \vdash \mathsf{p}.e \colon U \quad \Gamma, \mathsf{q}.x \colon U, k[\mathtt{A}] \colon T, k[\mathtt{B}] \colon T' \vdash_{\mathsf{min}} C}{\Gamma, k[\mathtt{A}] \colon !\mathtt{B}.\{o(U); T\}, k[\mathtt{B}] \colon ?\mathtt{A}.\{o(U); T'\} \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathsf{q}[\mathtt{B}].o(x); C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{COM}}\rceil$$

$$\frac{j \in I \quad \Gamma \vdash \mathsf{p} \colon k[\mathtt{A}] \quad \mathsf{q} \colon k[\mathtt{B}] \notin \Gamma \quad \Gamma \vdash \mathsf{p}.e \colon U_j \quad \Gamma, k[\mathtt{A}] \colon T_j \vdash_{\mathsf{min}} C}{\Gamma, k[\mathtt{A}] \colon !\mathtt{B}.\{o_i(U_i); T_i\}_{i \in I} \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o_j; C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{SEND}}\rceil$$

$$\frac{\Gamma \vdash \mathsf{q} \colon k[\mathtt{B}] \quad \mathsf{p} \colon k[\mathtt{A}] \notin \Gamma \quad \forall\, i \in I.\ \Gamma, \mathsf{q}.x_i \colon U_i, k[\mathtt{B}] \colon T_i \vdash C_i}{\Gamma, k[\mathtt{B}] \colon ?\mathtt{A}.\{o_i(U_i); T_i\}_{i \in I} \vdash k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I}} \ \lfloor^{\mathrm{Min}}|_{\mathrm{RECV}}\rceil$$

$$\frac{\Gamma_1 \vdash_{\mathsf{min}} C_1 \quad \Gamma_2 \vdash_{\mathsf{min}} C_2}{\Gamma_1 \circ \Gamma_2 \vdash_{\mathsf{min}} C_1 \mid C_2} \ \lfloor^{\mathrm{T}}|_{\mathrm{PAR}}\rceil \qquad \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash_{\mathsf{min}} \mathbf{0}} \ \lfloor^{\mathrm{T}}|_{\mathrm{END}}\rceil$$

$$\frac{\Gamma, X \colon \Gamma' \vdash_{\mathsf{min}} C \quad \Gamma', X \colon \Gamma' \vdash_{\mathsf{min}} C' \quad \Gamma'|_{\mathsf{locs}} \subseteq \Gamma}{\Gamma \triangledown \Gamma' \vdash_{\mathsf{min}} \mathsf{def}\ X = C'\ \mathsf{in}\ C} \ \lfloor^{\mathrm{T}}|_{\mathrm{DEF}}\rceil$$

$$\frac{\mathsf{end}(\Gamma) \quad \Gamma'' \subseteq \Gamma'}{\Gamma, \Gamma'', X \colon \Gamma' \vdash_{\mathsf{min}} X} \ \lfloor^{\mathrm{T}}|_{\mathrm{CALL}}\rceil \qquad \frac{\mathsf{pco}(\Gamma) \quad \Gamma \vdash_{\mathsf{min}} D \quad \Gamma \vdash_{\mathsf{min}} C}{\Gamma \vdash_{\mathsf{min}} D, C} \ \lfloor^{\mathrm{Min}}|_{\mathrm{DC}}\rceil$$

Figure E.2: Choreography Calculus — Minimal typing rules

where

$$[\![\Gamma]\!]_{\mathsf{p}} = \begin{array}{l} \{\ \mathsf{p}.x : U \mid \mathsf{p}.x : U \in \Gamma\ \} \cup \\ \{\ \mathsf{p} : k[\mathtt{A}], k[\mathtt{A}] : T \mid \{\mathsf{p} : k[\mathtt{A}], k[\mathtt{A}] : T\} \subseteq \Gamma\ \} \cup \\ \{\ X\langle \mathsf{p}\rangle : [\![\Gamma']\!]_{\mathsf{p}} \mid X\langle \tilde{\mathsf{r}}\rangle : \Gamma' \in \Gamma\ \wedge\ \mathsf{p} \in \{\tilde{\mathsf{r}}\}\ \} \cup \\ \{ l : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \mid l : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \in \Gamma \} \end{array}$$

Notably, in the definition of $[\![\Gamma]\!]_{\mathsf{p}}$ we omit to include in the typing of procedures *buffer types* ($b[k]^{\mathtt{A}}_{\mathtt{B}}$), *process locations* ($\mathsf{p}@l$), since they are not used in the typing of C. Contrarily, we include all service typings present in the typings of procedures, although they might not be minimal wrt the process that is object of the projection.

E.3.3 Proof of Theorem 6

We define some auxiliary lemmas used in the proof of Theorem 6.

Lemma 18 (Weakening)**.** *Let* $\Gamma \vdash_{\mathsf{min}} C$ *and* $\tilde{l}$ *be a service name such that* $\tilde{l} \notin \Gamma$ *then* $\Gamma, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle \vdash C$ *for any* $G, \mathtt{A}, \tilde{\mathtt{B}}, \tilde{\mathtt{C}}$.

Proof. Immediate since $\tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$ is never used in C. □

Lemma 19 (Composability of Typing Projections)**.** *Let* $\Gamma \circ \Gamma' = \Gamma''$ *then* $[\![\Gamma]\!] \circ [\![\Gamma']\!] = [\![\Gamma'']\!]$.

Proof. Directly from the fact that since the projection $[\![\Gamma]\!]$ returns exactly Γ except for the projection of the typings of the procedures. Since $\Gamma \circ \Gamma'$ is applicable, all procedure typings in common are equal and the same holds for their projections. □

We proceed defining well-annotated choreographies. Well annotated choreographies are ACs in which annotated procedures report the names of the active processes in their bodies.

Definition 36 (Well-annotated Choreographies)**.** *Let* C *be an annotated choreography,* C *is well-annotated iff*

$$\forall\ X\langle \tilde{\mathsf{p}}\rangle\ s.t\ C \equiv \mathsf{def}\ X\langle \tilde{\mathsf{p}}\rangle = C''\ \mathsf{in}\ C',\ \tilde{\mathsf{p}} = \mathsf{fp}(C'')$$

We prove Lemma 20 that states that given a well-annotated choreography C and a typing environment Γ for which $\Gamma \vdash_{\mathsf{min}} C$ then the projection of Γ, $[\![\Gamma]\!]$ types minimally the projection of C, $[\![C]\!]$.

Lemma 20 (Choreography EPP Typing Preservation)**.** *Let* C *be a well-annotated choreography and let* $\Gamma \vdash_{\mathsf{min}} C$ *then* $[\![\Gamma]\!] \vdash_{\mathsf{min}} [\![C]\!]$.

Proof. Like for the proof of Theorem 5, we assume our choreographies to be well-sorted. The proof is by induction on the typing derivation of $\Gamma \vdash_{\mathsf{min}} C$.

Case $\lfloor^{\mathrm{Min}}|_{\mathrm{START1}}\rceil$
From the premises we have $C = \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C'$. We can partition $\Gamma = \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \Gamma'$ and we can write the derivation

$$\dfrac{\Gamma', \mathsf{init}\big(\widetilde{\mathsf{r}[\mathtt{C}]}, k, G\big) \vdash_{\mathsf{min}} C' \quad \widetilde{\mathsf{r}[\mathtt{C}]} = \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \quad \tilde{\mathsf{q}} \notin \Gamma'}{\Gamma', \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash_{\mathsf{min}} \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C'}\ \lfloor^{\mathrm{Min}}|_{\mathrm{START1}}\rceil$$

Let $\widetilde{l.\mathsf{q}[\mathtt{B}]} = l_1.\mathsf{q}_1.[\mathtt{B}_1], \cdots, l_n.\mathsf{q}_n.[\mathtt{B}_n]$.

Let $\Gamma_c = \Gamma', \mathsf{init}\big(\widetilde{\mathsf{r}[\mathtt{C}]}, k, G\big)$, from the induction hypothesis we have that $\Gamma_c \vdash_{\mathsf{min}} C'$ and therefore $[\![\Gamma_c]\!] \vdash_{\mathsf{min}} [\![C']\!]$.

By its definition $[\![C']\!] \equiv C'_s \mid C''$ where

$$C'_s = [\![C']\!]_{\mathsf{p}} \mid [\![C']\!]_{\mathsf{q}_1} \mid \ldots \mid [\![C']\!]_{\mathsf{q}_n}$$

and

$$C'' = \prod_{\mathsf{r}\,\in\,\mathsf{fp}(C')/\{\mathsf{p},\tilde{\mathsf{q}}\}} [\![C']\!]_{\mathsf{r}} \quad \mid \quad \prod_{l} \left(\bigsqcup_{\mathsf{s}\,\in\,\lfloor C' \rfloor_l} [\![C']\!]_{\mathsf{s}} \right)$$

We partition $[\![\Gamma_c]\!]$ as

$$[\![\Gamma_c]\!] = \Gamma'_{\mathsf{p}} \circ \Gamma_{\tilde{\mathsf{q}}} \circ \Gamma''$$

where

$$\Gamma'_{\mathsf{p}} = \Gamma_{\mathsf{p}}, \mathsf{p} \colon k[\mathtt{A}], k[\mathtt{A}] \colon [\![G]\!]_{\mathsf{p}}$$

and

$$\Gamma_{\tilde{\mathsf{q}}} = \Gamma'_{\mathsf{q}_1} \circ \ldots \circ \Gamma'_{\mathsf{q}_n}$$

where

$$\Gamma'_{\mathsf{q}_i} = \Gamma_{\mathsf{q}_i}, \mathsf{q}_i \colon k[\mathtt{A}], k[\mathtt{A}] \colon [\![G]\!]_{\mathsf{q}_i}$$

. We such that we can write the derivation

$$\dfrac{\Gamma'' \vdash_{\mathsf{min}} C'' \qquad \dfrac{\Gamma'_{\mathsf{p}} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{p}} \qquad \dfrac{\Gamma'_{\mathsf{q}_1} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{q}_1} \qquad \dfrac{\vdots}{\Gamma'_{\mathsf{q}_2} \circ \ldots \circ \Gamma_{\mathsf{q}_n} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{q}_2} \mid \ldots \mid [\![C']\!]_{\mathsf{q}_n}}\ \lfloor^{\mathrm{Min}}|_{\mathrm{PAR}}\rceil}{\Gamma'_{\mathsf{q}_1} \circ \Gamma'_{\mathsf{q}_2} \circ \ldots \circ \Gamma'_{\mathsf{q}_n} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{q}_1} \mid \ldots \mid [\![C']\!]_{\mathsf{q}_n}}\ \lfloor^{\mathrm{Min}}|_{\mathrm{PAR}}\rceil}{\Gamma'_{\mathsf{p}} \circ \Gamma_{\tilde{\mathsf{q}}} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{p}} \mid [\![C']\!]_{\mathsf{q}_1} \mid \ldots \mid [\![C']\!]_{\mathsf{q}_n}}\ \lfloor^{\mathrm{Min}}|_{\mathrm{PAR}}\rceil}{\Gamma'' \circ \Gamma'_{\mathsf{p}} \circ \Gamma_{\tilde{\mathsf{q}}} \vdash_{\mathsf{min}} C'' \mid C'_s}\ \lfloor^{\mathrm{Min}}|_{\mathrm{PAR}}\rceil$$

Since the ownership and session typings for k in Γ_c belong to $\mathsf{init}\big(\widetilde{\mathsf{r}[\mathtt{C}]}, k, G\big)$[1] we have that $\Gamma'' \circ \Gamma_{\mathsf{p}} \circ \Gamma_{\mathsf{q}_1} \circ \ldots \circ \Gamma_{\mathsf{q}_n} = [\![\Gamma']\!]$ by Lemma 19.

Therefore we can partition $[\![\Gamma]\!]$ as (using Lemma 18 for Γ_{p})

$$[\![\Gamma]\!] = \Gamma'' \circ \Gamma_{\mathsf{p}}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \circ \Gamma_{\mathsf{q}_1}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_1\rangle \circ \ldots \circ \Gamma_{\mathsf{q}_n} \circ \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_n\rangle$$

Let $\widetilde{l.\mathsf{q}[\mathtt{B}]}\Big|_i = \{l_i.\mathsf{q}_i[\mathtt{B}_i], \ldots, l_n.\mathsf{q}_n[\mathtt{B}_n]\}$.

We prove the case by proving the typing derivation for $[\![\Gamma]\!] \vdash_{\mathsf{min}} [\![C]\!]$

From the definition of EPP (Definition 22) we can write

$$[\![C]\!] \equiv C_s \mid C''$$

where

$$C_s = \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; [\![C']\!]_{\mathsf{p}} \quad \mid \quad \prod_{l.\mathsf{r}[\mathtt{C}]\ \in\ \{\widetilde{l.\mathsf{q}[\mathtt{B}]}\}} \textbf{acc}\ k : l.\mathsf{r}[\mathtt{C}]; [\![C']\!]_{\mathsf{r}}$$

and

$$C'' = \prod_{\mathsf{r}\ \in\ \mathsf{fp}(C')/\{\mathsf{p}\}} [\![C']\!]_{\mathsf{r}} \quad \mid \quad \prod_{l' \neq l} \left(\bigsqcup_{\mathsf{s}\ \in\ \lfloor C' \rfloor_{l'}} [\![C']\!]_{\mathsf{s}} \right)$$

We now prove we can derive the typing of $[\![\Gamma]\!] \vdash_{\mathsf{min}} [\![C]\!]$

$$\dfrac{\Gamma'' \vdash_{\mathsf{min}} C'' \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma_{\mathsf{p}}, \mathsf{p} \colon k[\mathtt{A}], k[\mathtt{A}] \colon [\![G]\!]_{\mathtt{A}} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{p}} \\ \Gamma_{\mathsf{p}}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \vdash \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle\end{array}}{\Gamma_{\mathsf{p}}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \vdash_{\mathsf{min}} \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; [\![C']\!]_{\mathsf{p}}}\lfloor^{\text{Min}}|_{\text{REQ}}\rceil \qquad \Delta_1}{\Gamma_{\mathsf{p}}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \circ \Gamma_{\mathsf{q}_1}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_1\rangle \circ \ldots \circ \Gamma_{\mathsf{q}_n} \circ \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_n\rangle \vdash_{\mathsf{min}} C_s}\lfloor^{\text{Min}}|_{\text{PAR}}\rceil}{\Gamma'' \circ \Gamma_{\mathsf{p}}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \circ \Gamma_{\mathsf{q}_1}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_1\rangle \circ \ldots \circ \Gamma_{\mathsf{q}_n} \circ \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_n\rangle \vdash_{\mathsf{min}} C_s \mid C''}\lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

where

$$\Delta_i = \dfrac{\Delta_{i+1} \qquad \dfrac{\begin{array}{c} l_i \subseteq \tilde{l} \qquad \Gamma_{\mathsf{q}_i}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \circ \Gamma''_{\mathsf{q}_i} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{q}_i} \\ \Gamma''_{\mathsf{q}_i} \prec \mathsf{init}\big(\mathsf{q}_i[\mathtt{B}_i], k, G\big) \qquad \mathsf{q}_i \notin \Gamma_{\mathsf{q}_i}\end{array}}{\Gamma_{\mathsf{q}_i}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_i\rangle \vdash_{\mathsf{min}} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; [\![C']\!]_{\mathsf{q}_i}}\lfloor^{\text{Min}}|_{\text{ACC}}\rceil}{\begin{array}{c}\Gamma_{\mathsf{q}_i}, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_i\rangle \circ \ldots \circ \Gamma_{\mathsf{q}_n} \circ \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\mathtt{B}_n\rangle \\ \vdash_{\mathsf{min}} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; [\![C']\!]_{\mathsf{q}_i} \mid \displaystyle\prod_{l.\mathsf{r}[\mathtt{C}]\ \in\ \widetilde{l.\mathsf{q}[\mathtt{B}]}\big|_{i+1}} \textbf{acc}\ k : l.\mathsf{r}[\mathtt{C}]; [\![C']\!]_{\mathsf{r}}\end{array}}\lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

In particular we prove

[1]we omit buffer types as of Lemma 14.

① $\Gamma'' \vdash_{\mathsf{min}} C''$ and ② $\Gamma_{\mathsf{p}}, \mathsf{p}\colon k[\mathtt{A}], k[\mathtt{A}]\colon [\![G]\!]_{\mathtt{A}} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{p}}$ which hold by the induction hypothesis;

③ $\Gamma_{\mathsf{q}_i}, \tilde{l}\colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \circ \Gamma''_{\mathsf{q}_i} \vdash_{\mathsf{min}} [\![C']\!]_{\mathsf{q}_i}$ which holds by the induction hypothesis and Lemma 18;

④ $\Gamma''_{\mathsf{q}_i} \prec \mathsf{init}\big(\mathsf{q}_i[\mathsf{B}_i], k, G\big)$ holds as of Definition 33 and because, for Γ'_{q_i}, $i \in \{1, \ldots, n\}$, $\mathsf{init}\big(\mathsf{q}_i[\mathsf{B}_i], k, G\big) \subseteq \Gamma'_{\mathsf{q}_i}$.

Case $\lfloor^{\text{Min}}|_{\text{START2}}\rceil$
Similar to case $\lfloor^{\text{Min}}|_{\text{START1}}\rceil$.

Case $\lfloor^{\text{Min}}|_{\text{REQ}}\rceil$
Follows the proof of ② of case $\lfloor^{\text{Min}}|_{\text{START1}}\rceil$.

Case $\lfloor^{\text{Min}}|_{\text{ACC}}\rceil$
Follows the proof of ③ of case $\lfloor^{\text{Min}}|_{\text{START1}}\rceil$.

Case $\lfloor^{\text{Min}}|_{\text{COND}}\rceil$
By induction hypothesis on C_1 or C_2.

Case $\lfloor^{\text{Min}}|_{\text{COM}}\rceil$
From the premises we have $C = k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathsf{q}[\mathtt{B}].o(x); C'$ on which we can apply the typing derivation

$$\frac{\Gamma' \vdash \mathsf{p}\colon k[\mathtt{A}], \mathsf{q}\colon k[\mathtt{B}] \quad \Gamma' \vdash \mathsf{p}.e\colon U \quad \Gamma', \mathsf{q}.x\colon U, k[\mathtt{A}]\colon T, k[\mathtt{B}]\colon T' \vdash_{\mathsf{min}} C'}{\Gamma', k[\mathtt{A}]\colon !\mathtt{B}.o(U); T, k[\mathtt{B}]\colon ?\mathtt{A}.o(U); T' \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathsf{q}[\mathtt{B}].o(x); C'} \lfloor^{\text{Min}}|_{\text{COM}}\rceil$$

From the definition of EPP (Definition 22) we have $[\![C]\!] \equiv C_c \mid C''$ where

$$C_c = k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; [\![C']\!]_{\mathsf{p}} \;\mid\; k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].o(x); [\![C']\!]_{\mathsf{q}}$$

$$C'' = \prod_{\mathsf{r} \,\in\, \{\mathsf{fp}(C')/\{\mathsf{p},\mathsf{q}\}\}} [\![C']\!]_{\mathsf{r}} \;\mid\; \prod_{l} \left(\bigsqcup_{\mathsf{s} \,\in\, \lfloor C' \rfloor_l} [\![C']\!]_{\mathsf{s}} \right)$$

By the definition of $[\![\Gamma]\!]$ we can write

$$[\![\Gamma]\!] = [\![\Gamma']\!], k[\mathtt{A}]\colon !\mathtt{B}.o(U); T, k[\mathtt{A}]\colon ?\mathtt{A}.o(U); T'$$

from the induction hypothesis we have that, let $\Gamma_c = \Gamma', \mathsf{q}.x\colon U, k[\mathtt{A}]\colon T, k[\mathtt{B}]\colon T'$, $\Gamma_c \vdash_{\mathsf{min}} C'$ and therefore $[\![\Gamma_c]\!] \vdash_{\mathsf{min}} [\![C']\!]$. We can partition $[\![\Gamma_c]\!]$ as

$$\llbracket \Gamma_c \rrbracket = \Gamma_{\mathsf{p}}, k[\mathtt{A}] : T \circ \Gamma_{\mathsf{q}}, \mathsf{q}.x : U, k[\mathtt{B}] : T' \circ \Gamma''$$

such that

$$\dfrac{\Gamma'' \vdash_{\mathsf{min}} C'' \qquad \dfrac{\Gamma_{\mathsf{p}}, k[\mathtt{A}] : T \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{p}} \qquad \Gamma_{\mathsf{q}}, \mathsf{q}.x : U, k[\mathtt{B}] : T' \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{q}}}{\Gamma_{\mathsf{p}}, k[\mathtt{A}] : T \circ \Gamma_{\mathsf{q}}, \mathsf{q}.x : U, k[\mathtt{B}] : T' \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{p}} \mid \llbracket C' \rrbracket_{\mathsf{q}}} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil}{\Gamma'' \circ \Gamma_{\mathsf{p}}, k[\mathtt{A}] : T \circ \Gamma_{\mathsf{q}}, \mathsf{q}.x : U k[\mathtt{B}] : T' \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{p}} \mid \llbracket C' \rrbracket_{\mathsf{q}} \mid C''} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

From the derivation on Rule $\lfloor^{\text{Min}}|_{\text{COM}}\rceil$ we know that

$$\llbracket \Gamma' \rrbracket = \Gamma'' \circ \Gamma_{\mathsf{p}} \circ \Gamma_{\mathsf{q}}$$

and therefore that

$$\llbracket \Gamma \rrbracket = \Gamma'' \circ \Gamma_{\mathsf{p}}, k[\mathtt{A}] : !\mathtt{B}.o(U); T \circ \Gamma_{\mathsf{q}}, k[\mathtt{B}] : !\mathtt{A}.o(U); T'$$

To prove $\llbracket \Gamma \rrbracket \vdash_{\mathsf{min}} \llbracket C \rrbracket$ we prove that we can apply Rule $\lfloor^{\text{Min}}|_{\text{PAR}}\rceil$ on it.

$$\dfrac{\Gamma'' \vdash_{\mathsf{min}} C'' \qquad \dfrac{\dfrac{\begin{matrix}\Gamma_{\mathsf{p}} \vdash p : k[A] \quad \mathsf{q} : k[\mathtt{B}] \notin \Gamma_{\mathsf{q}} \\ \Gamma_{\mathsf{p}} \vdash \mathsf{p}.e : U \\ \Gamma_{\mathsf{p}}, k[\mathtt{A}] : T \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{p}}\end{matrix}}{\begin{matrix}\Gamma_{\mathsf{p}}, k[\mathtt{A}] : !\mathtt{B}.o(U); T \\ \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; \llbracket C' \rrbracket_{\mathsf{p}}\end{matrix}} \lfloor^{\text{Min}}|_{\text{SEND}}\rceil \quad \dfrac{\begin{matrix}\Gamma_{\mathsf{q}} \vdash \mathsf{p} : k[\mathtt{B}] \qquad \mathsf{p} : k[\mathtt{A}] \notin \Gamma_{\mathsf{q}} \\ \Gamma_{\mathsf{q}}, \mathsf{q}.x : U, k[\mathtt{A}] : T' \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{q}}\end{matrix}}{\begin{matrix}\Gamma_{\mathsf{p}}, k[\mathtt{B}] : ?\mathtt{A}.o(U); T' \\ \vdash_{\mathsf{min}} k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].o(x); \llbracket C' \rrbracket_{\mathsf{p}}\end{matrix}} \lfloor^{\text{Min}}|_{\text{RECV}}\rceil}{\begin{matrix}\Gamma_{\mathsf{p}}, k[\mathtt{A}] : !\mathtt{B}.o(U); T \circ \Gamma_{\mathsf{q}}, k[\mathtt{B}] : !\mathtt{A}.o(U); T' \\ \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; \llbracket C' \rrbracket_{\mathsf{p}} \mid k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].o(x); \llbracket C' \rrbracket_{\mathsf{q}}\end{matrix}} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil}{\begin{matrix}\Gamma'' \circ \Gamma_{\mathsf{p}}, k[\mathtt{A}] : !\mathtt{B}.o(U); T \circ \Gamma_{\mathsf{q}}, k[\mathtt{B}] : !\mathtt{A}.o(U); T' \\ \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; \llbracket C' \rrbracket_{\mathsf{p}} \mid k : \mathtt{A} \texttt{->} \mathsf{q}[\mathtt{B}].o(x); \llbracket C' \rrbracket_{\mathsf{q}} \mid C''\end{matrix}} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

Case $\lfloor^{\text{Min}}|_{\text{SEND}}\rceil$
From the premises we have $C = k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; C'$ on which we can apply the typing derivation

$$\dfrac{j \in I \quad \Gamma' \vdash \mathsf{q} : k[\mathtt{A}] \quad \mathsf{q} : k[\mathtt{B}] \notin \Gamma' \quad \Gamma' \vdash \mathsf{p}.e : U_j \quad \Gamma', k[\mathtt{A}] : T_j \vdash_{\mathsf{min}} C'}{\Gamma', k[\mathtt{A}] : !\mathtt{B}.\{o_i(U_i); T_i\}_{i \in I} \vdash_{\mathsf{min}} k : \mathsf{p}[\mathtt{A}].e \texttt{->} \mathtt{B}.o; C'} \lfloor^{\text{Min}}|_{\text{SEND}}\rceil$$

the case is proven by applying the induction hypothesis similarly to case $\lfloor^{\text{Min}}|_{\text{COM}}\rceil$.

