Statically Safe Distributed Programming

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
Genehmigte Dissertation von Malte Christian Viering aus Volkmarsen
Tag der Einreichung: 5.01.2022, Tag der Prüfung: 16.02.2022

1. Gutachten: Prof. Dr. Dr. h.c. Mira Mezini
2. Gutachten: Prof. Dr. Patrick Eugster
3. Gutachten: Dr. Raymond Hu
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Distributed Systems Programming Group

Statically Safe Distributed Programming

Accepted doctoral thesis by Malte Christian Viering

1. Review: Prof. Dr. Dr. h.c. Mira Mezini
2. Review: Prof. Dr. Patrick Eugster
3. Review: Dr. Raymond Hu

Date of submission: 5.01.2022
Date of thesis defense: 16.02.2022

Darmstadt

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-220222
URL: http://tuprints.ulb.tu-darmstadt.de/22022

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

To my family and friends, I would not be here without you. Thanks!

I am deeply thankful and grateful to my supervisor Patrick Eugster for all the support, guidance and opportunities over the years. Patrick, you deserve major credits for me reaching this goal. I further want to express my gratitude towards my co-supervisor Raymond Hu. Our technical and non-technical discussions over the years were unbelievably valuable to me and kept me going.
I want to thank all my colleagues and friends I have worked with over the years. All our great work is only possible because of our amazing collaborations.
Finally, I want to thank my colleagues and friends in the Distributed System Programming group in Darmstadt. I cannot imagine the last years without your support and our wonderful discussions and long evenings together.

Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 5.01.2022

Malte Christian Viering

Abstract

The Internet and the services it provides have become an omnipresent part of our lives. Asynchronous distributed systems form the basis of these services. Resiliency in the face of *partial failures* is an essential requirement for many distributed systems