Case $\lfloor^{\text{Min}}|_{\text{RECV}}\rceil$
Analogous to case $\lfloor^{\text{Min}}|_{\text{SEND}}\rceil$.

Case $\lfloor^{\text{Min}}|_{\text{PAR}}\rceil$
From the premises we know that $C = C_1 \mid C_2$ on which we can apply the typing derivation

$$\frac{\Gamma_1 \vdash_{\mathsf{min}} C_1 \quad \Gamma_2 \vdash_{\mathsf{min}} C_2}{\Gamma_1 \circ \Gamma_2 \vdash_{\mathsf{min}} C_1 \mid C_2} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

the case is proven by Lemma 18 and the induction hypothesis.

Case $\lfloor^{\text{Min}}|_{\text{END}}\rceil$
Trivial.

Case $\lfloor^{\text{Min}}|_{\text{DEF}}\rceil$
From the premises we know that $C = \mathsf{def}\ X\langle\tilde{\mathsf{r}}\rangle = C''\ \mathsf{in}\ C'$ on which we can apply the typing derivation, with $\Gamma = \Gamma_d \nabla \Gamma'$

$$\frac{\Gamma', X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d \vdash_{\mathsf{min}} C' \quad \Gamma_d, X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d \vdash_{\mathsf{min}} C'' \quad \Gamma_d|_{\mathsf{locs}} \subseteq \Gamma'}{\Gamma_d \nabla \Gamma' \vdash_{\mathsf{min}} \mathsf{def}\ X\langle\tilde{\mathsf{r}}\rangle = C''\ \mathsf{in}\ C'} \lfloor^{\text{Min}}|_{\text{DEF}}\rceil$$

By its definition

$$[\![\mathsf{def}\ X\langle\tilde{\mathsf{r}}\rangle = C''\ \mathsf{in}\ C']\!] = C_d \mid C'''$$

Where

$$C_d = \prod_{\mathsf{p}\,\in\,\tilde{\mathsf{r}}} \mathsf{def}\ X_{\mathsf{p}} = [\![C'']\!]_{\mathsf{p}}\ \ \mathsf{in}\ \ [\![C']\!]_{\mathsf{p}}$$

and

$$C''' = \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C')/\{\tilde{\mathsf{r}}\}} [\![C']\!]_{\mathsf{q}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s}\,\in\,\lfloor C'\rfloor_l} [\![C']\!]_{\mathsf{s}} \right)$$

From the induction hypothesis we know that $[\![\Gamma', X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!] \vdash_{\mathsf{min}} [\![C']\!]$.

Let $[\![\Gamma', X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!] = [\![\Gamma']\!] \circ [\![X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!] = [\![\Gamma']\!] \circ X_{\mathsf{p}_1} : [\![\Gamma_d]\!]_{\mathsf{p}_1} \circ \cdots X_{\mathsf{p}_n} : [\![\Gamma_d]\!]_{\mathsf{p}_n}$ for $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\} = \{\tilde{\mathsf{r}}\}$ where $[\![\Gamma']\!]$ can be partitioned as $[\![\Gamma']\!] = \Gamma'' \circ \Gamma_d^{\mathsf{p}_1} \circ \ldots \circ \Gamma_d^{\mathsf{p}_n}$ such that $\Gamma_{ds} = \Gamma_d^{\mathsf{p}_1}, X_{\mathsf{p}_1} : [\![\Gamma_d]\!]_{\mathsf{p}_1} \circ \ldots \circ \Gamma_d^{\mathsf{p}_n}, X_{\mathsf{p}_n} : [\![\Gamma_d]\!]_{\mathsf{p}_n}$ and $[\![\Gamma', X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!] = \Gamma'' \circ \Gamma_{ds}$.

Likewise, we apply the induction hypothesis on $[\![\Gamma_d, X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!] \vdash_{\mathsf{min}} C''$ and we can partition $[\![\Gamma_d, X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!]$ in a similar way wrt $[\![\Gamma', X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!]$.

Finally we can write the derivation that proves the case

$$\dfrac{\Gamma'' \vdash_{\mathsf{min}} C''' \qquad \dfrac{\Delta \qquad \dfrac{\vdots}{\Gamma_d/\Gamma_d^{\mathsf{p}_1} \vdash_{\mathsf{min}} \prod_{i=2}^{n} \mathsf{def}\ X_{\mathsf{p}_i} = \llbracket C'' \rrbracket_{\mathsf{p}_i}\ \ \mathsf{in}\ \ \llbracket C' \rrbracket_{\mathsf{p}_i}}\ \lfloor^{\text{Min}}|_{\text{PAR}}\rceil}{\Gamma_{ds} \vdash_{\mathsf{min}} \mathsf{def}\ X_{\mathsf{p}_1} = \llbracket C'' \rrbracket_{\mathsf{p}_1}\ \ \mathsf{in}\ \ \llbracket C' \rrbracket_{\mathsf{p}_1}\ \mid\ \prod_{i=2}^{n} \mathsf{def}\ X_{\mathsf{p}_i} = \llbracket C'' \rrbracket_{\mathsf{p}_i}\ \ \mathsf{in}\ \ \llbracket C' \rrbracket_{\mathsf{p}_i}}\ \lfloor^{\text{Min}}|_{\text{PAR}}\rceil}{\llbracket \Gamma \rrbracket \vdash_{\mathsf{min}} C_d \mid C'''}\ \lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

where

$$\Delta = \left\{ \dfrac{\Gamma_d^{\mathsf{p}_1}, X_{\mathsf{p}_1} : \llbracket \Gamma_d \rrbracket_{\mathsf{p}_1} \vdash_{\mathsf{min}} \llbracket C' \rrbracket_{\mathsf{p}_1} \qquad \llbracket \Gamma_d \rrbracket_{\mathsf{p}_1}, X_{\mathsf{p}_1} : \llbracket \Gamma_d \rrbracket_{\mathsf{p}_1} \vdash_{\mathsf{min}} \llbracket C'' \rrbracket_{\mathsf{p}_1} \qquad \llbracket \Gamma_d \rrbracket_{\mathsf{p}_1}\Big|_{\mathsf{locs}} \subseteq \Gamma_d^{\mathsf{p}_1}}{\Gamma_d^{\mathsf{p}_1}, X_{\mathsf{p}_1} : \llbracket \Gamma_d \rrbracket_{\mathsf{p}_1} \vdash_{\mathsf{min}} \mathsf{def}\ X_{\mathsf{p}_1} = \llbracket C'' \rrbracket_{\mathsf{p}_1}\ \ \mathsf{in}\ \ \llbracket C' \rrbracket_{\mathsf{p}_1}}\ \lfloor^{\text{Min}}|_{\text{DEF}}\rceil \right.$$

Case $\lfloor^{\text{Min}}|_{\text{CALL}}\rceil$
From the premises we know that $C = X\langle \tilde{\mathsf{r}} \rangle$, on which we can apply the typing derivation

$$\dfrac{\mathsf{end}(\Gamma) \qquad \Gamma' \subseteq \Gamma_d}{\Gamma, \Gamma', X\langle \tilde{\mathsf{r}} \rangle : \Gamma_d \vdash_{\mathsf{min}} X\langle \tilde{\mathsf{r}} \rangle}\ \lfloor^{\text{Min}}|_{\text{CALL}}\rceil$$

Since $\Gamma, \Gamma', X\langle \tilde{\mathsf{r}} \rangle : \Gamma_d \vdash_{\mathsf{min}} X\langle \tilde{\mathsf{r}} \rangle$ we know that

$$\llbracket X\langle \tilde{\mathsf{r}} \rangle \rrbracket = \prod_{\mathsf{p} \,\in\, \tilde{\mathsf{r}}} X_{\mathsf{p}}$$

Let $\Gamma_c = \Gamma', \Gamma'', X\langle \tilde{\mathsf{r}} \rangle : \Gamma_d$, following Lemma 19 we can partition $\llbracket \Gamma_c \rrbracket$ such that, let $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\} = \tilde{\mathsf{r}}$

$$\llbracket \Gamma_c \rrbracket = \llbracket \Gamma' \rrbracket\ , \llbracket \Gamma'' \rrbracket_{,}\ \llbracket X\langle \tilde{\mathsf{r}} \rangle : \Gamma_d \rrbracket$$

where

$$\llbracket X\langle \tilde{\mathsf{r}} \rangle : \Gamma_d \rrbracket = X_{\mathsf{p}_1} :\ \llbracket \Gamma_d \rrbracket_{\mathsf{p}_1} \circ \ldots \circ X_{\mathsf{p}_n} :\ \llbracket \Gamma_d \rrbracket_{\mathsf{p}_n}$$

$$\llbracket \Gamma' \rrbracket = \Gamma'_{\mathsf{p}_1} \circ \ldots \circ \Gamma'_{\mathsf{p}_n}$$

$$\llbracket \Gamma'' \rrbracket = \Gamma''_{\mathsf{p}_1} \circ \ldots \circ \Gamma''_{\mathsf{p}_n}$$

We can write the derivation that proves the case

$$\dfrac{\dfrac{\mathsf{end}(\Gamma'_{\mathsf{p}_1}) \quad \Gamma''_{\mathsf{p}_1} \subseteq [\![\Gamma_d]\!]_{\mathsf{p}_1}}{\Gamma'_{\mathsf{p}_1}, \Gamma''_{\mathsf{p}_1}, X_{\mathsf{p}_1} : [\![\Gamma_d]\!]_{\mathsf{p}_1} \vdash_{\mathsf{min}} X_{\mathsf{p}_1}} \lfloor^{\text{Min}}|_{\text{CALL}}\rceil \quad \Delta}{[\![\Gamma']\!], [\![\Gamma'']\!]\ [\![X\langle\tilde{\mathsf{r}}\rangle : \Gamma_d]\!] \vdash_{\mathsf{min}} [\![X\langle\tilde{\mathsf{r}}\rangle]\!]} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

$$\Delta = \dfrac{\vdots}{\Gamma'_{\mathsf{p}_2}, \Gamma''_{\mathsf{p}_2}, X_{\mathsf{p}_2} : [\![\Gamma_d]\!]_{\mathsf{p}_2} \circ \ldots \circ \Gamma'_{\mathsf{p}_n}, \Gamma''_{\mathsf{p}_n}, X_{\mathsf{p}_n} : [\![\Gamma_d]\!]_{\mathsf{p}_n} \vdash_{\mathsf{min}} \prod_{\mathsf{p}\in\{\tilde{\mathsf{p}}\}/\mathsf{p}_1} X_{\mathsf{p}}} \lfloor^{\text{Min}}|_{\text{PAR}}\rceil$$

as, for all $i \in \{1, \ldots, n\}$, $\mathsf{end}(\Gamma'_{\mathsf{p}_i})$ and $\Gamma''_{\mathsf{p}_i} \subseteq [\![\Gamma']\!]_{\mathsf{p}_i}$ hold by the premises and by construction.

□

We finally prove Theorem 6.
Theorem 6 (EPP Type Preservation) Let $\Gamma \vdash_{\mathsf{min}} D, C$ then $[\![\Gamma]\!] \vdash_{\mathsf{min}} D, [\![C]\!]$.

Proof. To prove Theorem 6 we need to prove that we can apply Rule $\lfloor^{\text{Min}}|_{\text{DC}}\rceil$ on $[\![\Gamma]\!] \vdash_{\mathsf{min}} D, [\![C]\!]$

$$\dfrac{\mathsf{pco}([\![\Gamma]\!]) \quad [\![\Gamma]\!] \vdash_{\mathsf{min}} D \quad [\![\Gamma]\!] \vdash_{\mathsf{min}} [\![C]\!]}{[\![\Gamma]\!] \vdash_{\mathsf{min}} D, [\![C]\!]} \lfloor^{\text{Min}}|_{\text{DC}}\rceil$$

where

- $\mathsf{pco}([\![\Gamma]\!])$ and $[\![\Gamma]\!] \vdash_{\mathsf{min}} D$ hold because $[\![\Gamma]\!]$ alters only procedure definitions in Γ;
- $[\![\Gamma]\!] \vdash_{\mathsf{min}} [\![C]\!]$ holds from Lemma 20 and the assumption of well-sortedness on C (if C is well-sorted also $[\![C]\!]$ is well-sorted and typable by Γ).

□

E.3.4 EPP Theorem

Before proving Theorem 7 we define some auxiliary concepts to establish a correspondence between a choreography and its projection.

Lemma 21 (EPP Swap Invariance)**.** *Let $C \simeq_{\mathsf{C}} C'$ then $[\![C]\!] \simeq_{\mathsf{C}} [\![C']\!]$.*

Proof Sketch. In the proof we show that the projection is invariant under the rules for the swapping relation $\simeq_{\mathsf{C}}$ defined in Figure D.1. $\lfloor^{\text{CS}}|_{\text{EtaEta}}\rceil$ is trivial. For Rule $\lfloor^{\text{CS}}|_{\text{EtaCnd}}\rceil$ we need to check that the projections of the processes in the swapped interaction η do not change, which holds by the definition of EPP for $(cond)$ terms and the merging operator (merging the same η returns η). The same reasoning on the EPP and the merging operator applies to all other cases. □

Lemma 22 (EPP under $\equiv$)**.** *Let $C \equiv C'$ then $[\![C]\!] \equiv [\![C']\!]$.*

Proof. Easy by cases on the rules of $\equiv$ for C. □

Lemma 23 (Compositional EPP)**.** *Let C be well-typed and $C = C_1 \mid C_2$ then $[\![C]\!] \equiv [\![C_1]\!] \mid [\![C_2]\!]$.*

Proof. By definition of EPP

$$[\![C]\!] = \prod_{\mathsf{p}\,\in\,\mathsf{fp}(C)} [\![C]\!]_{\mathsf{p}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s}\in\lfloor C\rfloor_l} [\![C]\!]_{\mathsf{s}} \right)$$

Since C is well-typed and $C = C_1 \mid C_2$, Rule $\lfloor^{\text{T}}|_{\text{Par}}\rceil$ applies and by definition of $\Gamma_1 \circ \Gamma_2$ there cannot be a process p such that $\mathsf{p} \in \mathsf{fp}(C_1) \cap \mathsf{fp}(C_2)$. Therefore we can write

$$[\![C]\!] \equiv \prod_{\mathsf{p}\,\in\,\mathsf{fp}(C_1)} [\![C_1]\!]_{\mathsf{p}} \mid \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C_2)} [\![C_2]\!]_{\mathsf{q}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s}\in\lfloor C\rfloor_l} [\![C]\!]_{\mathsf{s}} \right)$$

By the definition of service typing we know that *i*) locations can implement only one role in a choreography and *ii*) a location can appear only in one service typing. Therefore there cannot be two service processes at the same location in C_1 and C_2. Thus we can write

$$[\![C]\!] \equiv \underbrace{\prod_{\mathsf{p}\,\in\,\mathsf{fp}(C_1)} [\![C_1]\!]_{\mathsf{p}}}_{C_1^a} \mid \underbrace{\prod_{\mathsf{q}\,\in\,\mathsf{fp}(C_2)} [\![C_2]\!]_{\mathsf{q}}}_{C_2^a} \mid \underbrace{\prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C_1\rfloor_l} [\![C_1]\!]_{\mathsf{r}} \right)}_{C_1^s} \mid \underbrace{\prod_{l'} \left(\bigsqcup_{\mathsf{s}\,\in\,\lfloor C_2\rfloor_{l'}} [\![C_2]\!]_{\mathsf{s}} \right)}_{C_2^s}$$

where $[\![C_1]\!] = C_1^a \mid C_1^s$ and $[\![C_2]\!] = C_2^a \mid C_2^s$ by definition of EPP. □

Pruning

Following our definition of EPP, the projection of $(start)$ terms on service processes yield a parallel composition of (acc) terms on the locations subject of the

$(start)$. However, the reduction of a $(start)$ term might remove the availability to start new processes on the locations subject of the $(start)$ (i.e., if the reductum does not contain another $(start)$ term on the same locations). Contrarily, (acc) terms remain always available.

A similar observation can be drawn between conditional branches that contain (com) terms whose projection merges all possible communications into $(recv)$ and $(send)$ terms. Also in this case, reducing the condition and projecting the result yields a subset of all possible branches for the considered communication.

Similarly to [14] and [96], we deal with these asymmetries by introducing the *pruning relation*, which allows us to ignore unused *i*) endpoint services and *ii*) input branches.

Definition 37 (Pruning)**.** *Let* $\Gamma \vdash_{\mathsf{min}} [\![C]\!]$, $\Gamma = \Gamma_p, \Gamma_{acc}$ *where* $\Gamma_{acc}|_{\mathsf{locs}} = \Gamma_{acc}$ *and* $\Gamma_p|_{\mathsf{locs}} = \emptyset$ *and* $[\![C]\!] = C_p \mid C_{acc}$*,* $\Gamma_p \vdash_{\mathsf{min}} C_p$*,* $\Gamma_{acc} \vdash_{\mathsf{min}} C_{acc}$*.*
Let $C' = C'_p \mid C'_{acc}$ *such that* $\Gamma_p, \Gamma_{acc}, \Gamma'_{acc} \vdash_{\mathsf{min}} C'$ *where*

- $\Gamma_{acc}, \Gamma'_{acc} \vdash_{\mathsf{min}} C'_{acc}$*,* $\Gamma'_{acc}|_{\mathsf{locs}} = \Gamma'_{acc}$ *and* $\Gamma'_{acc} \cap \Gamma_{acc} = \emptyset$*,*
- $\Gamma_p \vdash_{\mathsf{min}} C'_p$ *and* $C'_p \sqcup C_p = C'_p$

then $[\![C]\!]$ *prunes* C' *under* Γ*, written* $\Gamma \vdash_{\mathsf{min}} [\![C]\!] \prec C'$*.*

Like in [96] we can write $[\![C]\!] \prec C'$ as its does not lose any precision since it is always possible to reconstruct appropriate typings.

The $\prec$ relation is a typed strong bisimulation because if $[\![C]\!] \prec C'$ then $[\![C]\!]$ and C' have the same observable behaviours except for the visible input actions at pruned inputs (either input branches or services).

Lemma 24 (Pruning Lemma)**.** $\prec$ *is a strong reduction bisimilation over partial choreographies if* $[\![C]\!] \prec C'$ *and for some* D*,* D'

- *if* $D, [\![C]\!] \to D', C''$ *then* $D, C' \to D', C'''$ *and* $C'' \prec C'''$ *and*
- *if* $D, C' \to D, C'''$ *then* $D, [\![C]\!] \to D, C''$ *and* $C'' \prec C'''$

Proof Sketch. Since $[\![C]\!] \prec C'$ we know that C' does not use any of the locations not typed by the minimal typing of $[\![C]\!]$. Similarly, the input branches pruned in $[\![C]\!]$ are only those that are never used in C' and cannot contribute in the reduction. □

Before continuing with the last auxiliary results and the proof of Theorem 7 we need to augment the labels of the semantics of annotated ACs with the identifiers of the processes involved in a reduction

$$\beta ::= k : \mathsf{p}[\mathtt{A}] \mathbin{\texttt{->}} \mathtt{B}.o \mid \mathtt{A} \rightsquigarrow \mathsf{q}[\mathtt{B}].o(x) \mid \tau @ \mathsf{p} \mid \tau$$

and the annotation of the reduction with Rule $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$ as

$$\frac{i = 1 \text{ if } \mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{ true}, i = 2 \text{ otherwise}}{D,\ \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \quad \xrightarrow{\tau@\mathsf{p}} \quad D,\ C_i} \lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$$

Let also $\mathsf{pn}(k : \mathsf{p}[\mathtt{A}] \texttt{->} \mathtt{B}.o) = \{\mathsf{p}\}$, $\mathsf{pn}(\mathtt{A} \rightsquigarrow \mathsf{q}[\mathtt{B}].o(x)) = \{\mathsf{q}\}$, $\mathsf{pn}(\tau@\mathsf{p}) = \{\mathsf{p}\}$, and $\mathsf{pn}(\tau) = \emptyset$

Lemma 25 (Passive Processes Pruning Invariance). *$D, C \xrightarrow{\beta} D', C'$ implies that for all $\mathsf{p} \in \mathsf{fp}(C)/\mathsf{pn}(\beta)$, $[\![C']\!]_{\mathsf{p}} \prec [\![C]\!]_{\mathsf{p}}$.*

Sketch Proof. By cases on the derivation of C. The only interesting case is $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$ in which the the projection of the processes receiving selections are merged. The thesis follows directly from Definition 37 and Lemmas 21 and 22. □

Lemma 26 (Deployment Invariance). *Let D and D' be two deployments such that*

1. $\forall\, \mathsf{p}.x : U \in \Gamma,\ x(D(\mathsf{p}).\mathsf{st}) = x(D'(\mathsf{p}).\mathsf{st})$
2. $\forall\, \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \in \Gamma,\ D(\mathsf{p}).\mathsf{que}(\underline{\mathbf{k.A.B}}(D(\mathsf{p}).\mathsf{st})) = D'(\mathsf{p}).\mathsf{que}(\underline{\mathbf{k.A.B}}(D'(\mathsf{p}).\mathsf{st}))$
3. *$\Gamma \vdash_{\mathsf{min}} D, C$ and $\Gamma \vdash_{\mathsf{min}} D', C$ for some C.*

then $D, C \sim D', C$.

Sketch Proof. By cases on the derivation of C. Reductions on C are invariant wrt to a specific deployment as long as it is well-typed (3.) and for each process: typed variables have the same value (1.) and queues hold the same messages (2.). □

E.3.5 Proof of Theorem 7

We restate Theorem 7 to include annotated reductions.

Theorem 7 (EPP Operational Correspondence)
Let D, C be well-typed and well-annotated. Then,

1. *(Completeness) $D, C \xrightarrow{\beta} D', C'$ implies $D, [\![C]\!] \xrightarrow{\beta} D', C''$ and $[\![C']\!] \prec C''$.*
2. *(Soundness) $D, [\![C]\!] \xrightarrow{\beta} D', C''$ implies $D, C \xrightarrow{\beta} D', C'$ and $[\![C']\!] \prec C''$.*

We report below the respective proofs of *(Completeness)* and *(Soundness)* separately.

Proof (Completeness).

Proof by induction on the derivation of $D, C \xrightarrow{\beta} D', C'$.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$
we know that $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C_c$ and we can write the derivation

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x) \quad D, k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o \blacktriangleright D'}{D,\ \eta; C \quad \xrightarrow{k:\ \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathtt{B}.o} \quad D',\ k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C_c} \lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$$

and $C' = k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C_c$.
From the definition of EPP we have that $[\![C]\!] = C_{act} \mid C_s$ such that

$$C_{act} = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o; [\![C_c]\!]_{\mathsf{p}} \mid k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); [\![C_c]\!]_{\mathsf{q}} \mid \prod_{\mathsf{r}\, \in\, \mathsf{fp}(C)/\{\mathsf{p},\mathsf{q}\}} [\![C_c]\!]_{\mathsf{r}}$$

and

$$C_s = \prod_{l} \left(\bigsqcup_{\mathsf{s}\, \in\, \lfloor C \rfloor_l} [\![C_c]\!]_{\mathsf{s}} \right)$$

Whilst $[\![C']\!] \equiv C'_{act} \mid C_s$

$$C'_{act} = [\![C_c]\!]_{\mathsf{p}} \mid k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); [\![C_c]\!]_{\mathsf{q}} \mid \prod_{\mathsf{r}\, \in\, \mathsf{fp}(C')/\{\mathsf{p},\mathsf{q}\}} [\![C_c]\!]_{\mathsf{r}}$$

We can apply Rules $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, and $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ on $D, [\![C]\!]$ such that

$$\frac{\eta = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o \quad D, \eta \blacktriangleright D''}{D,\ [\![C]\!] \quad \xrightarrow{k:\ \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathtt{B}.o} \quad D'',\ C''} \lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$$

for which it holds that $D' = D''$ by Rule $\lfloor^{\mathrm{D}}|_{\mathrm{SEND}}\rceil$.

$$C'' = [\![C_c]\!]_{\mathsf{p}} \mid k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); [\![C_c]\!]_{\mathsf{q}} \mid \prod_{\mathsf{r}\, \in\, \mathsf{fp}(C')/\{\mathsf{p},\mathsf{q}\}} [\![C_c]\!]_{\mathsf{r}} \mid C_s$$

for which it holds that $[\![C']\!] \prec C''$.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$

Similar to case $\lfloor^{\mathrm{C}}|_{\mathrm{COM}}\rceil$.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$

we know that $C = D,\ k : \mathtt{A} \rightarrow \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i\in I}$ and we can write the derivation

$$\dfrac{j \in I \quad D, k : \mathtt{A} \rightarrow \mathsf{q}[\mathtt{B}].o_j(x_j) \blacktriangleright D'}{D,\ k : \mathtt{A} \rightarrow \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i\in I} \quad \xrightarrow{k:\mathtt{A} \rightsquigarrow \mathtt{B}.o_j(x_j)} \quad D',\ C_j}\ \lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$$

for $\beta = k : \mathtt{A} \rightsquigarrow \mathtt{B}.o_j(x_j)$ and $C' = C_j$.
By the definition of EPP we have

$$[\![C]\!] \equiv k : \mathtt{A} \rightarrow \mathsf{q}[\mathtt{B}].\{o_i(x_i); [\![C_i]\!]_{\mathsf{q}}\}_{i\in I} \mid \prod_{\mathsf{p}\,\in\,\mathsf{fp}(C)/\{\mathsf{q}\}} \left(\bigsqcup_{i\,\in\,I} [\![C_i]\!]_{\mathsf{p}} \right) \mid \prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

Then we can apply Rules $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, and $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$ such that

$$\dfrac{j \in I \quad D, k : \mathtt{A} \rightarrow \mathsf{q}[\mathtt{B}].o_j(x_j) \blacktriangleright D''}{D,\ [\![C]\!] \quad \xrightarrow{k:\mathtt{A} \rightsquigarrow \mathtt{B}.o_j(x_j)} \quad D'',\ [\![C_j]\!]_{\mathsf{q}} \mid \prod_{\mathsf{p}\,\in\,\mathsf{fp}(C)/\{\mathsf{q}\}} \left(\bigsqcup_{i\,\in\,I} [\![C_i]\!]_{\mathsf{p}} \right) \mid \prod_l \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)}\ \lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$$

and

$$C'' = [\![C_j]\!]_{\mathsf{q}} \mid \prod_{\mathsf{p}\,\in\,\mathsf{fp}(C)/\{\mathsf{q}\}} \left(\bigsqcup_{i\,\in\,I} [\![C_i]\!]_{\mathsf{p}} \right) \mid \prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

From Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ we know that $D'' = D'$.
$[\![C']\!] \prec C''$ by Definition 37 and Lemma 25.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{START}}\rceil$

we know that $C = \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C_c$ and we can write the derivation

$$\dfrac{\#\tilde{\mathsf{r}} \quad \#k' \quad \mathsf{p} \in D(l) \quad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l.\mathsf{r}[\mathtt{B}]} \quad D, \delta \blacktriangleright D'}{D,\ \textbf{start}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C_c \quad \xrightarrow{\tau} \quad D',\ C_c[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]}\ \lfloor^{\mathrm{C}}|_{\mathrm{START}}\rceil$$

and $C' = C_c[k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}]$.

From the definition of EPP we have

$$[\![C']\!] = \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C')} [\![C']\!]_{\mathsf{q}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s}\,\in\,\lfloor C' \rfloor_l} [\![C']\!]_{\mathsf{s}} \right)$$

and

$$[\![C]\!] \equiv \begin{array}{l} \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; [\![C_c]\!]_{\mathsf{p}} \mid \prod_{\mathsf{r}\,\in\,\mathsf{fp}(C)/\{\mathsf{p}\}} [\![C]\!]_{\mathsf{r}} \\ \mid \prod_{l.\mathsf{q}[\mathtt{B}]\,\in\,\widetilde{l.\mathsf{q}[\mathtt{B}]}} \textbf{acc}\ k : l.\mathsf{q}[\mathtt{B}]; [\![C_c]\!]_{\mathsf{q}} \mid \prod_{l' \notin \tilde{l}} \left(\prod_{\mathsf{s}\,\in\,\lfloor C \rfloor_{l'}} [\![C]\!]_{\mathsf{s}} \right) \end{array}$$

we can apply Rules $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$ such that

$$\frac{\begin{array}{c} i \in \{1, \dots, n\} \quad \#k'' \quad \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{\widetilde{l_i.\mathtt{B}_i}\} \\ \#\tilde{\mathsf{r}}' \quad \{\tilde{\mathsf{r}}'\} = \bigcup_i \{\tilde{\mathsf{r}}'_i\} \quad \mathsf{p} \in D(l) \\ \delta = \textbf{start}\ k'' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l_1.\mathsf{r}'_1}[\mathtt{B}_1], \dots, \widetilde{l_n.\mathsf{r}'_n}[\mathtt{B}_n] \qquad D, \delta \blacktriangleright D'' \end{array}}{D, [\![C]\!] \quad \xrightarrow{\tau} \quad D'', C''} \ \lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$$

where

$$C'' \equiv \left\{ \begin{array}{l} [\![C_c]\!]_{\mathsf{p}}[k''/k] \quad \mid \prod_{(\mathsf{q},\mathsf{r}')\,\in\,\left\{(\mathsf{q}_1,\mathsf{r}'_1),\dots,(\mathsf{q}_n,\mathsf{r}'_n)\right\}} [\![C_c]\!]_{\mathsf{q}}[k''/k][\mathsf{q}/\mathsf{r}'] \\ \quad \mid \prod_{\mathsf{r}\,\in\,\mathsf{fp}(C_c)/\{\mathsf{p},\tilde{\mathsf{q}}\}} [\![C_c]\!]_{\mathsf{r}} \quad \mid \prod_{l.\mathsf{q}[\mathtt{B}]\,\in\,\widetilde{l.\mathsf{q}[\mathtt{B}]}} \textbf{acc}\ k : l.\mathsf{q}[\mathtt{B}]; [\![C_c]\!]_{\mathsf{q}} \\ \quad \mid \prod_{l' \notin \tilde{l}} \left(\prod_{\mathsf{s}\,\in\,\lfloor C_c \rfloor_{l'}} [\![C_c]\!]_{\mathsf{s}} \right) \end{array} \right.$$

Observe that we can α-rename k'' to k' and $\tilde{\mathsf{r}}'$ to $\tilde{\mathsf{r}}$ as k'', k', $\tilde{\mathsf{r}}'$, and $\tilde{\mathsf{r}}$ are all fresh wrt D, C.

From the application of Rule $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$ we can find Γ such that

$$\Gamma \vdash_{\mathsf{min}} (\ D'', C''\ )[k'/k''][\tilde{\mathsf{r}}/\tilde{\mathsf{r}}']$$

and

$$\Gamma \vdash_{\mathsf{min}} D', C''[k'/k''][\tilde{\mathsf{r}}/\tilde{\mathsf{r}}']$$

and by Lemma 26 we have that

$$D, [\![C]\!] \xrightarrow{\tau} D', C''[k'/k''][\tilde{\mathsf{r}}/\tilde{\mathsf{r}}']$$

Finally $[\![C']\!] \prec C''[k'/k''][\tilde{\mathsf{r}}/\tilde{\mathsf{r}}']$ by Lemma 25.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$

Similar to (in particular the second part of) the proof of case $\lfloor^{\mathrm{C}}|_{\mathrm{START}}\rceil$.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$

we know that $C \equiv \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}$ and we can write the derivation

$$\frac{i = 1 \text{ if } \mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{ true}, i = 2 \text{ otherwise}}{D,\ \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \quad \xrightarrow{\tau @ \mathsf{p}} \quad D,\ C_i}\ \lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$$

We only consider the case for $\mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{true}$ as $\mathsf{eval}(e, D(\mathsf{p}).\mathsf{st}) = \text{false}$ follows alike.

$C' = C_1$ and by the definition of EPP

$$[\![C]\!] \equiv \mathsf{if}\ \mathsf{p}.e\ \{[\![C_1]\!]_{\mathsf{p}}\}\ \mathsf{else}\ \{[\![C_2]\!]_{\mathsf{p}}\} \mid \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C')/\{\mathsf{p}\}} [\![C_1]\!]_{\mathsf{q}} \sqcup [\![C_2]\!]_{\mathsf{q}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

and

$$[\![C']\!] \equiv [\![C_1]\!]_{\mathsf{p}} \mid \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C')/\{\mathsf{p}\}} [\![C_1]\!]_{\mathsf{q}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C_1 \rfloor_l} [\![C_1]\!]_{\mathsf{r}} \right)$$

We can apply rules $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, and $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$ such that $D, [\![C]\!] \xrightarrow{\tau @ \mathsf{p}} D, C''$ where

$$C'' = [\![C_1]\!]_{\mathsf{p}} \mid \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C')/\{\mathsf{p}\}} [\![C_1]\!]_{\mathsf{q}} \sqcup [\![C_2]\!]_{\mathsf{q}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

and $[\![C']\!] \prec C''$ by Lemma 25.

Case $\lfloor^{\mathsf{C}}|_{\text{CTX}}\rceil$ and **Case** $\lfloor^{\mathsf{C}}|_{\text{PAR}}\rceil$

Proven by the definition of EPP and the induction hypothesis.

Case $\lfloor^{\mathsf{C}}|_{\text{EQ}}\rceil$

We can write the derivation

$$\frac{\mathcal{R} \in \{\equiv, \simeq_{\mathsf{C}}\} \quad C_1 \,\mathcal{R}\, C_1' \quad D, C_1' \xrightarrow{\beta} D', C_2' \quad C_2' \,\mathcal{R}\, C_2}{D, C_1 \quad \xrightarrow{\beta} \quad D', C_2} \lfloor^{\mathsf{C}}|_{\text{EQ}}\rceil$$

For $\mathcal{R} \;= \equiv$, proven by the definition of EPP and the induction hypothesis.

For $\mathcal{R} \;= \simeq_{\mathsf{C}}$, proven by the definition of EPP, Lemma 21, and the induction hypothesis.

$\square$

Proof (Soundness). Proof by induction on the structure of C.

Case $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C_c$
From the definition of EPP we have

$$[\![C]\!] \equiv k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o; [\![C_c]\!]_{\mathsf{p}} \mid k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); [\![C_c]\!]_{\mathsf{q}} \mid \prod_{\mathsf{r} \in \mathsf{fp}(C)} [\![C_c]\!]_{\mathsf{r}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s} \in \lfloor C \rfloor_l} [\![C]\!]_{\mathsf{s}} \right)$$

we proceed by subcases on the last applied Rule in the derivation of $D, [\![C]\!] \xrightarrow{\beta} D', C''$.

Case $\lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$

Divided into subcases whether $\beta = k : \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathtt{B}.o$ holds or not.

Case $\beta = k : \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathtt{B}.o$
$D, [\![C]\!]$ reduces to D', C'' with Rules $\lfloor^{\mathsf{C}}|_{\text{PAR}}\rceil$, $\lfloor^{\mathsf{C}}|_{\text{EQ}}\rceil$, ending with Rule $\lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$ such that

$$C'' = [\![C_c]\!]_{\mathsf{p}} \mid k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); [\![C_c]\!]_{\mathsf{q}} \mid \prod_{\mathsf{r} \in \mathsf{fp}(C)} [\![C_c]\!]_{\mathsf{r}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s} \in \lfloor C \rfloor_l} [\![C]\!]_{\mathsf{s}} \right)$$

D, C mimics $D, [\![C]\!]$ with Rule $\lfloor^{\text{C}}|_{\text{COM}}\rceil$ for which $D, C \xrightarrow{\beta} D'', C'$, $D' = D''$ by Rule $\lfloor^{\text{D}}|_{\text{SEND}}\rceil$,

$$[\![C']\!] \equiv [\![C_c]\!]_{\mathsf{p}} \mid k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); [\![C_c]\!]_{\mathsf{q}} \mid \prod_{\mathsf{r} \in \mathsf{fp}(C)} [\![C_c]\!]_{\mathsf{r}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s} \in \lfloor C \rfloor_l} [\![C]\!]_{\mathsf{s}} \right)$$

and $[\![C']\!] \prec C''$ by Lemma 25.

Case $\beta \neq k : \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathtt{B}.o$

In this case D, C can mimic $D, [\![C]\!]$ with the application of Rules $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$ and $\lfloor^{\text{C}}|_{\text{PAR}}\rceil$, ending the derivation with either Rule $\lfloor^{\text{C}}|_{\text{COM}}\rceil$ or $\lfloor^{\text{C}}|_{\text{SEND}}\rceil$. The thesis follows by the induction hypothesis.

Case $\lfloor^{\text{C}}|_{\text{RECV}}\rceil$, $\lfloor^{\text{C}}|_{\text{PSTART}}\rceil$, or $\lfloor^{\text{C}}|_{\text{COND}}\rceil$

In this case $D, [\![C]\!]$ reduces with Rules $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$, $\lfloor^{\text{C}}|_{\text{PAR}}\rceil$, and respectively ends the derivation with either $\lfloor^{\text{C}}|_{\text{RECV}}\rceil$, $\lfloor^{\text{C}}|_{\text{PSTART}}\rceil$, or $\lfloor^{\text{C}}|_{\text{COND}}\rceil$, i.e., some process $\mathsf{r} \in \mathsf{fp}(C)$ (p and q included) either receives a message, starts a new session with some service processes, or reduces to some branch. D, C can mimic $D, [\![C]\!]$ applying Rules $\lfloor^{\text{C}}|_{\text{EQ}}\rceil$, $\lfloor^{\text{C}}|_{\text{PAR}}\rceil$ and terminates the derivation with either Rules $\lfloor^{\text{C}}|_{\text{RECV}}\rceil$, $\lfloor^{\text{C}}|_{\text{PSTART}}\rceil$, (or $\lfloor^{\text{C}}|_{\text{START}}\rceil$, depending on the form of C) or $\lfloor^{\text{C}}|_{\text{COND}}\rceil$. The thesis follows by the induction hypothesis.

Case $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o; C_c$

Similar to case $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C_c$.

Case $C = k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I}$

From the definition of EPP we have

$$[\![C]\!] \equiv k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); [\![C_i]\!]_{\mathsf{q}}\}_{i \in I} \mid \prod_{i \in I} \left(\bigsqcup_{\mathsf{p} \in \mathsf{fp}(C_i)} [\![C_i]\!]_{\mathsf{p}} \right) \mid \prod_{k} \left(\bigsqcup_{\mathsf{r} \in \lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

we proceed by subcases on last applied Rule in the derivation of $D, [\![C]\!] \xrightarrow{\beta} D', C''$.

Case $\lfloor^{\text{C}}|_{\text{RECV}}\rceil$

Divided into subcases whether $\beta = k \colon \mathtt{A} \rightsquigarrow \mathsf{q}[\mathtt{B}].o_j$, $j \in I$ or not.

Case $\beta = k\colon \mathtt{A} \rightsquigarrow \mathsf{q}[\mathtt{B}].o_j, j \in I$
$D, [\![C]\!]$ reduces to D', C'' with Rules $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, and terminates with Rule $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ such that

$$C'' = [\![C_j]\!]_{\mathsf{q}} \mid \prod_{i \in I} \left(\bigsqcup_{\mathsf{p} \in \mathsf{fp}(C_i)} [\![C_i]\!]_{\mathsf{p}} \right) \mid \prod_{k} \left(\bigsqcup_{\mathsf{r} \in \lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

D, C mimics $D, [\![C]\!]$ with Rule $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ for which $D, C \xrightarrow{\beta} D'', C'$ where $D'' = D'$ by Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ and

$$C'' = [\![C_j]\!]_{\mathsf{q}} \mid \prod_{\mathsf{p} \in \mathsf{fp}(C_j)} [\![C_j]\!]_{\mathsf{p}} \mid \prod_{k} \left(\bigsqcup_{\mathsf{r} \in \lfloor C_j \rfloor_l} [\![C_j]\!]_{\mathsf{r}} \right)$$

and $[\![C']\!] \prec C''$ by Lemma 25.
Case $\beta \neq k\colon \mathtt{A} \rightsquigarrow \mathsf{q}[\mathtt{B}].o_j$
For any β of this case D, C can mimic $D, [\![C]\!]$ with the application of Rules $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$ and $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, terminating with Rule $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ and the thesis follows by the induction hypothesis.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{SEND}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$, or $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$
is similar to subcase **Case** $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$, or $\lfloor^{\mathrm{C}}|_{\mathrm{COND}}\rceil$ of **Case** $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].o(x); C_c$.

Case $C =$ **start** $k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathsf{q}[\mathtt{B}]}; C_c$

$$[\![C]\!] \equiv \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C_c \mid \prod_{\mathsf{r} \in \mathsf{fp}(C_c)} [\![C_c]\!]_{\mathsf{r}} \mid \prod_{l} \left(\bigsqcup_{\mathsf{s} \in \lfloor C \rfloor_l} [\![C]\!]_{\mathsf{s}} \right)$$

we proceed by subcases on last applied Rule in the derivation of $D, [\![C]\!] \xrightarrow{\beta} D, C''$.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$
$D, [\![C]\!]$ can reduce to D', C'' with a process r (including p) that starts a new session with some service processes. D, C can reduce to D'', C' mimicking $D, [\![C]\!]$ by applying Rules $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, terminating with either Rule $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$ or $\lfloor^{\mathrm{C}}|_{\mathrm{START}}\rceil$.
The proof follows from Lemma 26 as D'' and D' differ only on the correlation values used in for session k.

Case $\lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$, $\lfloor^{\mathsf{C}}|_{\text{RECV}}\rceil$, and $\lfloor^{\mathsf{C}}|_{\text{COND}}\rceil$
are similar to the corresponding proof for the previous cases.

Case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}$

From the definition of EPP we have

$$[\![C]\!] \equiv \mathsf{if}\ \mathsf{p}.e\ \{[\![C_1]\!]_{\mathsf{p}}\}\ \mathsf{else}\ \{[\![C_2]\!]_{\mathsf{p}}\} \mid \prod_{\mathsf{q}\,\in\,\mathsf{fp}(C_1)\,\cup\,\mathsf{fp}(C_2)/\{\mathsf{p}\}} [\![C_1]\!]_{\mathsf{q}} \sqcup [\![C_2]\!]_{\mathsf{q}} \mid \mid \prod_{l} \left(\bigsqcup_{\mathsf{r}\,\in\,\lfloor C \rfloor_l} [\![C]\!]_{\mathsf{r}} \right)$$

we proceed by subcases on the derivation of $D, [\![C]\!] \xrightarrow{\beta} D', C''$.

Case $\lfloor^{\mathsf{C}}|_{\text{COND}}\rceil$
$D, [\![C]\!]$ can reduce to D', C'' with:

Case $\beta = \tau@\mathsf{p}$
that reduces to a branch. D, C can mimic $D, [\![C]\!]$ applying Rules $\lfloor^{\mathsf{C}}|_{\text{EQ}}\rceil$, $\lfloor^{\mathsf{C}}|_{\text{PAR}}\rceil$, and terminating the derivation with Rule $\lfloor^{\mathsf{C}}|_{\text{COND}}\rceil$. The case is proven by Lemma 25.

Case $\beta = \tau@\mathsf{r}$
for $\mathsf{r} \neq \mathsf{p}$. A process r reduced to a branch. The case follows the proof of the previous case where the thesis follows by the induction hypothesis.

Case $\lfloor^{\mathsf{C}}|_{\text{RECV}}\rceil$, $\lfloor^{\mathsf{C}}|_{\text{SEND}}\rceil$, $\lfloor^{\mathsf{C}}|_{\text{PSTART}}\rceil$
are similar to the corresponding proof for the previous cases.

Case $C = \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C_c$
Case not allowed by the hypothesis that $D, [\![C]\!] \xrightarrow{\beta} D, C''$.

Case $C = \textbf{acc}\ k : \widetilde{l.\mathsf{q}[\mathtt{B}]}; C_c$
Case not allowed by the hypothesis that $D, [\![C]\!] \xrightarrow{\beta} D, C''$.

Case $C = \mathsf{def}\ X\langle\tilde{\mathsf{p}}\rangle = C_d\ \mathsf{in}\ C_c$
Proven by Lemma 22 and the induction hypothesis.

Case $C = X\langle\tilde{\mathsf{p}}\rangle$
Case not allowed by the hypothesis that C is well-sorted.

Case $C = C_1 \mid C_2$

$[\![C]\!] \equiv [\![C_1]\!] \mid [\![C_2]\!]$ by Lemma 23.

we proceed by subcases for n equal to the length of the derivation of $D, [\![C]\!] \xrightarrow{\beta} D', C''$

Case $n = 1$
In this case the only applicable Rule is $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$ where, by the definition of EPP, we can infer, let

$$\widetilde{l.\mathsf{q}[\mathtt{B}]} = l_1.\mathsf{q}_1[\mathtt{B}_1], \ldots, l_i.\mathsf{q}_i[\mathtt{B}_i], l_{i+1}.\mathsf{q}_{i+1}[\mathtt{B}_{i+1}] \ldots, l_n.\mathsf{q}_n[\mathtt{B}_n]$$

that

$$[\![C_1]\!] \equiv \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}};\ [\![C_1^r]\!]_{\mathsf{p}} \mid \prod_{j=1}^{i} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j];\ [\![C_1^j]\!]_{\mathsf{q}_j}$$

$$[\![C_2]\!] \equiv \prod_{j=i+1}^{n} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j];\ [\![C_2^j]\!]_{\mathsf{q}_j}$$

Observe that we can proceed without loss of generality as the symmetric case (with $\mathsf{p} \in \mathsf{fp}(C_2)$) follows the same structure.

$$\frac{\begin{array}{c} i \in \{1, \ldots, n\} \quad \#k' \quad \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{\widetilde{l_i.\mathtt{B}_i}\} \\ \#\tilde{\mathsf{r}} \quad \{\tilde{\mathsf{r}}\} = \bigcup_i \{\tilde{\mathsf{r}}_i\} \quad \mathsf{p} \in D(l) \\ \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], \widetilde{l_1.\mathsf{r}_1[\mathtt{B}_1]}, \ldots, \widetilde{l_n.\mathsf{r}_n[\mathtt{B}_n]} \qquad D, \delta \blacktriangleright D'' \end{array}}{D, [\![C_1]\!] \mid [\![C_2]\!] \quad \xrightarrow{\tau} \quad D', C''} \lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$$

where

$$C'' \equiv \begin{array}{l} [\![C_1^r]\!]_{\mathsf{p}}[k'/k] \mid \left(\begin{array}{c} \prod_{j=1}^{i} [\![C_1^j]\!]_{\mathsf{q}_j} \\ \mid \prod_{j=i+1}^{n} [\![C_2^j]\!]_{\mathsf{q}_j} \end{array} \right) [k'/k][\tilde{\mathsf{r}}/\tilde{\mathsf{q}}] \\ \mid \left(\begin{array}{c} \prod_{j=1}^{i} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j];\ [\![C_1^j]\!]_{\mathsf{q}_j} \\ \mid \prod_{j=i+1}^{n} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j];\ [\![C_2^j]\!]_{\mathsf{q}_j} \end{array} \right) \end{array}$$

Since both $[\![C_1]\!]$ and $[\![C_2]\!]$ reduce, we can infer from the definition of EPP that,

$$C_1 \equiv \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C_1^r \mid \prod_{j=1}^{i} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j]; C_1^j$$

$$C_2 \equiv \prod_{j=i+1}^{n} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j]; C_2^j$$

Then D, C can mimic $D, [\![C]\!]$ applying Rule $\lfloor^{\mathrm{C}}|_{\mathrm{PS\scriptstyle TART}}\rceil$ with reduction

$$\frac{\begin{array}{c} i \in \{1,\ldots,n\} \qquad \#k'' \qquad \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{\widetilde{l_i.\mathtt{B}_i}\} \qquad \#\tilde{\mathsf{r}'} \qquad \{\tilde{\mathsf{r}'}\} = \bigcup_i \{\tilde{\mathsf{r}}'_i\} \\ \mathsf{p} \in D(l) \qquad \delta = \textbf{start}\ k'' : \ l.\mathsf{p}[\mathtt{A}], \widetilde{l_1.\mathsf{r}'_1}[\mathtt{B}_1], \ldots, \widetilde{l_n.\mathsf{r}'_n}[\mathtt{B}_n] \qquad D, \delta \blacktriangleright D'' \end{array}}{D, [\![C_1]\!] \ | \ [\![C_2]\!] \quad \xrightarrow{\tau} \quad D'', C'} \ \lfloor^{\mathrm{C}}|_{\mathrm{PS\scriptstyle TART}}\rceil$$

where

$$C' \equiv C_1^r[k''/k] \,|\, \left(\begin{array}{c} \prod_{j=1}^{i} C_1^j \ | \\ \prod_{j=i+1}^{n} C_2^j \end{array} \right) [k''/k][\tilde{\mathsf{r}}'/\tilde{\mathsf{q}}] \,|\, \left(\begin{array}{c} \prod_{j=1}^{i} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j]; C_1^j \\ | \ \prod_{j=i+1}^{n} \textbf{acc}\ k : l_j.\mathsf{q}_j[\mathtt{B}_j]; C_2^j \end{array} \right)$$

Following the structure of the second part of the proof of **Case** $\lfloor^{\mathrm{C}}|_{\mathrm{S\scriptstyle TART}}\rceil$ for the proof of *Completeness* of Theorem 7, we have, by Lemma 26, $D'', C' \sim D', C'[k'/k''][\tilde{\mathsf{r}}/\tilde{\mathsf{r}}']$ and $[\![C']\!] \prec C''$.

Case $n > 1$

For $n > 1$ we have a derivation similar to

$$\frac{\begin{array}{c} R \\ \vdots \ \ \begin{array}{l} n-1 \text{ times, each either} \\ \lfloor^{\mathrm{C}}|_{\mathrm{P\scriptstyle AR}}\rceil \text{ or } \lfloor^{\mathrm{C}}|_{\mathrm{E\scriptstyle Q}}\rceil \end{array} \end{array}}{D, [\![C_1]\!] \ | \ [\![C_2]\!] \quad \xrightarrow{\beta} \quad D', C_1'' \ | \ [\![C_2]\!]} \ \lfloor^{\mathrm{C}}|_{\mathrm{P\scriptstyle AR}}\rceil$$

where R is the last applied Rule, $R \in \{\lfloor^{\mathrm{C}}|_{\mathrm{S\scriptstyle END}}\rceil, \lfloor^{\mathrm{C}}|_{\mathrm{C\scriptstyle OM}}\rceil, \lfloor^{\mathrm{C}}|_{\mathrm{S\scriptstyle END}}\rceil, \lfloor^{\mathrm{C}}|_{\mathrm{PS\scriptstyle TART}}\rceil, \lfloor^{\mathrm{C}}|_{\mathrm{C\scriptstyle OND}}\rceil\}$. The thesis follows from the induction hypothesis.

The proof for the mirror case $D, [\![C_1]\!] \ | \ [\![C_2]\!] \quad \xrightarrow{\beta} \quad D', [\![C_1]\!] \ | \ C_2''$ follows the same structure.

Case $C = \mathbf{0}$
trivial.

$\square$

E.4 Proof of Compilation

E.4.1 Proof of Theorem 8

Lemma 27 (Compilation Invariance)**.** *Let* $\Gamma \vdash D, C$*,* $D, C \to D', C'$*, and* $\Gamma' \vdash D', C'$ *for some* Γ'*,* $\Gamma|_{\mathsf{locs}} = \Gamma'|_{\mathsf{locs}}$ *and* $\{\mathsf{p}@l \mid \mathsf{p}@l \in \Gamma\} = \{\mathsf{p}@l \mid \mathsf{p}@l \in \Gamma'\}$*, then* $\boxed{D', C'}^{\Gamma} = \boxed{D', C'}^{\Gamma'}$*.*

Proof Sketch. By cases on the derivation of $D, C \to D', C'$. From Definition 23, the only information used by the compiler present in Γ (respectively Γ') are *i*) the location of running processes ($\mathsf{p}@l$) and *ii*) the service typings ($\tilde{l}\colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$). Therefore, provided Γ and Γ' have the same service typings and locations the compilation of the given choreography under Γ produces the same result as the compilation of such choreography under Γ'. $\square$

We provide some auxiliary results on variable substitution. We remind that the only bound names in DCC are the variables in $(accept)$ terms (e.g., x in $!(x); B$). However, the following lemmas prove that renaming free variables with fresh names in processes (and, by extension, in services) preservers bisimilarity.

In the following, we abuse the notation for α-renaming to denote variable renaming in running processes. We define the variable renaming operator for DCC processes $P[x'/x]$.

Definition 38 (DCC Variable Renaming Operator)**.** *Let* $B \cdot t \cdot M$ *be a DCC process, then* $(B \cdot t \cdot M)[x'/x] = B[x'/x] \cdot t \lhd (\, x', x(t)\,) \lhd (\, x, t_\perp\,) \cdot M$ *where* $B[x'/x]$ *substitutes every occurrence of* x *with* x'*.*

Lemma 28 (DCC Process Variable Renaming)**.** *Let* P *and* P' *be two DCC processes such that* $P = B \cdot t \cdot M$ *and* $P' = P[x'/x]$ *and* x' *is fresh in* B *then* $P \xrightarrow{\lambda} P'' \iff P' \xrightarrow{\lambda[x'/x]} P''[x'/x]$*.*

Proof. The proof is by induction on the form of P. We report the most interesting cases. Below we consider $t' = t \lhd (\, x', x(t)\,) \lhd (\, x, t_\perp\,)$.

Case $P = o(y)$ `from` $e; B \cdot t \cdot M$
On both P and P' the only applicable Rule is $\lfloor^{\mathrm{DCC}}|_{\mathrm{RECV}}\rceil$, hence we consider the interesting case in which M contains a message for the queue defined by e. In the other case the Lemma holds as both P and P' cannot reduce. The case unfolds on the combination of whether *i*) $y \neq x$ and *ii*) expression e contains x. Below we consider the comprehensive case for $y = x$ and e that contains x. The proof of the other cases is either trivial or a slight modification of the reported one.

Since we assume we can apply Rule $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$ we take $t_c = \mathsf{eval}(e, t)$ and $M(t_c) = (o, t') :: \tilde{m}$. From Definition 38 we have that $t_c = \mathsf{eval}(e[x'/x], t')$.

Therefore we have the reductions $P \xrightarrow{o \texttt{ from } e} B \cdot t \lhd (\, x, t_m \,) \cdot M[t_c \mapsto \tilde{m}]$ and $P' \xrightarrow{o \texttt{ from } e[x'/x]} B[x'/x] \cdot t' \lhd (\, x', t_m \,) \cdot M[t_c \mapsto \tilde{m}]$.

Case $P = \sum_{i \in I} [o_i(x_i) \texttt{ from } e] \{B_i\} \cdot t \cdot M$
The only applicable Rule on both P and P' is $\lfloor^{\text{DCC}}|_{\text{CHOICE}}\rceil$. The most comprehensive case is for M that contains a message for operation o_j, $j \in I$ where $x_j = x$ and expression e contains x. The remainder of the proof follows that of the previous case.

Case $P = \mathsf{if}\ e\ \{B_1\}\ \mathsf{else}\ \{B_2\} \cdot t \cdot M$
Trivial by Definition 38 for which $\mathsf{eval}(e, t) = \mathsf{eval}(e[x'/x], t')$.

Case $P = y = e; B \cdot t \cdot M$
The only applicable Rule on both P and P' is $\lfloor^{\text{DCC}}|_{\text{ASSIGN}}\rceil$. The most comprehensive case is for $y = x$ and expression e that contains x. The case is proven considering that, by Definition 38, it holds that $\mathsf{eval}(e, t) = \mathsf{eval}(e[x'/x], t')$.

Case $P = \mathsf{def}\ X = B_1\ \mathsf{in}\ B \cdot t \cdot M$
The thesis follows from the application of Rule $\lfloor^{\text{DCC}}|_{\text{CTX}}\rceil$ and the induction hypothesis.

□

Lemma 29 (DCC Service Variable Renaming)**.** *Let S and S' be two DCC services such that $S = \langle B_s, P \mid Q\rangle_l$ and $S' = \langle B_s, P[x'/x] \mid Q\rangle_l$ then*
$S \xrightarrow{\lambda} \langle B_s, P'' \mid Q'\rangle_l \iff S' \xrightarrow{\lambda[x'/x]} \langle B_s, P''[x'/x] \mid Q'\rangle_l$.

Proof. The proof is by induction on the derivation of S. Below we consider $t' = t \lhd (\, x', x(t) \,) \lhd (\, x, t_\perp \,)$. We report the most interesting cases.

Case $\lfloor^{\text{DCC}}|_{\text{EQ}}\rceil$
The thesis follows from the application of the induction hypothesis on $S \equiv S_1$, $S_1 \to S_1'$, and $S_1' \equiv S'$.

Case $\lfloor^{\text{DCC}}|_{\text{PAR}}\rceil$
From Lemma 28 we have that, for any form of P of the kind $P = B \cdot t \cdot M$, $P[x'/x]$ can mimic the reduction of P such that $P \to P''$ and $P[x'/x] \to P''[x'/x]$ and vice versa. In this case Q remains the same in both reductions.

Case $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$
We consider the case where $P = \mathsf{cq}(x); B \cdot t \cdot M$. Let $Q = \prod_i B_i \cdot t_i \cdot M_i$ and $t_c \notin \bigcup_i \mathsf{dom}(M_i) \cup \mathsf{dom}(M)$. We have the reduction
$S \xrightarrow{\mathsf{cq}(x)} \langle B_s, B \cdot t \lhd (\, x, t_c, \cdot\,) M[t_c \mapsto \varepsilon] \mid Q\rangle_l$. S' can mimic such behaviour by taking the fresh value $t'_c = t_c$, obtaining the reduction
$S' \xrightarrow{\mathsf{cq}(x')} \langle B_s, B \cdot t \lhd (\, x', t'_c, \cdot\,) M[t'_c \mapsto \varepsilon] \mid Q\rangle_l$. The same holds if we let S' reduce and prove that S can mimic it.

Case $\lfloor^{\text{DCC}}|_{\text{InSend}}\rceil$
We consider the case where $P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \cdot M$ and expressions e_1, e_2 and e_3 contain x. From Definition 38 we know that $\mathsf{eval}(e_1, t) = \mathsf{eval}(e_1[x'/x], t')$. Similarly the couples e_2 and $e_2[x'/x]$ and e_3 and $e_3[x'/x]$ enjoy the same property when evaluated respectively on t and t'.

We analyse the case in which P moves and $P[x'/x]$ mimics it. The other case, for $P[x'/x]$ that reduces and P that mimics it, follows the same structure. Since $\lfloor^{\text{DCC}}|_{\text{InSend}}\rceil$ applies we can assume that $Q = B \cdot t \cdot M \mid B_r \cdot t_r \cdot M_r \mid P_1$.

$$\frac{\begin{array}{ccc} P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \cdot M & \mathsf{eval}(e_1, t) = l & \mathsf{eval}(e_3, t) = t_c \\ \mathsf{eval}(e_2, t) = t_m & \multicolumn{2}{c}{M'_r = M_r[t_c \mapsto M_r(t_c) :: (o, t_m)]} \end{array}}{\langle B_s,\ P \mid B_r \cdot t_r \cdot M_r \mid P_1\rangle_l \xrightarrow{o@e_3} \langle B_s,\ B \cdot t \cdot M \mid B_r \cdot t_r \cdot M'_r \mid P_1\rangle_l}\ \lfloor^{\text{DCC}}|_{\text{InSend}}\rceil$$

and

$$\frac{\begin{array}{c} P[x'/x] = o@e_1[x'/x](e_2[x'/x])\ \mathtt{to}\ e_3[x'/x]; B[x'/x] \cdot t' \cdot M \\ \mathsf{eval}(e_1[x'/x], t) = l \qquad \mathsf{eval}(e_3[x'/x], t) = t_c \\ \mathsf{eval}(e_2[x'/x], t) = t_m \quad M'_r = M_r[t_c \mapsto M_r(t_c) :: (o, t_m)] \end{array}}{\begin{array}{c} \langle B_s,\ P[x'/x] \mid B_r \cdot t_r \cdot M_r \mid P_1\rangle_l \xrightarrow{o@e_3[x'/x]} \\ \langle B_s,\ B[x'/x] \cdot t' \cdot M \mid B_r \cdot t_r \cdot M'_r \mid P_1\rangle_l \end{array}}\ \lfloor^{\text{DCC}}|_{\text{InSend}}\rceil$$

Case $\lfloor^{\text{DCC}}|_{\text{InStart}}\rceil$
We consider the case where $P = ?@e_1(e_2); B \cdot t \cdot M$ and expressions e_1 and e_2 contain x. From Definition 38 we know that $\mathsf{eval}(e_1, t) = \mathsf{eval}(e_1[x'/x], t')$. Similarly e_2 and $e_2[x'/x]$ and enjoy the same property when evaluated respectively on t and t'.

Below we describe the case in which P moves and $P[x'/x]$ mimics it. The other case, for $P[x'/x]$ that reduces and P that mimics it, follows the same structure. We assume the start behaviour $B_s = ?(y); B_c$.

$$\frac{P = ?@e_1(e_2); B \cdot t \cdot M \quad \mathsf{eval}(e_1, t) = l \quad P_1 = B_c \cdot t_\perp \triangleleft (\, y, \mathsf{eval}(e_2, t)\,) \cdot \emptyset}{\langle !(y); B_c,\ P \mid Q \rangle_l \quad \xrightarrow{?(e_2)} \quad \langle !(y); B_c,\ P_1 \mid B \cdot t \cdot M \mid Q \rangle_l} \lfloor^{\text{DCC}}|_{\text{InStart}}\rceil$$

and

$$\frac{\begin{array}{c} P' = ?@e_1[x'/x](e_2[x'/x]); B[x'/x] \cdot t' \cdot M \\ \mathsf{eval}(e_1[x'/x], t') = l \qquad P_1 = B_c \cdot t_\perp \triangleleft (\, y, \mathsf{eval}(e_2[x'/x], t')\,) \cdot \emptyset \end{array}}{\langle !(y); B,\ P' \mid Q \rangle_l \quad \xrightarrow{?(e_2[x'/x])} \quad \langle !(y); B',\ P_1 \mid B[x'/x] \cdot t' \cdot M \mid Q \rangle_l} \lfloor^{\text{DCC}}|_{\text{InStart}}\rceil$$

□

Lemma 30 (DCC Network Variable Renaming)**.** *Let S and S' be two DCC network such that $S = \langle B_s, P \mid Q \rangle_l \mid S_*$ and $S' = \langle B_s, P[x'/x] \mid Q \rangle_l \mid S_*$ then $S \xrightarrow{\lambda} \langle B_s, P'' \mid Q' \rangle_l \mid S'_* \iff S' \xrightarrow{\lambda[x'/x]} \langle B_s, P''[x'/x] \mid Q' \rangle_l \mid S'_*$.*

Proof. The proof is by induction on the derivation of S. Below we consider $t' = t \triangleleft (\, x', x(t)\,) \triangleleft (\, x, t_\perp\,)$. We report the most interesting cases.

Case $\lfloor^{\text{DCC}}|_{\text{Send}}\rceil$
We know that $P = o@e_1(e_2)$ $\texttt{to}$ $e_3; B \cdot t \cdot M$. We consider the case in which all expressions e_1, e_2, and e_3 contain x. Let suppose that $\mathsf{eval}(e_1, t) = l'$ and $\mathsf{eval}(e_3, t) = t_c$ then, since $\lfloor^{\text{DCC}}|_{\text{Send}}\rceil$ applies, we know that
$S_* \equiv \langle B'_s, B_r \cdot t_r \cdot M_r \mid P_2 \rangle_{l'}$ where $t_c \in \mathsf{dom}(M_r)$.

From Definition 38 we know that $\mathsf{eval}(e_1, t) = \mathsf{eval}(e_1[x'/x], t')$. Similarly the couples e_2 and $e_2[x'/x]$ and e_3 and $e_3[x'/x]$ enjoy the same property when evaluated respectively on t and t'.

We proceed proving that S reduces and S' can mimic it. The other case, for S' that reduces and S that mimics it, follows the same structure.

$$\frac{\begin{array}{c} P = o@e_1(e_2)\ \texttt{to}\ e_3; B \cdot t \cdot M \quad \mathsf{eval}(e_1, t) = l' \quad \mathsf{eval}(e_3, t) = t_c \\ \mathsf{eval}(e_2, t) = t_m \qquad M'_r = M_r[t_c \mapsto M_r(t_c) :: (o, t_m)] \end{array}}{\begin{array}{c} \langle B_s, P \mid P_1 \rangle_l \mid \langle B'_s, B_r \cdot t_r \cdot M_r \mid P_2 \rangle_{l'} \quad \xrightarrow{o@e_3} \\ \langle B_s, B \cdot t \cdot M \mid P_1 \rangle_l \mid \langle B'_s, B_r \cdot t_r \cdot M'_r \mid P_2 \rangle_{l'} \end{array}} \lfloor^{\text{DCC}}|_{\text{Send}}\rceil$$

and

$$\dfrac{\begin{array}{c} P[x'/x] = o@e_1[x'/x](e_2[x'/x])\ \mathtt{to}\ e_3[x'/x]; B[x'/x]\cdot t\cdot M \\ \mathsf{eval}(e_1[x'/x],t)=l' \qquad \mathsf{eval}(e_3[x'/x],t)=t_c \\ \mathsf{eval}(e_2[x'/x],t)=t_m \qquad M_r' = M_r[t_c \mapsto M_r(t_c) :: (o,t_m)] \end{array}}{\begin{array}{c}\langle B_s, P[x'/x] \mid P_1\rangle_l \mid \langle B_s', B_r\cdot t_r\cdot M_r \mid P_2\rangle_{l'} \xrightarrow{o@e_3[x'/x]} \\ \langle B_s, B[x'/x]\cdot t'\cdot M \mid P_1\rangle_l \mid \langle B_s', B_r\cdot t_r\cdot M_r' \mid P_2\rangle_{l'}\end{array}} \lfloor^{\mathrm{DCC}}|_{\mathrm{SEND}}\rceil$$

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{START}}\rceil$
We know that $P = ?@e_1(e_2); B\cdot t\cdot M$. We consider the case in which both expressions e_1 and e_2 contain x. Let suppose that $\mathsf{eval}(e_1,t)=l'$. Since $\lfloor^{\mathrm{DCC}}|_{\mathrm{SEND}}\rceil$ applies, we know that $S_* \equiv \langle !(y); B_c, P_1\rangle_{l'}$.

From Definition 38 we know that $\mathsf{eval}(e_1,t) = \mathsf{eval}(e_1[x'/x],t')$. Similarly also e_2 and $e_2[x'/x]$ enjoy the same property when evaluated respectively on t and t'.

We proceed proving that S reduces and S' can mimic it. The other case, for S' that reduces and S that mimics it, follows the same structure.

$$\dfrac{\begin{array}{c} P = ?@e_1(e_2); B\cdot t\cdot M \qquad \mathsf{eval}(e_1,t_1)=l \\ P_2 = B\cdot t_\perp \lhd (\,y, \mathsf{eval}(e_2,t)\,)\cdot\emptyset \end{array}}{\begin{array}{c}\langle !(y); B_c,\ P_1\rangle_{l'} \mid \langle B_s',\ P \mid Q\rangle_l \xrightarrow{?(e_2)} \\ \langle !(y); B_c,\ P_2 \mid P_1\rangle_{l'} \mid \langle B_s',\ B\cdot t\cdot M \mid Q\rangle_l\end{array}} \lfloor^{\mathrm{DCC}}|_{\mathrm{START}}\rceil$$

$$\dfrac{\begin{array}{c} P[x'/x] = ?@e_1[x'/x](e_2[x'/x]); B[x'/x]\cdot t'\cdot M \qquad \mathsf{eval}(e_1[x'/x],t_1)=l' \\ P_2 = B\cdot t_\perp \lhd (\,y, \mathsf{eval}(e_2[x'/x],t')\,)\cdot\emptyset \end{array}}{\begin{array}{c}\langle !(y); B_c,\ P_1\rangle_{l'} \mid \langle B_s',\ P[x'/x] \mid Q\rangle_l \xrightarrow{?(e_2[x'/x])} \\ \langle !(y); B_c,\ P_2 \mid P_1\rangle_{l'} \mid \langle B_s',\ B[x'/x]\cdot t'\cdot M \mid Q\rangle_l\end{array}} \lfloor^{\mathrm{DCC}}|_{\mathrm{START}}\rceil$$

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{EQ}}\rceil$ and $\lfloor^{\mathrm{DCC}}|_{\mathrm{SPAR}}\rceil$
In both cases the thesis follows from the application of the induction hypothesis and Lemma 29.

□

We report below the statement of Theorem 8, enriched with annotation on the transitions of D, C.
Theorem 8 *(Compilation Operational Correspondence)*
Let C be a composition of endpoint choreographies such that $\Gamma \vdash D, C$. Then we have that:

1. (Completeness) $D, C \xrightarrow{\beta} D', C'$ implies *i)* $\boxed{D, C}^{\Gamma} \rightarrow^{+} \boxed{D', C'}^{\Gamma'}$ for some Γ' such that *ii)* $\Gamma' \vdash D', C'$.

2. (Soundness) $\boxed{D, C}^{\Gamma} \rightarrow^{*} S$ implies *i)* $D, C \rightarrow^{*} D', C'$ and *ii)* $S \rightarrow^{*} \boxed{D', C'}^{\Gamma'}$ for some D', C', and Γ' such that *iii)* $\Gamma' \vdash D', C'$.

As a convention in the following we use the shortcuts $t_{\mathsf{p}} = D(\mathsf{p}).\mathsf{st}$ and $M_{\mathsf{p}} = D(\mathsf{p}).\mathsf{que}$ for p process in D.

Proof (Completeness). We proceed by induction on the derivation of $D, C \rightarrow D', C'$.

Case $\lfloor^{\text{C}}|_{\text{SEND}}\rceil$
We know that

- $C \equiv C_{\mathsf{p}} \mid C_c$ with $C_{\mathsf{p}} = k : \mathsf{p}[\mathsf{A}].e \text{ -> } \mathsf{B}.o; C'_p$;
- $D, C \xrightarrow{\beta} D', C'$ with $\lfloor^{\text{C}}|_{\text{SEND}}\rceil$ being the last applied Rule, where $\beta = k : \mathsf{p}[\mathsf{A}].e \text{ -> } \mathsf{B}.o$. $C' = C'_{\mathsf{p}} \mid C_c$; $D' = D\big[\mathsf{q} \mapsto (\ t_{\mathsf{q}}, M_{\mathsf{q}}[t_c \mapsto \tilde{m} :: (o_j, t_m)]\ )\big]$ by Rule $\lfloor^{\text{D}}|_{\text{SEND}}\rceil$, $t_c = \underline{\mathbf{k.A.B}}(t_{\mathsf{p}})$ and $t_m = \mathsf{eval}(e, t_{\mathsf{p}})$

Since $\lfloor^{\text{C}}|_{\text{SEND}}\rceil$ applies and there exists $\Gamma \vdash D, C$, then there exists a queue associated with a process q in D where the message is delivered.

We have two cases, whether the receiving process q is in the same location of the sender p or not. Let $\mathsf{p} \in D(l)$

Case $\mathsf{q} \in D(l)$

From Definition 23 we have that $\boxed{D, C}^{\Gamma} \equiv S \mid S_c$ where

- $S = \left\langle \boxed{C_c|_l}^{\Gamma},\ P \mid Q \mid R \right\rangle_l$
- $P = o@\underline{\mathbf{k.B.l}}(e)\ \mathtt{to}\ \underline{\mathbf{k.A.B}}; \boxed{C'_{\mathsf{p}}}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}}$
- $Q = \boxed{C_c|_{\mathsf{q}}}^{\Gamma} \cdot t_{\mathsf{q}} \cdot M_{\mathsf{q}}$
- $R = \prod_{\mathsf{r}\, \in\, D(l)/\{\mathsf{p},\mathsf{q}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}$
- $S_c = \prod_{l'\, \in\, \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma},\ \prod_{\mathsf{s} \in D(l')} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}} \right\rangle_{l'}$

In this case $\boxed{D, C}^{\Gamma}$ can mimic D, C applying Rules $\lfloor^{\text{DCC}}|_{\text{EQ}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{SPAR}}\rceil$, and $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$ where $S \mid S_c \rightarrow S' \mid S_c$ with $\lfloor^{\text{DCC}}|_{\text{SPAR}}\rceil$ and $S \rightarrow S'$ with

$$\frac{\begin{array}{ccc} o@\underline{\mathbf{k.B.l}}(e)\ \texttt{to}\ \underline{\mathbf{k.A.B}};\boxed{C'_{\mathsf{p}}}^{\Gamma}\cdot t_{\mathsf{p}}\cdot M_{\mathsf{p}} & \mathsf{eval}(\underline{\mathbf{k.B.l}}, t_{\mathsf{p}}) = l & \mathsf{eval}(\underline{\mathbf{k.A.B}}, t_{\mathsf{p}}) = t_c \\ \mathsf{eval}(e,t_{\mathsf{p}}) = t_m & & M'_{\mathsf{q}} = M_{\mathsf{q}}[t_c \mapsto M_{\mathsf{q}}(t_c) :: (o,t_m)] \end{array}}{\left\langle \boxed{C_c|_l}^{\Gamma}, P \mid Q \mid R\right\rangle_l \xrightarrow{k\colon \mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B}.o} \left\langle \boxed{C_c|_l}^{\Gamma}, P' \mid Q' \mid R\right\rangle_l}\ \lfloor^{\mathrm{DCC}}|_{\textsc{InSend}}\rceil$$

where $P' = \boxed{C'_{\mathsf{p}}}^{\Gamma}\cdot t_{\mathsf{p}}\cdot M_{\mathsf{p}}$ and $Q' = \boxed{C_c|_{\mathsf{q}}}^{\Gamma}\cdot t_{\mathsf{q}}\cdot M'_{\mathsf{q}}$.

From the hypothesis we know that $\Gamma = \Gamma_1, b[k]^{\mathtt{A}}_{\mathtt{B}}\colon T, k[\mathtt{A}]\colon !\mathtt{B}.o(U); T'$ (as $\Gamma \vdash D, C$).

By Rules $\lfloor^{\mathrm{T}}|_{\mathrm{DC}}\rceil$ and $\lfloor^{\mathrm{T}}|_{\textsc{Send}}\rceil$ we can find $\Gamma' = \Gamma_1, b[k]^{\mathtt{A}}_{\mathtt{B}}\colon T; ?\mathtt{A}.o(U), k[\mathtt{A}]\colon T'$ such that $\Gamma' \vdash D', C'$.

From Rule $\lfloor^{\mathrm{D}}|_{\textsc{Send}}\rceil$ and 23, we have that

$$\boxed{D', C'}^{\Gamma} \equiv \overbrace{\left\langle \boxed{C_c|_l}^{\Gamma}, \overbrace{\boxed{C'_{\mathsf{p}}}^{\Gamma}\cdot t_{\mathsf{p}}\cdot M_{\mathsf{p}}}^{P'} \mid \overbrace{\boxed{C_c|_{\mathsf{q}}}^{\Gamma}\cdot t_{\mathsf{q}}\cdot M'_{\mathsf{q}}}^{Q'} \mid \overbrace{\prod_{\mathsf{r}\,\in\, D'(l)/\{\mathsf{p},\mathsf{q}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma}\cdot t_{\mathsf{r}}\cdot M_{\mathsf{r}}}^{R} \right\rangle_l}^{S'}$$
$$\underbrace{\mid \prod_{l'\,\in\,\Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{s}\,\in\, D(l')} \boxed{C_c|_{\mathsf{s}}}^{\Gamma}\cdot t_{\mathsf{s}}\cdot M_{\mathsf{s}} \right\rangle_{l'}}_{S_c}$$

Therefore $\boxed{D', C'}^{\Gamma} = S' \mid S_c$, by construction of Γ' we can apply Lemma 27 for which $\boxed{D', C'}^{\Gamma} = \boxed{D', C'}^{\Gamma'}$ and therefore $S' \mid S_c = \boxed{D', C'}^{\Gamma'}$.

Case $\mathsf{q} \notin D(l)$

Similar to **Case** $\mathsf{q} \in D(l)$ except the last applied Rule for $\boxed{D, C}^{\Gamma} \to \boxed{D', C'}^{\Gamma'}$ is $\lfloor^{\mathrm{DCC}}|_{\textsc{Send}}\rceil$.

Case $\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil$
Does not apply since C is a composition of endpoint choreographies by hypothesis.

Case $\lfloor^{\mathrm{C}}|_{\textsc{Recv}}\rceil$
We know that

- $C \equiv C_{\mathsf{q}} \mid C_c$ with $C_{\mathsf{q}} = k : \mathtt{A}\ \texttt{->}\ \mathsf{q}[\mathrm{B}].\{o_i(x_i); C_i\}_{i\in I}$

- $D, C \xrightarrow{\beta} D, C'$ with Rule $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ where $\beta = k\colon \mathtt{A} \rightsquigarrow \mathsf{q}[\mathtt{B}].o_j(x_j)$, $C' \equiv C_j \mid C_c$, and $D' = D\big[\mathsf{q} \mapsto (\ t_{\mathsf{q}} \lhd (\, x_j, t_m\,), M_{\mathsf{q}}[t_c \mapsto \tilde{m}]\ )\big]$ where $t_c = \underline{\mathbf{k.A.B}}(t_{\mathsf{q}})$ and $M_{\mathsf{q}}(t_c) = (o_j, t_m) :: \tilde{m}$;

Let $\mathsf{q}@l \in \Gamma$, from the Definition 23 we have

$\boxed{D, C}^{\Gamma} \equiv S \mid S_c$ where

- $S = \left\langle \boxed{C_c|_l}^{\Gamma}, Q \mid R \right\rangle_l$
- $Q = \sum_{i \in I} [o_i(x_i)\ \mathtt{from}\ \underline{\mathbf{k.A.B}}]\ \left\{ \boxed{C_i}^{\Gamma} \right\}\ \cdot t_{\mathsf{q}} \cdot M_{\mathsf{q}}$
- $\prod_{\mathsf{r} \in D(l)/\{\mathsf{q}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}$
- $S_c = \prod_{l' \in \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{s} \in D(l')} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}} \right\rangle_{l'}$

In this case $\boxed{D, C}^{\Gamma}$ can mimic D, C applying Rules $\lfloor^{\mathrm{DCC}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{SPAR}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{PPAR}}\rceil$, and $\lfloor^{\mathrm{DCC}}|_{\mathrm{CHOICE}}\rceil$ where

$$\dfrac{\dfrac{\dfrac{j \in I \quad t_c = \mathsf{eval}(\underline{\mathbf{k.A.B}}, t_{\mathsf{q}}) \quad M_{\mathsf{q}} = (o_j, t_m) :: \tilde{m}}{\sum_{i \in I} [o_i(x_i)\ \mathtt{from}\ \underline{\mathbf{k.A.B}}]\ \left\{ \boxed{C_i}^{\Gamma} \right\}\ \cdot t_{\mathsf{q}} \cdot M_{\mathsf{q}} \quad \to \quad \boxed{C_j}^{\Gamma} \cdot t_{\mathsf{q}} \lhd (\, x_j, t_m\,) \cdot M_{\mathsf{q}}[t_c \mapsto \tilde{m}]}\ \lfloor^{\mathrm{DCC}}|_{\mathrm{CHOICE}}\rceil}{\langle C_c|_l\,, Q \mid R \rangle_l \quad \to \quad \left\langle C_c|_l\,, \boxed{C_j}^{\Gamma} \cdot t_{\mathsf{q}} \lhd (\, x_j, t_m\,) \cdot M_{\mathsf{q}}[t_c \mapsto \tilde{m}] \mid R \right\rangle_l}\ \lfloor^{\mathrm{DCC}}|_{\mathrm{PPAR}}\rceil}{S \mid S_c \quad \to \quad S' \mid S_c}\ \lfloor^{\mathrm{DCC}}|_{\mathrm{SPAR}}\rceil$$

Let $Q' = \boxed{C_j}^{\Gamma} \cdot t_{\mathsf{q}} \lhd (\, x_j, t_m\,) \cdot M_{\mathsf{q}}[t_c \mapsto \tilde{m}]$.

From the hypothesis we know that $\Gamma = \Gamma_1, b[k]^{\mathtt{A}}_{\mathtt{B}} \colon ?\mathtt{A}.o_j(U_j); T, k[\mathtt{B}] \colon ?\mathtt{A}.\{o_i(U_i); T_i\}_{i \in I}$ and we can find $\Gamma' = \Gamma_1, b[k]^{\mathtt{A}}_{\mathtt{B}} \colon T, k[\mathtt{B}] \colon T_j$ such that $\Gamma' \vdash D', C'$.

From Rule $\lfloor^{\mathrm{D}}|_{\mathrm{RECV}}\rceil$ and Definition 23, we have that, let $t'_{\mathsf{q}} = t_{\mathsf{q}} \lhd (\, x_j, t_m,\ )$ and $M'_{\mathsf{q}} = M_{\mathsf{q}}[t_c \mapsto \tilde{m}]$

$$\boxed{D', C'}^{\Gamma} \equiv \overbrace{\left\langle \boxed{C_c|_l}^{\Gamma}, \overbrace{\boxed{C_j|_{\mathsf{q}}}^{\Gamma} \cdot t'_{\mathsf{q}} \cdot M'_{\mathsf{q}}}^{Q'} \mid \overbrace{\prod_{\mathsf{r} \in D'(l)/\{\mathsf{q}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}}^{R} \right\rangle_l}^{S'} \mid \underbrace{\prod_{l' \in \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{s} \in D(l')} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}} \right\rangle_{l'}}_{S_c}$$

Therefore $\boxed{D', C'}^{\Gamma} = S' \mid S_c$, by construction of Γ' we can apply Lemma 27 for which $\boxed{D', C'}^{\Gamma} = \boxed{D', C'}^{\Gamma'}$ and therefore $S' \mid S_c = \boxed{D', C'}^{\Gamma'}$.

Case $\lfloor^{\mathsf{C}}|_{\textsc{Start}}\rceil$
Does not apply since C is a composition of endpoint choreographies by hypothesis.

Case $\lfloor^{\mathsf{C}}|_{\textsc{PStart}}\rceil$
We know that

- $C \equiv C_r \mid C_a \mid C_c$ where, let $\tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \in \Gamma$
- $C_r = \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow \widetilde{l.\mathtt{B}}; C'_r$
- let $l_1.\mathtt{B}_1, \ldots, l_n.\mathtt{B}_n = \widetilde{l.\mathtt{B}}$, $C_a = \prod_{i=1}^{n} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; C_{\mathsf{q}_i}$

We can apply Rules $\lfloor^{\mathsf{C}}|_{\textsc{Par}}\rceil$ and $\lfloor^{\mathsf{C}}|_{\textsc{Eq}}\rceil$ and lastly Rule $\lfloor^{\mathsf{C}}|_{\textsc{PStart}}\rceil$ such that

$$\frac{\begin{array}{c} i \in \{1,\ldots,n\} \qquad \#k' \qquad \{\widetilde{l.\mathtt{B}}\} = \biguplus_i \{l_i.\mathtt{B}_i\} \qquad \#\tilde{\mathsf{r}} \qquad \{\tilde{\mathsf{r}}\} = \bigcup_i \{\tilde{\mathsf{r}}_i\} \\ \mathsf{p} \in D(l) \qquad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], l_1.\mathsf{r}_1[\mathtt{B}_1], \ldots, l_n.\mathsf{r}_n[\mathtt{B}_n] \qquad D, \delta \blacktriangleright D' \end{array}}{D, C_r \mid C_a \quad \xrightarrow{\tau} \quad D',\ C'_r[k'/k] \mid \prod_{i=1}^{n} \big(\, C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i] \,\big) \mid C_a} \ \lfloor^{\mathsf{C}}|_{\textsc{PStart}}\rceil$$

and

$$D, C_r \mid C_a \mid C_c \quad \to \quad D', C'_r[k'/k] \mid \prod_i \big(\, C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i] \,\big) \mid C_a \mid C_c$$

thus $C' = C'_r[k'/k] \mid \prod_i \big(\, C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i] \,\big) \mid C_a \mid C_c$

From the hypothesis we know that $\Gamma \vdash D, C$ and therefore that $\Gamma = \Gamma_1, \tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle$. We can find $\Gamma' = \Gamma, \mathsf{init}(k', (\mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]}), G)$ and $\Gamma' \vdash D', C'$.

Remark 7. *We have two cases for, let* $\mathsf{p}@l \in \Gamma$*, whether* $l \in \{\tilde{l}\}$ *or not. For a clearer treatment of the case we proceed considering that* $l \notin \{\tilde{l}\}$ *(i.e., no service process is created in the same location — service — of the requester* p*). The other case follows the same structure of* $l \notin \{\tilde{l}\}$ *although the service located at* l *has* $\boxed{C_a|_l}^{\Gamma}$ *as start behaviour and* $\boxed{D, C}^{\Gamma}$ *applies Rule* $\lfloor^{DCC}|_{\textsc{InStart}}\rceil$ *in place of the* $\lfloor^{DCC}|_{\textsc{Start}}\rceil$ *for starting the DCC process located at* l*.*

Henceforth we proceed analysing the case for $l \notin \{\tilde{l}\}$*.*

From Definition 23 we have

$$\boxed{D', C'}^{\Gamma'} = \left\langle \boxed{C_c|_l}^{\Gamma'}, P'' \mid R' \right\rangle_l \mid \prod_{i=1}^{n} \langle Q''_i, Q^*_i \mid R'_{l_i} \rangle_{l_i} \mid S'_c$$

In the following, we use the shortcuts $t^*_{\mathsf{s}} = D'(\mathsf{s}).\mathsf{st}$ and $M^*_{\mathsf{s}} = D'(\mathsf{s}).\mathsf{que}$ for s process in D'.

- $P'' = \boxed{C'_r[k'/k]}^{\Gamma'} \cdot t^*_{\mathsf{p}} \cdot M^*_{\mathsf{p}}$
- $R' = \prod_{\mathsf{p}' \in D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{p}'}}^{\Gamma'} \cdot t^*_{\mathsf{p}'} \cdot M^*_{\mathsf{p}'}$
- $Q''_i = \mathsf{accept}(k, \mathtt{B}_i, G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle); \boxed{C_{\mathsf{q}_i}}^{\Gamma'}$
- $Q^*_i = \boxed{C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i]}^{\Gamma'} \cdot t^*_{q_i} \cdot M^*_{q_i}$
- $R'_{l_i} = \prod_{\mathsf{s} \in D(l_i)} \boxed{C_c|_{\mathsf{s}}}^{\Gamma'} \cdot t^*_{\mathsf{s}} \cdot M^*_{\mathsf{s}}$
- $S'_c = \prod_{l' \in \Gamma/\{l, \tilde{l}\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma'}, \prod_{\mathsf{s}' \in D(l')} \boxed{C_c|_{\mathsf{s}'}}^{\Gamma'} \cdot t^*_{\mathsf{s}'} \cdot M^*_{\mathsf{s}'} \right\rangle_{l'}$

From Rule $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$ we know that

$$\underline{\mathbf{k'}}(t^*_{\mathsf{p}}) = \underline{\mathbf{k'}}(t^*_{\mathsf{q}_1}) = \ldots = \underline{\mathbf{k'}}(t^*_{\mathsf{q}_n}) = t_{k'}$$

for some $t_{k'}$ session descriptor of session k'.

We proceed by proving that we can reduce $\boxed{D, C}^{\Gamma} \rightarrow^+ S$.

From Definition 23 we have

$$\boxed{D, C}^{\Gamma} \equiv \left\langle \boxed{C_c|_l}^{\Gamma}, P \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i, R_{l_i} \rangle_{l_i} \mid S_c$$

where

- $$\begin{array}{rl} P = & \mathsf{start}(\, k,\ (l.\mathtt{A}, \widetilde{l.\mathtt{B}})\,); \boxed{C'_r}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} = \\ = & \left(\begin{array}{l} \bigodot_{\underline{\mathbf{I}} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}} \ \underline{\mathbf{k.I.l}} = l_{\mathtt{I}} \ ; \\ \bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \Big(\mathsf{cq}(\underline{\mathbf{k.I.A}}); ?@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}}); \mathit{sync}(\underline{\mathbf{k}}) \ \mathtt{from} \ \underline{\mathbf{k.I.A}}\Big); \\ \bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \ \mathit{start}@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}}) \ \mathtt{to} \ \underline{\mathbf{k.A.I}}; \boxed{C'_r}^{\Gamma} \end{array} \right) \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} \end{array}$$

- $Q_i = \mathsf{accept}(k, \mathtt{B}_i, G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle); \boxed{C_{\mathsf{q}_i}}^{\Gamma} = \begin{array}{l} !(\underline{\mathbf{k}});\ \bigodot_{\underline{\mathbf{I}} \in \{\mathtt{A},\tilde{\mathtt{B}}\}/\{\mathtt{B}_i\}} \mathsf{cq}(\underline{\mathbf{k.I.B_i}})\ ; \\ sync@\underline{\mathbf{k.A.l}}(\underline{\mathbf{k}})\ \mathtt{to}\ \underline{\mathbf{k.B_i.A}}\ ; \\ start(\underline{\mathbf{k}})\ \mathtt{from}\ \underline{\mathbf{k.A.B_i}}\ ;\ \boxed{C_{\mathsf{q}_i}}^{\Gamma} \end{array}$
- $R = \prod_{\mathsf{p}' \in D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{p}'}}^{\Gamma} \cdot t_{\mathsf{p}'} \cdot M_{\mathsf{p}'}$
- $R_{l_i} = \prod_{\mathsf{s} \in D(l_i)} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}}$
- $S_c = \prod_{l' \in \Gamma/\{l,\tilde{l}\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{s}' \in D(l')} \boxed{C_c|_{\mathsf{s}'}}^{\Gamma} \cdot t_{\mathsf{s}'} \cdot M_{\mathsf{s}'} \right\rangle_{l'}$

$\boxed{D, C}^{\Gamma}$ can mimic D, C with the following sequence of reductions. Note that we make use of renaming on $(accept)$ terms in $Q_1, \ldots, Q_n$ and variable renaming on P (as of Definition 38) to align the evolution of $\boxed{D, C}^{\Gamma}$ with the evolution of D, C, in which k has been replaces with the fresh name k'.

Therefore we take $S_0^* \sim \boxed{D, C}^{\Gamma}$

$$S_0^* = \left\langle \boxed{C_c|_l}^{\Gamma}, P[\underline{\mathbf{k}'}/\underline{\mathbf{k}}] \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i[\underline{\mathbf{k}'}/\underline{\mathbf{k}}], R_{l_i} \rangle_{l_i} \mid S_c$$

$$S_0^* \to \begin{array}{l}
\xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{PPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{ASSIGN}} \rceil} \Big\}\ \textcircled{1}\ n+1 \text{ times} \\
\left.\begin{array}{l}
\textcircled{2.1} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{PPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{CQ}} \rceil} \\
\textcircled{2.2} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{START}} \rceil} \\
\textcircled{2.3} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{PPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{CQ}} \rceil} \Big\}\ n \text{ times} \\
\textcircled{2.4} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{SEND}} \rceil} \\
\textcircled{2.5} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{RECV}} \rceil}
\end{array}\right\}\ \textcircled{2}\ \text{n times} \\
\left.\begin{array}{l}
\textcircled{3.1} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{SEND}} \rceil} \\
\textcircled{3.2} \xrightarrow{\lfloor \text{DCC}|_{\text{EQ}} \rceil \quad \lfloor \text{DCC}|_{\text{SPAR}} \rceil \quad \lfloor \text{DCC}|_{\text{RECV}} \rceil}
\end{array}\right\}\ \textcircled{3}\ n \text{ times}
\end{array} \to S_1^*$$

We briefly comment the numbered transitions.

- In $\textcircled{1}$ $P[k'/k]$ proceeds to store (for $n+1$ times, l plus $l_i, i \in \{1, \ldots, n\}$) the locations of all roles under $\underline{\mathbf{k}'}$.
- In $\textcircled{2}$, for each location $l_i, i \in \{1, \ldots, n\}$ (for each service process):

  - P creates its receiving queue for the service process $\textcircled{2.1}$;
  - in $\textcircled{2.2}$ we apply Rule $\lfloor^{\mathrm{DCC}}|_{\mathrm{EQ}}\rceil$ such that we rename
  - P synchronises with the service at location l_i starting ($\lfloor^{\mathrm{DCC}}|_{\mathrm{START}}\rceil$) a new service process;
  - in $\textcircled{2.3}$ the service process creates its own queues for all other roles in the session (hence n times);
  - in $\textcircled{2.4}$ the service process sends the correlation values to P;
  - finally P receives the message in $\textcircled{2.5}$.
- In $\textcircled{3}$ for each service process (n times) $\textcircled{3.1}$ the starter sends a message to the service process to start the session and $\textcircled{3.2}$ the addressee receives it.

Finally we have

$$S_1^* = \left\langle \boxed{C_c|_l}^{\Gamma} \mid P' \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i[\underline{\mathbf{k'}}/\underline{\mathbf{k}}], Q'_i \mid R_{l_i}\rangle_{l_i} \mid S_c$$

where

- $P' = \boxed{C'_r}^{\Gamma}[\underline{\mathbf{k'}}/\underline{\mathbf{k}}] \cdot t'_p \cdot M'_p$, and
- $Q'_i = \boxed{C_{\mathsf{q}_i}}^{\Gamma}[\underline{\mathbf{k'}}/\underline{\mathbf{k}}] \cdot t_{k'} \cdot M_{\mathsf{q}_i}$

From the transitions presented above we know that there exists $t'_{k'}$ such that $t'_{\mathsf{p}} = t_{\mathsf{p}} \lhd (\underline{\mathbf{k'}}, t'_{k'})$, where $t'_{k'}$ is a session descriptor for session k' (i.e., it contains all the locations and correlations keys used by the processes in session k'). In this case, we take $t'_k = t_{k'}$ obtained from the derivation $D, C \xrightarrow{\tau} D', C'$.

Similarly, M'_{p} and $M_{\mathsf{q}_1}, \ldots, M_{\mathsf{q}_n}$ contain the necessary (empty) queues to support communication in session k'.

$$M'_{\mathsf{p}} = M_{\mathsf{p}}[\underline{\mathbf{k'.B_1.A}}(t_{k'}) \mapsto \varepsilon] \ \ldots \ [\underline{\mathbf{k'.B_n.A}}(t_{k'}) \mapsto \varepsilon]$$

and

$$M_{\mathsf{q}_i} = \emptyset \begin{array}{l} [\underline{\mathbf{k'.A.B_i}}(t_k) \mapsto \varepsilon][\underline{\mathbf{k'.B_1.B_i}}(t_k) \mapsto \varepsilon] \ \ldots \ [\underline{\mathbf{k'.B_{i-1}.B_i}}(t_k) \mapsto \varepsilon] \ \ldots \\ \ldots \ [\underline{\mathbf{k'.B_{i+1}.B_i}}(t_k) \mapsto \varepsilon] \ \ldots \ [\underline{\mathbf{k'.B_n.B_i}}(t_k) \mapsto \varepsilon] \end{array}$$

We proceed with the proof taking $S \sim S_1^*$ as S is simply the renaming of $\underline{\mathbf{k'}}$ to $\underline{\mathbf{k}}$ on start behaviours $Q_i, i \in \{1, \ldots, n\}$ (trivially $Q_i[\underline{\mathbf{k'}}/\underline{\mathbf{k}}][\underline{\mathbf{k}}/\underline{\mathbf{k'}}] = Q_i$)

$$S = \left\langle \boxed{C_c|_l}^{\Gamma} \mid P' \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i, Q_i' \mid R_{l_i} \rangle_{l_i} \mid S_c$$

We now proceed to prove that $\boxed{D, C}^{\Gamma} \to^{+} \boxed{D', C''}^{\Gamma'}$, i.e. that $\boxed{D', C''}^{\Gamma'} = S$ with $\Gamma' \vdash D', C'$.

We prove that

$$\overbrace{\left\langle \boxed{C_c|_l}^{\Gamma'}, P'' \mid R' \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i'', Q_i^{*} \mid R_{l_i}' \rangle_{l_i} \mid S_c'}^{\boxed{D', C''}^{\Gamma'}} =$$

$$\overbrace{\left\langle \boxed{C_c|_l}^{\Gamma} \mid P' \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i, Q_i' \mid R_{l_i} \rangle_{l_i} \mid S_c}^{S}$$

- $\boxed{C_c|_l}^{\Gamma} = \boxed{C_c|_l}^{\Gamma'}$ as $\Gamma|_{\mathsf{locs}} = \Gamma'|_{\mathsf{locs}}$ by construction;
- $P'' = P'$ is proven by

  $$\boxed{C_r'[k'/k]}^{\Gamma'} \cdot t_{\mathsf{p}}^{*} \cdot M_{\mathsf{p}}^{*} = \boxed{C_r'}^{\Gamma} [\underline{\mathbf{k}}'/\underline{\mathbf{k}}] \cdot t_p' \cdot M_p'$$

  which holds as

  i) $\boxed{C_r'[k'/k]}^{\Gamma'} = \boxed{C_r'}^{\Gamma} [\underline{\mathbf{k}}'/\underline{\mathbf{k}}]$ since
    (a) Γ' does not contain any new process used in C_r';
    (b) by renaming, and
    (c) Definition 38.

  ii) $t_{\mathsf{p}}^{*} = t_{\mathsf{p}}'$ by construction and Rule $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$;

  iii) $M_{\mathsf{p}}^{*} = M_{\mathsf{p}}'$ by construction and Rule $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$.
- $Q_i'^{*} = Q_i'$ proven by

  $$\boxed{C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i]}^{\Gamma'} \cdot t_{q_i}^{*} \cdot M_{q_i}^{*} = \boxed{C_{\mathsf{q}_i}}^{\Gamma} [\underline{\mathbf{k}}'/\underline{\mathbf{k}}] \cdot t_{k'} \cdot M_{\mathsf{q}_i}$$

  whose proof of equivalence follows that of $P'' = P'$, except that Γ' contains the location of the process (r_i) used in $C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i]$.
- $Q_i'' = Q_i$ proven by

  $$\mathtt{accept}(k, \mathtt{B}_i, G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle); \boxed{C_{\mathsf{q}_i}}^{\Gamma'} = \mathtt{accept}(k, \mathtt{B}_i, G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle); \boxed{C_{\mathsf{q}_i}}^{\Gamma}$$

  which holds as $\boxed{C_{\mathsf{q}_i}}^{\Gamma'} = \boxed{C_{\mathsf{q}_i}}^{\Gamma}$ because Γ and Γ' contain the same service typings.

- $R' = R$ is proven by

$$\prod_{\mathsf{p}' \,\in\, D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{p}'}}^{\Gamma'} \cdot t^*_{\mathsf{p}'} \cdot M^*_{\mathsf{p}'} = \prod_{\mathsf{p}' \,\in\, D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{p}'}}^{\Gamma} \cdot t_{\mathsf{p}'} \cdot M_{\mathsf{p}'}$$

  for which

  i) $\boxed{C_c|_{\mathsf{p}'}}^{\Gamma'} = \boxed{C_c|_{\mathsf{p}'}}^{\Gamma}$ as Γ' does not contain any new process used in C_c.

  ii) $t^*_{\mathsf{p}'} = t_{\mathsf{p}'}$ unchanged by the reduction of D, C and $\boxed{D, C}^{\Gamma}$;

  iii) $M^*_{\mathsf{p}'} = M_{\mathsf{p}'}$ unchanged by the reduction of D, C and $\boxed{D, C}^{\Gamma}$.

- $R'_{l_i} = R_{l_i}$ whose proof follows that of $R' = R$.
- $S'_c = S_c$ following the proof of $\boxed{C_c|_l}^{\Gamma} = \boxed{C_c|_l}^{\Gamma'}$ and $R'_{l_i} = R_{l_i}$.

Case $\lfloor{}^{\mathrm{C}}|_{\mathrm{COND}}\rceil$

We have $C = C_{\mathsf{p}} \mid C_c$ where $C_{\mathsf{p}} = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}$. Let $\mathsf{p}@l \in \Gamma$ and

- $P = \mathsf{if}\ e\ \{\ \boxed{C_1}^{\Gamma}\}\ \mathsf{else}\ \{\ \boxed{C_2}^{\Gamma}\} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}}$;
- $R = \prod_{\mathsf{r} \,\in\, D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}$
- $S_c = \prod_{l' \,\in\, \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{r} \,\in\, D(l')} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}} \right\rangle_{l'}$

From Definition 23 we have,

$$\boxed{D, C}^{\Gamma} \equiv \left\langle \boxed{C_c|_l}^{\Gamma}, P \mid R \right\rangle_l \mid S_c$$

we reduce D, C applying Rules $\lfloor{}^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, $\lfloor{}^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$ and lastly Rule $\lfloor{}^{\mathrm{C}}|_{\mathrm{COND}}\rceil$. We analyse only the case for $\mathsf{eval}(e, t_{\mathsf{p}}) = \mathsf{true}$ as the other case for $\mathsf{eval}(e, t_{\mathsf{p}}) = \mathsf{false}$ follows the same structure.

$$D, C \xrightarrow{\tau @ \mathsf{p}} D, C_1 \mid C_c$$

and $C' = C_1 \mid C_c$ and $D' = D$. We can choose $\Gamma = \Gamma'$, for which it holds that $\Gamma \vdash D, C'$.

From Definition 23 we have

$$\boxed{D', C'}^{\Gamma'} = \boxed{D, C'}^{\Gamma} = \left\langle \boxed{C_c|_l}^{\Gamma}, \boxed{C_1}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} \mid R \right\rangle_l \mid S_c$$

$\boxed{D, C}^{\Gamma}$ can mimic D, C applying Rules $\lfloor{}^{\mathrm{DCC}}|_{\mathrm{EQ}}\rceil$, $\lfloor{}^{\mathrm{DCC}}|_{\mathrm{SPAR}}\rceil$, $\lfloor{}^{\mathrm{DCC}}|_{\mathrm{PPAR}}\rceil$, and lastly $\lfloor{}^{\mathrm{DCC}}|_{\mathrm{COND}}\rceil$ for which

$$\boxed{D, C}^{\Gamma} \to \left\langle \boxed{C_c|_l}^{\Gamma}, \boxed{C_1}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} \mid R \right\rangle_l \mid S_c$$

Case $\lfloor^{\mathrm{C}}|_{\mathrm{CTX}}\rceil$
The thesis follows from the induction hypothesis as D, C applies Rule $\lfloor^{\mathrm{C}}|_{\mathrm{CTX}}\rceil$ and $\boxed{D, C}^{\Gamma}$ can mimic it with Rule $\lfloor^{\mathrm{DCC}}|_{\mathrm{CTX}}\rceil$.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$
The thesis follows from the induction hypothesis.

Case $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$
The thesis follows from the induction hypothesis. Starting from any configuration of D, C, $\boxed{D, C}^{\Gamma}$ can always mimic the evolution of D, C when it applies Rule $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$: in both cases that $\mathcal{R} = \equiv$ or $\mathcal{R} = \simeq_{\mathsf{C}}$, $\boxed{D, C}^{\Gamma}$ can apply $\lfloor^{\mathrm{DCC}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{SPAR}}\rceil$, and $\lfloor^{\mathrm{DCC}}|_{\mathrm{PPAR}}\rceil$ to mimic D, C.

$\square$

Before proceeding with the proof of (Soundness) of Theorem 8 we extend the semantics of DCC with annotations on transitions to bear the paths on which operations are executed. We range over transition labels with λ.

$$\lambda ::= x \mid \mathsf{cq}(x) \mid ?(x) \mid o \;\texttt{from}\; x \mid o@x \mid \tau$$

We report in Figure E.3 the annotated semantics of DCC.

$$\dfrac{t_c = \mathsf{eval}(e,t) \quad M(t_c) = (o,t') :: \tilde{m}}{o(x)\ \mathtt{from}\ e; B \cdot t \cdot M \xrightarrow{o\ \mathtt{from}\ e} B \cdot t \lhd (\, x, t' \,) \cdot M[t_c \mapsto \tilde{m}]}\ \lfloor^{\text{DCC}}|_{\textsc{Recv}}\rceil$$

$$\dfrac{B \cdot t \cdot M \xrightarrow{\lambda} B' \cdot t' \cdot M'}{\mathsf{def}\ X = B_1\ \mathsf{in}\ B \cdot t \cdot M \xrightarrow{\lambda} \mathsf{def}\ X = B_1\ \mathsf{in}\ B' \cdot t' \cdot M'}\ \lfloor^{\text{DCC}}|_{\textsc{Ctx}}\rceil$$

$$\dfrac{i = 1 \text{ if } \mathsf{eval}(e,t) = \text{true}, i = 2 \text{ otherwise}}{\mathsf{if}\ e\ \{B_1\}\ \mathsf{else}\ \{B_2\} \cdot t \cdot M \xrightarrow{\tau} B_i \cdot t \cdot M}\ \lfloor^{\text{DCC}}|_{\textsc{Cond}}\rceil$$

$$\dfrac{j \in I \quad t_c = \mathsf{eval}(e,t) \quad M(t_c) = (o_j,t') :: \tilde{m}}{\sum_{i \in I} [o_i(x_i)\ \mathtt{from}\ e]\, \{B_i\}\ \cdot t \cdot M \xrightarrow{o_j\ \mathtt{from}\ e} B_j \cdot t \lhd (\, x_j, t' \,) \cdot M[t_c \mapsto \tilde{m}]}\ \lfloor^{\text{DCC}}|_{\textsc{Choice}}\rceil$$

$$\dfrac{t' = \mathsf{eval}(e,t)}{x = e; B \cdot t \cdot M \quad \xrightarrow{x} \quad B \cdot t \lhd (\, x, t' \,) \cdot M}\ \lfloor^{\text{DCC}}|_{\textsc{Assign}}\rceil$$

$$\dfrac{P \quad \xrightarrow{\lambda} \quad P'}{\langle B_s, P \mid P_1 \rangle_l \quad \xrightarrow{\lambda} \quad \langle B_s, P' \mid P_1 \rangle_l}\ \lfloor^{\text{DCC}}|_{\textsc{PPar}}\rceil$$

$$\dfrac{P = \mathsf{cq}(x); B \cdot t \cdot M \quad t_c \notin \bigcup_i \mathsf{dom}(M_i) \cup \mathsf{dom}(M) \quad t' = t \lhd (\, x, t_c \,)}{\langle B_s,\ P \mid \prod_i B_i \cdot t_i \cdot M_i \rangle_l \xrightarrow{\mathsf{cq}(x)} \langle B_s,\ B \cdot t' \cdot M[t_c \mapsto \varepsilon] \mid \prod_i B_i \cdot t_i \cdot M_i \rangle_l}\ \lfloor^{\text{DCC}}|_{\textsc{CQ}}\rceil$$

$$\dfrac{\begin{array}{ccc} P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \cdot M & \mathsf{eval}(e_1,t) = l & \mathsf{eval}(e_3,t) = t_c \\ \mathsf{eval}(e_2,t) = t_m & \multicolumn{2}{c}{M'' = M'[t_c \mapsto M'(t_c) :: (o,t_m)]} \end{array}}{\langle B_s,\ P \mid B' \cdot t' \cdot M' \mid P_1 \rangle_l \quad \xrightarrow{o@e_3} \quad \langle B_s,\ B \cdot t \cdot M \mid B' \cdot t' \cdot M'' \mid P_1 \rangle_l}\ \lfloor^{\text{DCC}}|_{\textsc{InSend}}\rceil$$

$$\dfrac{\begin{array}{ccc} P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \cdot M & \mathsf{eval}(e_1,t) = l' & \mathsf{eval}(e_3,t) = t_c \\ \mathsf{eval}(e_2,t) = t_m & \multicolumn{2}{c}{M'' = M'[t_c \mapsto M'(t_c) :: (o,t_m)]} \end{array}}{\begin{array}{l} \langle B_s, P \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M' \mid P_2 \rangle_{l'} \xrightarrow{o@e_3} \\ \qquad \langle B_s, B \cdot t \cdot M \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M'' \mid P_2 \rangle_{l'} \end{array}}\ \lfloor^{\text{DCC}}|_{\textsc{Send}}\rceil$$

$$\dfrac{\begin{array}{c} P_1 = ?@e_1(e_2); B_1 \cdot t_1 \cdot M_1 \\ \mathsf{eval}(e_1,t_1) = l \qquad Q = B \cdot t_\perp \lhd (\, x, \mathsf{eval}(e_2,t_1) \,) \cdot \emptyset \end{array}}{\begin{array}{l} \langle !(x); B,\ P \rangle_l \mid \langle B'_s,\ P_1 \mid P_2 \rangle_{l'} \quad \xrightarrow{?(e_2)} \\ \qquad \langle !(x); B,\ Q \mid P \rangle_l \mid \langle B'_s,\ B_1 \cdot t_1 \cdot M_1 \mid P_2 \rangle_{l'} \end{array}}\ \lfloor^{\text{DCC}}|_{\textsc{Start}}\rceil$$

$$\dfrac{P = ?@e_1(e_2); B \cdot t \cdot M \quad \mathsf{eval}(e_1,t) = l \quad Q = B \cdot t_\perp \lhd (\, x, \mathsf{eval}(e_2,t) \,) \cdot \emptyset}{\langle !(x); B',\ P \mid P_1 \rangle_l \quad \xrightarrow{?(e_2)} \quad \langle !(x); B',\ Q \mid B \cdot t \cdot M \mid P_1 \rangle_l}\ \lfloor^{\text{DCC}}|_{\textsc{InStart}}\rceil$$

$$\dfrac{S \equiv S_1 \quad S_1 \xrightarrow{\lambda} S'_1 \quad S'_1 \equiv S'}{S \quad \xrightarrow{\lambda} \quad S'}\ \lfloor^{\text{DCC}}|_{\textsc{Eq}}\rceil \qquad \dfrac{S_1 \xrightarrow{\lambda} S'_1}{S_1 \mid S \quad \xrightarrow{\lambda} \quad S'_1 \mid S}\ \lfloor^{\text{DCC}}|_{\textsc{SPar}}\rceil$$

Figure E.3: Correlation Calculus, annotated semantics

We also introduce the filtering and complement operators on sequences of DCC transition labels. Let $\lambda, \tilde{\lambda}$ be a sequence of DCC labels, the filtering of $\lambda, \tilde{\lambda}$ on k , written $(\lambda, \tilde{\lambda})\Big|_k$ is defined as

$$(\lambda, \tilde{\lambda})\Big|_k = \begin{cases} \lambda, (\tilde{\lambda}\Big|_k) & \text{if } \lambda \in \left\{ \begin{array}{c} \underline{\mathbf{k.x.y}},\ \mathsf{cq}(\underline{\mathbf{k.x.y}}),\ ?(\underline{\mathbf{k}}), \\ \mathit{sync}\ \mathtt{from}\ \underline{\mathbf{k.x.y}},\ \mathit{sync}@\underline{\mathbf{k.x.y}}, \\ \mathit{start}\ \mathtt{from}\ \underline{\mathbf{k.x.y}},\ \mathit{start}@\underline{\mathbf{k.x.y}} \end{array} \right\} \\ \tilde{\lambda}\Big|_k & \text{otherwise} \end{cases}$$

Let $\lambda_1, \tilde{\lambda}_1$ and $\lambda_2, \tilde{\lambda}_2$ be two sequences of DCC labels, the complement of $\lambda_1, \tilde{\lambda}$ on $\lambda_2, \tilde{\lambda}_2$, written $(\lambda_1, \tilde{\lambda}_1)/(\lambda_2, \tilde{\lambda}_2)$ is defined as

$$(\lambda_1, \tilde{\lambda}_1)/(\lambda_2, \tilde{\lambda}_2) = \begin{cases} \tilde{\lambda}_1/\tilde{\lambda}_2 & \text{if } \lambda_1 = \lambda_2 \wedge \tilde{\lambda}_2 \subseteq \tilde{\lambda}_1 \\ \lambda_1, (\tilde{\lambda}_1/(\lambda_2, \tilde{\lambda}_2)) & \text{if } \lambda_1 \neq \lambda_2 \wedge (\lambda_2, \tilde{\lambda}_2) \subseteq \tilde{\lambda}_1 \end{cases}$$

Below we state Lemma 32 that proves that we can permute the order of transitions in DCC systems and obtain the same final configurations. In particular prove such result on transitions that regard the start of new sessions. The proof of Lemma 32 is based on the simpler Lemma 31 that proves that we can change the order of one start transition and eventually obtain the same DCC system.

Lemma 31 (Single DCC Start Permutation)**.** *Let S be a composition of DCC services such that* $S \xrightarrow{\tilde{\lambda}_1} S_1$ *and* $S_1 \xrightarrow{\lambda_2} S_2$ *where* $\tilde{\lambda}_1\Big|_k = \varepsilon$ *and* $\lambda_2|_k = \lambda_2$ *then* $S \xrightarrow{\lambda_2} S_1'$ *and* $S_1' \xrightarrow{\tilde{\lambda}_1} S_2$*.*

Proof Sketch. The proof is by induction on the length of $\tilde{\lambda}_1$. Note that we can exclude to have transitions on session k in $\tilde{\lambda}_1$ since by the premises they happen before λ_2 which, by the premises, at most may correspond to the last transition of the start of session k. □

Lemma 32 (DCC Start Permutations)**.** *Let S be a composition of DCC services such that* $S \xrightarrow{\tilde{\lambda}} S'$ *and* $\tilde{\lambda} = (\underline{\mathbf{k.C.l}}, \tilde{\lambda}')$ *then* $S \xrightarrow{\tilde{\lambda}|_k} S''$ *and* $S'' \xrightarrow{\tilde{\lambda}/\tilde{\lambda}|_k} S'$*.*

Proof Sketch. Let $\tilde{\lambda}\Big|_k = \lambda_1^k, \ldots, \lambda_m^k$ and

$$\tilde{\lambda} = \lambda_1^k, \lambda_{1.1}, \ldots, \lambda_{1.n_1}, \ldots \lambda_i^k, \lambda_{i.1}, \ldots, \lambda_{i.n_i}, \ldots \lambda_m^k, \lambda_{m.1}, \ldots, \lambda_{m.n_m}$$

we can apply Lemma 32 inductively on each couple of the kind λ_i^k and $\lambda_{i.1}, \ldots, \lambda_{i.n_i}$, $i \in \{1, \ldots, n\}$. Note that the proof holds also for configurations of $\tilde{\lambda}$ in which the elements of $\tilde{\lambda}\Big|_k$ appear next to each other (e.g., $\tilde{\lambda} = \lambda_1^k, \ldots, \lambda_i^k, \lambda_i^k{+}1, \ldots, \lambda_{m.n_m}$) since it is a base case of Lemma 31 in which there are no transitions in $\tilde{\lambda}_1$. □

Next we state Lemma 33 that proves that, *i*) given a well-typed AC choreography, *ii*) its DCC projection, and *iii*) the DCC system that results from a sequence of reductions belonging to the start of a session, we can complete the remaining sequences of interactions and obtain the originating AC choreography after the reduction to start the session.

Lemma 33 (DCC Start Completion). *Let* $\Gamma \vdash D, C$,

$$C = \textbf{\textit{req}}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow l_1.[\mathtt{B}_1], \dots, l_n.[\mathtt{B}_n]; C_r \mid \prod_{i=1}^{n} \textbf{\textit{acc}}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; C_{\mathsf{q}_i}$$

and $\boxed{D, C}^{\Gamma} = S$ *such that* $S \xrightarrow{\tilde{\lambda}} S'$ *where* $\tilde{\lambda}\Big|_k = \tilde{\lambda}$ *then* i*)* $S' \xrightarrow{\tilde{\lambda}'} S''$, ii*)* $D, C \to D', C'$*, and* iii*) there exists some* Γ' *s.t.* $\Gamma' \vdash D', C'$ *and* $\boxed{D', C'}^{\Gamma'} = S''$.

Proof. Proof by case analysis on the length of $\tilde{\lambda}$.

Let $\mathsf{p}@l \in \Gamma$. To proceed, we have two subcases whether $l \in \{l_1, \dots, l_n\}$, i.e., whether one of the service processes is at the same location of p. Since the subcases follow the same structure, we detail only the proof for $l \notin \{l_1, \dots, l_n\}$ which allows for a uniform treatment. In the other case, *i*) we should account for transitions on the same service of p with Rules $\lfloor^{\text{DCC}}|_{\text{InStart}}\rceil$ and $\lfloor^{\text{DCC}}|_{\text{InSend}}\rceil$ and *ii*) we would have a newly created process in parallel with p in S'' and $\boxed{D, C}^{\Gamma'}$.

Provided n is the number of service processes involved in the start of the session k, from Definition 23 we can count the number of transitions needed to complete the start of a session. Indeed, given a D, C with

$$C = \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow l_1.[\mathtt{B}_1], \dots, l_n.[\mathtt{B}_n]; C_r \mid \prod_{i=1}^{n} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; C_{\mathsf{q}_i}$$

and $\boxed{D, C}^{\Gamma} = S$ then we can write the sequence of transitions of the projected DCC system

$$S \quad \overbrace{\xrightarrow{\underline{\mathbf{k.I.l}}}}^{\textcircled{1}}$$

$$\underbrace{\overbrace{\xrightarrow{\mathtt{cq}(\underline{\mathbf{k.A.I}})}}^{\textcircled{2.1}}\ \overbrace{\xrightarrow{?(\underline{\mathbf{k}})}}^{\textcircled{2.2}}\ \overbrace{\xrightarrow{\mathtt{cq}(\underline{\mathbf{k.I.I'}})}}^{\textcircled{2.3}}\ \overbrace{\xrightarrow{\mathit{sync}@\underline{\mathbf{k.I'.A}}}}^{\textcircled{2.4}}\ \overbrace{\xrightarrow{\mathit{sync}\ \mathtt{from}\ \underline{\mathbf{k.I'.A}}}}^{\textcircled{2.5}}}_{\textcircled{2}}$$

$$\underbrace{\overbrace{\xrightarrow{\mathit{start}@\underline{\mathbf{k.I'.A}}}}^{\textcircled{3.1}}\ \overbrace{\xrightarrow{\mathit{start}\ \mathtt{from}\ \underline{\mathbf{k.I'.A}}}}^{\textcircled{3.2}}}_{\textcircled{3}} \quad S''$$

and count the number of all the transitions to complete the start, let it be m, as the sum of:

$\textcircled{1}$ n+1 times, for $\underline{\mathbf{I}} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}$, with last Rule $\lfloor^{\text{DCC}}|_{\text{Assign}}\rceil$;

$\textcircled{2}$ n times, for $\underline{\mathbf{I}} \in \tilde{\mathtt{B}}$:

- $\textcircled{2.1}$ reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$;
- $\textcircled{2.2}$ reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{Start}}\rceil$;
- $\textcircled{2.3}$ n times for $\underline{\mathbf{I'}} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}/\{\underline{\mathbf{I}}\}$, reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$;
- $\textcircled{2.4}$ reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{Send}}\rceil$;
- $\textcircled{2.5}$ reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{Recv}}\rceil$;

$\textcircled{3}$ n times, for $\underline{\mathbf{I}} \in \tilde{\mathtt{B}}$:

- $\textcircled{3.1}$ reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{Send}}\rceil$;
- $\textcircled{3.2}$ reduces with last applied Rule $\lfloor^{\text{DCC}}|_{\text{Recv}}\rceil$;

and $m = n^2 + 7n + 1$. We proceed unfolding the proof on the length of $\tilde{\lambda}$.

Case $|\{\tilde{\lambda}\}| = 1$

Since the cardinality of $\tilde{\lambda}$ is one and that from the premises we know that $\tilde{\lambda}$ contains only transitions belonging to the start of session k, we can infer that $\tilde{\lambda} = \underline{\mathbf{k.C.l}}$ where $\mathtt{C} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}$.

To prove the thesis we let S' do all the remaining transitions to start the session and show that D, C can mimic it. Let $\widetilde{l.\mathtt{B}} = l_1.\mathtt{B}_1, \dots, l_n.\mathtt{B}_n$ and $\tilde{l} \colon G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \in \Gamma$.

From Definition 23 we have

$$\boxed{D, C}^{\Gamma} \equiv \left\langle \boxed{C_c|_l}^{\Gamma}, P \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i, R_{l_i} \rangle_{l_i} \mid S_c$$

where

$$P = \ \mathsf{start}(\ k,\ (l.\mathtt{A}, \widetilde{l.\mathtt{B}})\ ); \boxed{C_r}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} =$$

- $$= \left(\begin{array}{l} \bigodot\limits_{\underline{\mathtt{I}} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}} \ \underline{\mathbf{k.I.l}} = l_{\mathtt{I}}\ ; \\ \bigodot\limits_{\underline{\mathtt{I}} \in \{\tilde{\mathtt{B}}\}} \Big(\mathsf{cq}(\underline{\mathbf{k.I.A}}); ?@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}}); \mathit{sync}(\underline{\mathbf{k}})\ \mathtt{from}\ \underline{\mathbf{k.I.A}} \Big); \\ \bigodot\limits_{\underline{\mathtt{I}} \in \{\tilde{\mathtt{B}}\}} \ \mathit{start}@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}})\ \mathtt{to}\ \underline{\mathbf{k.A.I}}; \end{array} \right) ; \boxed{C_r}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}}$$

- $$Q_i = \mathsf{accept}(k, \mathtt{B}_i, G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{B}} \rangle); \boxed{C_{\mathsf{q}_i}}^{\Gamma} = \begin{array}{l} !(\underline{\mathbf{k}}); \quad \bigodot\limits_{\underline{\mathtt{I}} \in \{\mathtt{A}, \tilde{\mathtt{B}}\}/\{\mathtt{B}_i\}} \quad \mathsf{cq}(\underline{\mathbf{k.I.B_i}})\ ; \\ \mathit{sync}@\underline{\mathbf{k.A.l}}(\underline{\mathbf{k}})\ \mathtt{to}\ \underline{\mathbf{k.B_i.A}}\ ; \\ \mathit{start}(\underline{\mathbf{k}})\ \mathtt{from}\ \underline{\mathbf{k.A.B_i}}\ ; \ \boxed{C_{\mathsf{q}_i}}^{\Gamma} \end{array}$$

- $$R = \prod_{\mathsf{p}' \in D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{p}'}}^{\Gamma} \cdot t_{\mathsf{p}'} \cdot M_{\mathsf{p}'}$$

- $$R_{l_i} = \prod_{\mathsf{s} \in D(l_i)} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}}$$

- $$S_c = \prod_{l' \in \Gamma/\{l, \tilde{l}\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{s}' \in D(l')} \boxed{C_c|_{\mathsf{s}'}}^{\Gamma} \cdot t_{\mathsf{s}'} \cdot M_{\mathsf{s}'} \right\rangle_{l'}$$

The first transition, $\lambda = \underline{\mathbf{k.C.l}}$ consumed the first assignment of location and assigned the location of role $\mathtt{C}$ to $\underline{\mathbf{k.C.l}}$ in the state of the starter t_{p}.

Let us suppose, without loss of generality, that $\mathtt{C} = \mathtt{A}$, then we have

$$P' = \left(\begin{array}{l} \bigodot\limits_{\underline{\mathtt{I}} \in \{\tilde{\mathtt{B}}\}} \ \underline{\mathbf{k.I.l}} = l_{\mathtt{I}}\ ; \\ \bigodot\limits_{\underline{\mathtt{I}} \in \{\tilde{\mathtt{B}}\}} \Big(\mathsf{cq}(\underline{\mathbf{k.I.A}}); ?@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}}); \mathit{sync}(\underline{\mathbf{k}})\ \mathtt{from}\ \underline{\mathbf{k.I.A}} \Big); \\ \bigodot\limits_{\underline{\mathtt{I}} \in \{\tilde{\mathtt{B}}\}} \ \mathit{start}@\underline{\mathbf{k.I.l}}(\underline{\mathbf{k}})\ \mathtt{to}\ \underline{\mathbf{k.A.I}}; \end{array} \right) ; \boxed{C_r}^{\Gamma} \cdot t_{\mathsf{p}} \triangleleft (\,\underline{\mathbf{k.A.l}}, l\,) \cdot M_{\mathsf{p}}$$

and $\boxed{D, C}^{\Gamma} \xrightarrow{\underline{\mathbf{k.A.l}}} S'$ where

$$S' = \left\langle \boxed{C_c|_l}^{\Gamma}, P' \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i, R_{l_i} \rangle_{l_i} \mid S_c$$

Since in its reduction D, C renames the new session with a fresh name, we first rename session k in P and the service processes Q_i to k', which is fresh. We take

$$P'' = P'[k'/k] = \left(\begin{array}{l} \bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \underline{\mathbf{k'.I.l}} = l_{\mathtt{I}} \,; \\ \bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \Big(\mathsf{cq}(\underline{\mathbf{k'.I.A}}); ?@\underline{\mathbf{k'.I.l}}(\underline{\mathbf{k'}}); ?@\underline{\mathbf{k'.I.l}}(\underline{\mathbf{k'}}); \Big); \\ \bigodot_{\underline{\mathbf{I}} \in \{\tilde{\mathtt{B}}\}} \mathit{sync}(\underline{\mathbf{k'}}) \ \mathtt{from}\ \underline{\mathbf{k'.I.A}} \end{array} \right) ; \boxed{C_r}^{\Gamma} [\underline{\mathbf{k'}}/\underline{\mathbf{k}}] \cdot t''_{\mathsf{p}} \cdot M_{\mathsf{p}}$$

where, let $t'_{\mathsf{p}} = t_{\mathsf{p}} \triangleleft (\, \underline{\mathbf{k.A.l}}, l \,)$, $t''_{\mathsf{p}} = t'_{\mathsf{p}} \triangleleft (\, \underline{\mathbf{k'}}, \underline{\mathbf{k}}(t'_{\mathsf{p}}) \,) \triangleleft (\, \underline{\mathbf{k}}, t_{\perp} \,)$.

By Lemma 28 we have $P' \sim P''$. We take

$$S_0^* = \left\langle \boxed{C_c|_l}^{\Gamma}, P'' \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i[\underline{\mathbf{k'}}/\underline{\mathbf{k}}], R_{l_i} \rangle_{l_i} \mid S_c$$

and by Corollary 30 we have $S_0^* \sim S'$.

Now we can proceed with the rest of the transitions of the start procedure, as defined at the beginning of the proof. Finally we have

$$S'' = \left\langle \boxed{C_c|_l}^{\Gamma} \mid P''' \mid R \right\rangle_l \mid \prod_{i=1}^{n} \langle Q_i[\underline{\mathbf{k'}}/\underline{\mathbf{k}}], Q'_i \mid R_{l_i} \rangle_{l_i} \mid S_c$$

where $P''' = \boxed{C_r}^{\Gamma} [\underline{\mathbf{k'}}/\underline{\mathbf{k}}] \cdot t'_p \cdot M'_p$ and $Q'_i = \boxed{C_{\mathsf{q}_i}}^{\Gamma} [\underline{\mathbf{k'}}/\underline{\mathbf{k}}] \cdot t_{k'} \cdot M_{\mathsf{q}_i}$

From the transitions presented above we know that there exists $t'_{k'}$ such that $t'_{\mathsf{p}} = t_{\mathsf{p}} \triangleleft (\, \underline{\mathbf{k'}}, t'_{k'} \,)$, where $t'_{k'}$ is a session descriptor for session k' (i.e., it contains all the locations and correlations keys used by the processes in session k').

We proceed by proving that D, C can mimic $\boxed{D, C}^{\Gamma}$.

We can apply Rules $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$ and $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$ and lastly Rule $\lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$ such that

$$\frac{\begin{array}{c} i \in \{1, \ldots, n\} \qquad \#k' \qquad \{\widetilde{l.\mathsf{B}}\} = \biguplus_i \{l_i.\mathsf{B}_i\} \qquad \#\tilde{\mathsf{r}} \qquad \{\tilde{\mathsf{r}}\} = \bigcup_i \{\tilde{\mathsf{r}}_i\} \\ \mathsf{p} \in D(l) \qquad \delta = \textbf{start}\ k' :\ l.\mathsf{p}[\mathtt{A}], l_1.\mathsf{r}_1[\mathsf{B}_1], \ldots, l_n.\mathsf{r}_n[\mathsf{B}_n] \qquad D, \delta \blacktriangleright D' \end{array}}{D, C \quad \rightarrow \quad D',\ C_r[k'/k] \mid \prod_{i=1}^{n} \big(\, C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i] \,\big) \mid C_a} \ \lfloor^{\mathrm{C}}|_{\mathrm{PSTART}}\rceil$$

and

$$\begin{array}{l} D, C \mid C_c \quad \rightarrow \\ \quad D', C_r[k'/k] \mid \prod_i \big(\, C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i] \,\big) \mid \prod_{i=1}^{n} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathsf{B}_i]; C_{\mathsf{q}_i} \mid C_c \end{array}$$

thus $C' = C_r[k'/k] \mid \prod_{i=1}^{n} \big(C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i] \big) \mid \prod_{i=1}^{n} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; C_{\mathsf{q}_i} \mid C_c$.

From the hypothesis we know that $\Gamma \vdash D, C$ and therefore that $\Gamma = \Gamma_1, \tilde{l} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle$. We can find $\Gamma' = \Gamma, \mathsf{init}(k', (\mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]}), G)$ and $\Gamma' \vdash D', C'$.

Finally, we need to prove that $S_1^* = \boxed{D', C'}^{\Gamma'}$.

From Definition 23 we have

$$\boxed{D', C'}^{\Gamma'} = \Big\langle \boxed{C_c|_l}^{\Gamma'}, P'' \mid R' \Big\rangle_l \mid \prod_{i=1}^{n} \langle Q_i'', Q_i^* \mid R'_{l_i} \rangle_{l_i} \mid S'_c$$

In the following, we use the shortcuts $t^*_{\mathsf{s}} = D'(\mathsf{s}).\mathsf{st}$ and $M^*_{\mathsf{s}} = D'(\mathsf{s}).\mathsf{que}$ for s process in D'.

- $P'' = \boxed{C'_r[k'/k]}^{\Gamma'} \cdot t^*_{\mathsf{p}} \cdot M^*_{\mathsf{p}}$
- $R' = \prod_{\mathsf{p}' \in D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{p}'}}^{\Gamma'} \cdot t^*_{\mathsf{p}'} \cdot M^*_{\mathsf{p}'}$
- $Q''_i = \mathsf{accept}(k, \mathtt{B}_i, G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle); \boxed{C_{\mathsf{q}_i}}^{\Gamma'}$
- $Q^*_i = \boxed{C_{\mathsf{q}_i}[k'/k][\mathsf{r}_i/\mathsf{q}_i]}^{\Gamma'} \cdot t^*_{qi} \cdot M^*_{qi}$
- $R'_{l_i} = \prod_{\mathsf{s} \in D(l_i)} \boxed{C_c|_{\mathsf{s}}}^{\Gamma'} \cdot t^*_{\mathsf{s}} \cdot M^*_{\mathsf{s}}$
- $S'_c = \prod_{l' \in \Gamma/\{l,\tilde{l}\}} \Big\langle \boxed{C_c|_{l'}}^{\Gamma'}, \prod_{\mathsf{s}' \in D(l')} \boxed{C_c|_{\mathsf{s}'}}^{\Gamma'} \cdot t^*_{\mathsf{s}'} \cdot M^*_{\mathsf{s}'} \Big\rangle_{l'}$

From Rule $\lfloor^{\mathrm{D}}|_{\mathrm{START}}\rceil$ we know that

$$\underline{\mathbf{k'}}(t^*_{\mathsf{p}}) = \underline{\mathbf{k'}}(t^*_{\mathsf{q}_1}) = \ldots = \underline{\mathbf{k'}}(t^*_{\mathsf{q}_n}) = t_{k'}$$

for some $t_{k'}$ session descriptor of session k'.

We prove the case by taking $t_{k'} = t'_{k'}$, $t'_{k'}$ obtained from the derivation of $\boxed{D, C}^{\Gamma}$ and $M^*_{\mathsf{p}} = M'_{\mathsf{p}}$ and $M^*_{\mathsf{q}_i} = M_{\mathsf{q}_i}$, $i \in \{1, \ldots, n\}$.

Case $1 < |\{\tilde{\lambda}\}| < m - 1$
The case follows the same structure of the previous case. We rename k to k' on p and all the newly created service processes. Then we let the system complete all the transitions and prove that the reductum corresponds to the compilation of D', C'.

Case $|\{\tilde{\lambda}\}| = m$
Since $|\{\tilde{\lambda}\}| = m$ then $S = S'$ where S' has terminated all the transitions to start the session. Here we only have to rename k to k' for all the involved processes and prove that $S' = \boxed{D', C'}^{\Gamma'}$.

□

We now proceed to prove the *(Soundness)* of Theorem 8. In the following we use the shortcut

$$C_{start} = \textbf{req}\ k : \mathsf{p}[\mathtt{A}] \Leftrightarrow l_1.[\mathtt{B}_1], \dots, l_n.[\mathtt{B}_n]; C_r \mid \prod_{i=1}^{n} \textbf{acc}\ k : l_i.\mathsf{q}_i[\mathtt{B}_i]; C_{\mathsf{q}_i}$$

Proof (Soundness). We proceed by induction on the cardinality of $\tilde{\lambda}$.

Case $|\{\tilde{\lambda}\}| = 0$

Trivial, $\boxed{D, C}^{\Gamma} = S = \boxed{D', C'}^{\Gamma'}$, $D, C = D', C'$, and $\Gamma \vdash D', C'$.

Case $|\{\tilde{\lambda}\}| = 1$

We unfold the case on the structure of D, C. We do not consider the inapplicable cases for C containing (com) or $(start)$ terms, which cannot be present in endpoint choreographies.

Case $C = k : \mathsf{p}[\mathtt{A}].e \rightarrow \mathtt{B}.o; C_{\mathsf{p}} \mid C_c$

We proceed analysing the last Rule applied in the derivation of $\boxed{D, C}^{\Gamma}$

Case $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$

We have $\tilde{\lambda} = \lambda = o@\underline{\mathbf{k.A.B}}$ where $o \notin \{start, sync\}$. Indeed operations $start$ and $sync$ are reserved for the starting of new sessions and cannot appear as first action of the compilation of $\boxed{D, C}^{\Gamma}$. Therefore we can exclude the case that $\boxed{D, C}^{\Gamma}$ is starting a new session.
Since $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$ applies we can infer that, let $\mathsf{p}@l \in \Gamma$, then $\mathsf{q}@l \in \Gamma$ such that

$$\boxed{D, C}^{\Gamma} \equiv \left\langle \boxed{C_c|_l}^{\Gamma}, P \mid Q \mid R \right\rangle_l \mid S_c$$

where

- $P = o@\underline{\mathbf{k.B.l}}\ \texttt{to}\ \underline{\mathbf{k.A.B}}; \boxed{C_{\mathsf{p}}}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}}$
- $Q = \boxed{C_c|_{\mathsf{q}}}^{\Gamma} \cdot t_{\mathsf{q}} \cdot M_{\mathsf{q}}$
- $R = \prod_{\mathsf{r}\,\in\, D(l)/\{\mathsf{p},\mathsf{q}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}$
- $S_c = \prod_{l' \in \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod_{\mathsf{s}\,\in\, D(l')} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}} \right\rangle_{l'}$

$\boxed{D, C}^{\Gamma}$ reduces, applying as last Rule $\lfloor^{\mathrm{DCC}}|_{\textsc{InSend}}\rceil$, let $t_c = \mathsf{eval}(\underline{\mathbf{k.A.B}}, t_{\mathsf{p}})$, $t_m = \mathsf{eval}(e, t_{\mathsf{p}})$, and $M_{\mathsf{q}}(t_c) = \tilde{m}$

$$\boxed{D, C}^{\Gamma} \quad \xrightarrow{o@\underline{\mathbf{k.A.B}}} \quad S$$

where

$$S = \left\langle \boxed{C_c|_{l}}^{\Gamma}, \boxed{C_{\mathsf{p}}}^{\Gamma} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} \mid \boxed{C_c|_{\mathsf{q}}}^{\Gamma} \cdot t_{\mathsf{q}} \cdot M_{\mathsf{q}}[t_c \mapsto \tilde{m} :: (o, t_m)] \mid R \right\rangle_{l} \mid S_c$$

D, C can mimic $\boxed{D, C}^{\Gamma}$ with Rules $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{C}}|_{\textsc{Par}}\rceil$, and $\lfloor^{\mathrm{C}}|_{\textsc{Send}}\rceil$ for which

$$D, C \quad \rightarrow \quad D', C_{\mathsf{p}} \mid C_c$$

where $D' = D[\mathsf{q} \mapsto (t_{\mathsf{q}}, M_{\mathsf{q}}[t_c \mapsto \tilde{m} :: (o, t_m)])]$.
Since from the premises $\Gamma \vdash D, C$ then $\Gamma = \Gamma_1, k[\mathtt{A}] : !\mathtt{B}.o(U); T, b[k]^{\mathtt{A}}_{\mathtt{B}} : T'$ and we can find $\Gamma' = \Gamma_1, k[\mathtt{A}] : T, b[k]^{\mathtt{A}}_{\mathtt{B}} : T'; ?\mathtt{A}.o(U)$ such that $\Gamma' \vdash D, C$.
From Lemma 27 we have that $\boxed{D', C'}^{\Gamma'} = \boxed{D', C'}^{\Gamma}$. Applying Definition 23 to $\boxed{D', C'}^{\Gamma}$, we have $\boxed{D', C'}^{\Gamma} = S$ and therefore $S = \boxed{D', C'}^{\Gamma'}$.

Case $\lfloor^{\mathrm{DCC}}|_{\textsc{Send}}\rceil$
Similar to the proof of case $\lfloor^{\mathrm{DCC}}|_{\textsc{InSend}}\rceil$.

Case $\lfloor^{\mathrm{DCC}}|_{\textsc{Choice}}\rceil$
The case unfolds following the proof of case $C = k : \mathtt{A} \texttt{->} \mathsf{q}[\mathsf{B}].\{o_i(x_i); C_i\}_i; C_{\mathsf{q}} \mid C_c$, subcase $\lfloor^{\mathrm{DCC}}|_{\textsc{Choice}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\mathrm{DCC}}|_{\textsc{Cond}}\rceil$
The case unfolds following the proof of case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}; C_{\mathsf{q}} \mid C_c$, subcase $\lfloor^{\mathrm{DCC}}|_{\textsc{Cond}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\mathrm{DCC}}|_{\textsc{Assign}}\rceil$
The case unfolds following the proof of case $C = C_{start} \mid C_c$, subcase $\lfloor^{\mathrm{DCC}}|_{\textsc{Assign}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\mathrm{DCC}}|_{\textsc{Recv}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\textsc{InStart}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\textsc{Start}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\mathrm{CQ}}\rceil$
None of the Rules $\lfloor^{\mathrm{DCC}}|_{\textsc{Recv}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\textsc{InStart}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\textsc{Start}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{CQ}}\rceil$ apply as

$\boxed{D, C}^{\Gamma}$ makes only one transition whilst all the listed Rules can appear only in a sequence of interactions greater than one.

Case $C = k : \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \mid C_c$
We proceed analysing the last Rule applied in the derivation of $\boxed{D, C}^{\Gamma}$

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{INSEND}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\mathrm{SEND}}\rceil$
The case unfolds following the proof of case $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o; C_{\mathsf{q}} \mid C_c$, respectively subcases $\lfloor^{\mathrm{DCC}}|_{\mathrm{INSEND}}\rceil$ and $\lfloor^{\mathrm{DCC}}|_{\mathrm{SEND}}\rceil$. The thesis follows by applying the induction hypothesis.

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{CHOICE}}\rceil$
We have $\tilde{\lambda} = \lambda = o_j \texttt{ from } \underline{\mathbf{k.A.B}}$ where $o_j \notin \{start, sync\}$, following the same reasoning of case $C = k : \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B}.o; C_{\mathsf{p}} \mid C_c$ subcase $\lfloor^{\mathrm{DCC}}|_{\mathrm{SEND}}\rceil$.
Let $\mathsf{q}@l \in \Gamma$, from Definition 23 we have

$$\boxed{D, C}^{\Gamma} \equiv \left\langle \boxed{C_c|_l}^{\Gamma}, Q \mid R \right\rangle_l \mid S_c$$

where

- $Q = \sum_{i \in I} [o_i(x_i) \texttt{ from } \underline{\mathbf{k.A.B}}] \left\{\boxed{C_i}^{\Gamma}\right\} \cdot t_{\mathsf{q}} \cdot M_{\mathsf{q}}$
- $R = \prod\limits_{\mathsf{r}\, \in\, D(l)/\{\mathsf{q}\}} \boxed{C_c|_{\mathsf{r}}}^{\Gamma} \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}$
- $S_c = \prod\limits_{l' \in \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^{\Gamma}, \prod\limits_{\mathsf{s}\, \in\, D(l')} \boxed{C_c|_{\mathsf{s}}}^{\Gamma} \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}} \right\rangle_{l'}$

and can apply Rules $\lfloor^{\mathrm{DCC}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{SPAR}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{PPAR}}\rceil$ and lastly $\lfloor^{\mathrm{DCC}}|_{\mathrm{CHOICE}}\rceil$ such that, let $t_c = \mathsf{eval}(\underline{\mathbf{k.A.B}}, t_{\mathsf{p}})$, $t_m = \mathsf{eval}(e, t_{\mathsf{p}})$, and $M_{\mathsf{q}}(t_c) = (o_j, t_m) :: \tilde{m}$

$$\boxed{D, C}^{\Gamma} \xrightarrow{o_j \texttt{ from } \underline{\mathbf{k.A.B}}} S$$

where

$$S = \left\langle \boxed{C_c|_l}^{\Gamma}, \boxed{C_j}^{\Gamma} \cdot t_{\mathsf{q}} \lhd (\, x_j, t_m\,) \cdot M_{\mathsf{p}}[t_c \mapsto \tilde{m}] \mid R \right\rangle_l \mid S_c$$

D, C can mimic $\boxed{D, C}^{\Gamma}$ with Rules $\lfloor^{\mathrm{C}}|_{\mathrm{EQ}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{PAR}}\rceil$, and $\lfloor^{\mathrm{C}}|_{\mathrm{RECV}}\rceil$ for which

$$D, C \quad \rightarrow \quad D', C_{\mathsf{p}} \mid C_c$$

where $D' = D[\mathsf{q} \mapsto (t_{\mathsf{q}} \lhd (\, x_j, t_m\,), M_{\mathsf{q}}[t_c \mapsto \tilde{m}])]$.
Since from the premises $\Gamma \vdash D, C$ then
$\Gamma = \Gamma_1, k[\mathtt{A}] : ?\mathtt{A}.\{o_i(U_i); T_i\}_{i \in I}, b[k]^{\mathtt{A}}_{\mathtt{B}} : ?\mathtt{A}.o_j(U_j); T'$ and we can find
$\Gamma' = \Gamma_1, k[\mathtt{A}] : T_j, b[k]^{\mathtt{A}}_{\mathtt{B}} : T'$ such that $\Gamma' \vdash D, C$.
From Lemma 27 we have that $\boxed{D', C'}^{\Gamma'} = \boxed{D', C'}^{\Gamma}$. Applying Definition 23 to $\boxed{D', C'}^{\Gamma}$, we have $\boxed{D', C'}^{\Gamma} = S$ and therefore $S = \boxed{D', C'}^{\Gamma'}$.

Case $\lfloor^{\text{DCC}}|_{\text{COND}}\rceil$
The case unfolds following the proof of case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}; C_{\mathsf{q}} \mid C_c$, subcase $\lfloor^{\text{DCC}}|_{\text{COND}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\text{DCC}}|_{\text{ASSIGN}}\rceil$
The case unfolds following the proof of case $C = C_{start} \mid C_c$, subcase $\lfloor^{\text{DCC}}|_{\text{ASSIGN}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{INSTART}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{START}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$
None of the Rules $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{INSTART}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{START}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$ apply as $\boxed{D, C}^{\Gamma}$ makes only one transition whilst all the listed Rules can appear only in a sequence of interactions greater than one.

Case $C = C_{start} \mid C_c$
We proceed analysing the last Rule applied in the derivation of $\boxed{D, C}^{\Gamma}$

Case $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{SEND}}\rceil$
The case unfolds following the proof of case $C = k : \mathsf{p}[\mathsf{A}].e \text{ -> } \mathsf{B}.o; C_{\mathsf{q}} \mid C_c$, respectively subcases $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$ and $\lfloor^{\text{DCC}}|_{\text{SEND}}\rceil$. The thesis follows by applying the induction hypothesis.

Case $\lfloor^{\text{DCC}}|_{\text{CHOICE}}\rceil$
The case unfolds following the proof of case
$C = k : \mathsf{A} \text{ -> } \mathsf{q}[\mathsf{B}].\{o_i(x_i); C_i\}_{i\in I}; C_{\mathsf{q}} \mid C_c$, respectively subcases $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$ and $\lfloor^{\text{DCC}}|_{\text{SEND}}\rceil$. The thesis follows by applying the induction hypothesis.

Case $\lfloor^{\text{DCC}}|_{\text{COND}}\rceil$
The case unfolds following the proof of case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\}; C_{\mathsf{q}} \mid C_c$, subcase $\lfloor^{\text{DCC}}|_{\text{COND}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\text{DCC}}|_{\text{ASSIGN}}\rceil$
Let $\mathsf{p}@l \in \Gamma$, we have two subcases whether $l \in \{l_1, \dots, l_n\}$, i.e., whether one of the service processes is at the same location of p.

Case $l \notin \{l_1, \dots, l_n\}$
Let $\mathsf{C} \in \{\mathsf{A}, \tilde{\mathsf{B}}\}$, we have two subcases whether $\tilde{\lambda} = \lambda = \underline{\textbf{k.C.l}}$ or $\tilde{\lambda} = \lambda = \underline{\textbf{k}''\textbf{.C.l}}$, i.e., whether we are stating the session on k or we are starting another session k''.

Case $\lambda = \underline{\textbf{k.C.l}}$
In this case $\boxed{D, C}^{\Gamma}$ is starting a new session on k. The case is proven applying Lemma 33.

Case $\lambda \neq \underline{\textbf{k}'\textbf{.C.l}}$
In this case we are starting a session on $k'' \neq k$. The case unfolds following the proof of case $C = C_{start}|C_c$, subcase $\lfloor^{\text{C}}|_{\text{ASSIGN}}\rceil$, subcase $\lambda = \underline{\textbf{k.C.l}}$. The thesis follows by applying the induction hypothesis.

Case $l \in \{l_1, \ldots, l_n\}$
The case follows the same structure of the case $l \notin \{\tilde{l}\}$ although the service located at l has $\boxed{C_a|_l}^\Gamma$ as start behaviour and $\boxed{D, C}^\Gamma$ applies Rule $\lfloor^{\mathrm{DCC}}|_{\mathrm{InStart}}\rceil$ in place of the $\lfloor^{\mathrm{DCC}}|_{\mathrm{Start}}\rceil$ for starting the DCC process located at l.

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{Recv}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\mathrm{InStart}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\mathrm{Start}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\mathrm{CQ}}\rceil$
None of the Rules $\lfloor^{\mathrm{DCC}}|_{\mathrm{Recv}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{InStart}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{Start}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{CQ}}\rceil$ apply as $\boxed{D, C}^\Gamma$ makes only one transition whilst all the listed Rules can appear only in a sequence of interactions greater than one.

Case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \mid C_c$
We proceed analysing the last Rule applied in the derivation of $\boxed{D, C}^\Gamma$.

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{InSend}}\rceil$ or $\lfloor^{\mathrm{DCC}}|_{\mathrm{Send}}\rceil$
The case unfolds following the proof of case $C = k : \mathsf{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o; C_{\mathsf{q}} \mid C_c$, respectively subcases $\lfloor^{\mathrm{DCC}}|_{\mathrm{InSend}}\rceil$ and $\lfloor^{\mathrm{DCC}}|_{\mathrm{Send}}\rceil$. The thesis follows by applying the induction hypothesis.

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{Choice}}\rceil$
The case unfolds following the proof of case
$C = k : \mathtt{A} \text{ -> } \mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_i; C_{\mathsf{q}} \mid C_c$, subcase $\lfloor^{\mathrm{DCC}}|_{\mathrm{Choice}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\mathrm{DCC}}|_{\mathrm{Cond}}\rceil$
We have $\tilde{\lambda} = \lambda = \tau$. Let $\mathsf{p}@l \in \Gamma$. From Definition 23 we have

$$\boxed{D, C}^\Gamma \equiv \left\langle \boxed{C_c|_l}^\Gamma, P \mid R \right\rangle_l \mid S_c$$

where

- $P = \mathsf{if}\ \mathsf{p}.e\ \{\boxed{C_1}^\Gamma\}\ \mathsf{else}\ \{\boxed{C_2}^\Gamma\} \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}}$
- $R = \prod_{\mathsf{r} \in D(l)/\{\mathsf{p}\}} \boxed{C_c|_{\mathsf{r}}}^\Gamma \cdot t_{\mathsf{r}} \cdot M_{\mathsf{r}}$
- $S_c = \prod_{l' \in \Gamma/\{l\}} \left\langle \boxed{C_c|_{l'}}^\Gamma, \prod_{\mathsf{s} \in D(l')} \boxed{C_c|_{\mathsf{s}}}^\Gamma \cdot t_{\mathsf{s}} \cdot M_{\mathsf{s}} \right\rangle_{l'}$

The case unfolds into two cases, on whether $\mathsf{eval}(e, D.\mathsf{st}(\mathsf{p}) = \mathsf{true}$. Here we proceed with the positive case. The other case follows the same structure.
We proceed considering that $\mathsf{eval}(e, D.\mathsf{st}(\mathsf{p}) = \mathsf{true}$. $\boxed{D, C}^\Gamma$ reduces with Rules $\lfloor^{\mathrm{DCC}}|_{\mathrm{Eq}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{SPar}}\rceil$, $\lfloor^{\mathrm{DCC}}|_{\mathrm{Par}}\rceil$, and $\lfloor^{\mathrm{DCC}}|_{\mathrm{Cond}}\rceil$ such that

$$\boxed{D, C}^\Gamma \quad \to \quad \left\langle \boxed{C_c|_l}^\Gamma, \boxed{C_c|_{\mathsf{p}}}^\Gamma \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} \mid R \right\rangle_l \mid S_c$$

where $S = \left\langle \boxed{C_c|_l}^\Gamma, \boxed{C_c|_l}^\Gamma \cdot t_{\mathsf{p}} \cdot M_{\mathsf{p}} \mid R \right\rangle_l \mid S_c$. D, C can mimic $\boxed{D, C}^\Gamma$ with Rules $\lfloor^{\mathrm{C}}|_{\mathrm{Eq}}\rceil$, $\lfloor^{\mathrm{C}}|_{\mathrm{Par}}\rceil$, and $\lfloor^{\mathrm{C}}|_{\mathrm{Cond}}\rceil$ such that

$$D, C \quad \to \quad D, C_1 \mid C_c$$

We choose $\Gamma' = \Gamma$ for which it holds that $\Gamma \vdash D, C_1 \mid C_c$.
Finally, $\boxed{D, C_1 \mid C_c}^{\Gamma} = S$ by Definition 23.

Case $\lfloor^{\text{DCC}}|_{\text{ASSIGN}}\rceil$
The case unfolds following the proof of case $C = C_{start} \mid C_c$, subcase $\lfloor^{\text{DCC}}|_{\text{ASSIGN}}\rceil$, and the thesis follows by applying the induction hypothesis.

Case $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{INSTART}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{START}}\rceil$ or $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$
None of the Rules $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{INSTART}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{START}}\rceil$, $\lfloor^{\text{DCC}}|_{\text{CQ}}\rceil$ apply as $\boxed{D, C}^{\Gamma}$ makes only one transition whilst all the listed Rules can appear only in a sequence of interactions greater than one.

Case $C = \mathsf{def}\ X = C'\ \mathsf{in}\ C_{\mathsf{p}} \mid C_c$ or $C = C_1 \mid C_2$
In either cases the proof unfolds with the same structure of the previous cases and the thesis follows by applying the induction hypothesis.

Case $C = \mathbf{0}$
Impossible since $|\{\tilde{\lambda}\}| = 1$.

Case $|\{\tilde{\lambda}\}| > 1$
We unfold the case on the structure of D, C. Like in the previous case, we do not consider the inapplicable cases for C containing (com) or $(start)$ terms, which cannot be present in endpoint choreographies.

For each AC configuration, we analyse the subcases on the structure of $\tilde{\lambda} = \lambda', \tilde{\lambda}'$. Note that $\tilde{\lambda}$ contains all the annotation labels of execution since the compilation of $\boxed{D, C}^{\Gamma}$. Hence, we can exclude that the labels of the kind $\mathsf{cq}(x)$ and $?(x)$ as they can only appear later in $\tilde{\lambda}'$ after a sequence of assignments.

Case $C = k : \mathsf{p}[\mathtt{A}].e$ `->` $\mathtt{B}.o; C_{\mathsf{p}} \mid C_c$

Case $\lambda' = o@x$
We can follow the same structure of the proof for case $|\{\lambda\}| = 1$, $C = k : \mathsf{p}[\mathtt{A}].e$ `->` $\mathtt{B}.o; C_{\mathsf{p}} \mid C_c$, Rule $\lfloor^{\text{DCC}}|_{\text{INSEND}}\rceil$ (or $\lfloor^{\text{DCC}}|_{\text{SEND}}\rceil$, depending on whether $\{\mathsf{p}@l, \mathsf{q}[\mathtt{B}]\colon k, \mathsf{q}@l\} \subseteq \Gamma$ or not), to prove that $D, C \to D', C'$, $\boxed{D, C}^{\Gamma} \xrightarrow{\lambda'} S$ and that there exists a Γ' such that $\Gamma' \vdash D', C'$ and $\boxed{D, C}^{\Gamma'} = S$. Then we can prove the thesis applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$ and D', C'.

Case $\lambda' = o$ `from` x
The case unfolds following the proof of case $|\{\tilde{\lambda}\}| > 1$,
$C = k : \mathtt{A}$ `->` $\mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \mid C_c$, $\lambda' = o$ `from` x and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $\lambda' = x$
The case unfolds following the proof of case $|\{\tilde{\lambda}\}| > 1$, $C = C_{start} \mid C_c$, $\lambda' = x$. Let $x =$ **k.C.l**, the thesis follows by applying Lemma 32, Lemma 33 and the induction hypothesis on the remaining transitions in $\tilde{\lambda} / \left.\tilde{\lambda}\right|_k$.
Case $\lambda' = \tau$
The case unfolds following the proof of case $|\{\tilde{\lambda}\}| > 1$,
$C = \mathtt{if}\ \mathtt{p}.e\ \{C_1\}\ \mathtt{else}\ \{C_2\} \mid C_c$, $\lambda' = \tau$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $C = k : \mathtt{A} \text{ -> } \mathtt{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \mid C_c$

Case $\lambda' = o@x$
The case unfolds following the proof of case $|\{\tilde{\lambda}\}| > 1$,
$C = k : \mathtt{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o; C_{\mathtt{p}} \mid C_c$, $\lambda' = o@x$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.
Case $\lambda' = o\ \mathtt{from}\ x$
We follow the same structure of the proof for case $|\{\lambda\}| = 1$,
$C = k : \mathtt{A} \text{ -> } \mathtt{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \mid C_c$, Rule $\lfloor^{\text{DCC}}|_{\text{CHOICE}}\rceil$ to prove that $D, C \to D', C'$, $\boxed{D, C}^{\Gamma} \xrightarrow{\lambda'} S$ and that there exists a Γ' such that $\Gamma' \vdash D', C'$ and $\boxed{D, C}^{\Gamma'} = S$. Then we can prove the thesis applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$. Note that we can exclude that λ' has been generated applying Rule $\lfloor^{\text{DCC}}|_{\text{RECV}}\rceil$ since such occurrence cannot happen right after the compilation of D, C.
Case $\lambda' = x$
The case unfolds following the proof of case $|\{\tilde{\lambda}\}| > 1$, $C = C_{start} \mid C_c$, $\lambda' = x$. Let $x =$ **k.C.l**, the thesis follows by applying Lemma 32, Lemma 33 and the induction hypothesis on the remaining transitions in $\tilde{\lambda} / \left.\tilde{\lambda}\right|_k$.
Case $\lambda' = \tau$
The case unfolds following the proof of case
$|\{\tilde{\lambda}\}| > 1$, $C = \mathtt{if}\ \mathtt{p}.e\ \{C_1\}\ \mathtt{else}\ \{C_2\} \mid C_c$, $\lambda' = \tau$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $C = C_{start} \mid C_c$

Case $\lambda' = o@x$
The case unfolds following following the proof of case $|\{\tilde{\lambda}\}| > 1$, $C = k : \mathtt{p}[\mathtt{A}].e \text{ -> } \mathtt{B}.o; C_{\mathtt{p}} \mid C_c$, $\lambda' = o@x$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.
Case $\lambda' = o\ \mathtt{from}\ x$
We follow the proof of case $|\{\tilde{\lambda}\}| > 1$, $C = k : \mathtt{A} \text{ -> } \mathtt{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \mid C_c$,

$\lambda' = o$ `from` x and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $\lambda' = x$

Since $\lambda' = x$, we know that we are at the beginning of the start of a new session. We have two subcases on whether $x = \underline{\textbf{k.C.l}}$ or $x = \underline{\textbf{k''.C.l}}$. The latter case is proven following the same structure of this case for C that contains the start of a session on k''. Below we develop the first case.

First we extract all the transitions in $\tilde{\lambda}$ regarding the start of session k. Let $\tilde{\lambda}_k = \tilde{\lambda}\Big|_k$. From Lemma 33 we know that, given the set of transitions $\tilde{\lambda}_k$, we can complete the start at DCC level such that $\boxed{D, C}^{\Gamma} \xrightarrow{\tilde{\lambda}_k, \tilde{\lambda}_{m-k}} S'$ where $\tilde{\lambda}_{m-k}$ are the remaining transitions to complete the start. We remind that, since at AC level we replace k with a fresh session name k', we follow the same principle at DCC level applying the renaming $[k'/k]$ — k' fresh — on all the involved processes. Then we can find $D, C \to D', C'$ with Rule $\lfloor^{\mathsf{C}}|_{\mathrm{PSTART}}\rceil$ such that, for some Γ', $\Gamma' \vdash D', C'$ and $\boxed{D', C'}^{\Gamma'} = S'$. Finally, we prove the thesis applying Lemma 32 and the induction hypothesis on the remaining and properly renamed transitions $\left(\tilde{\lambda}/\ \tilde{\lambda}\Big|_k\right)[k'/k]$.

Case $\lambda' = \tau$

The case follows the proof of case $|\{\tilde{\lambda}\}| > 1, C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \mid C_c$, $\lambda' = \tau$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \mid C_c$

Case $\lambda' = o@x$

The case follows the proof of case $|\{\tilde{\lambda}\}| > 1, C = k : \mathsf{p}[\mathtt{A}].e$ -> $\mathtt{B}.o; C_{\mathsf{p}} \mid C_c$, $\lambda' = o@x$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $\lambda' = o$ `from` x

The case unfolds following the proof of case
$|\{\tilde{\lambda}\}| > 1, C = k : \mathtt{A}$ -> $\mathsf{q}[\mathtt{B}].\{o_i(x_i); C_i\}_{i \in I} \mid C_c$, $\lambda' = o$ `from` x and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $\lambda' = x$

The case unfolds following the proof of case $|\{\tilde{\lambda}\}| > 1$, $C = C_{start} \mid C_c$, $\lambda' = x$. Let $x = \underline{\textbf{k.C.l}}$, the thesis follows by applying Lemma 32, Lemma 33, and the induction hypothesis on the remaining transitions in $\tilde{\lambda}/\ \tilde{\lambda}\Big|_k$.

Case $\lambda' = \tau$

The case follows the same proof of case $|\{\tilde{\lambda}\}| = 1, C = \mathsf{if}\ \mathsf{p}.e\ \{C_1\}\ \mathsf{else}\ \{C_2\} \mid C_c$,

$\lambda' = \tau$ and the thesis follows by applying the induction hypothesis on the remaining transitions in $\tilde{\lambda}'$.

Case $C = \mathsf{def}\ X = C'\ \mathsf{in}\ C_{\mathsf{p}} \mid C_c$ or $C = C_1 \mid C_2$
In either cases the proof unfolds with the same structure of the previous cases and the thesis follows by applying the induction hypothesis.

Case $C = \mathbf{0}$
Impossible since $|\{\tilde{\lambda}\}| > 1$.

□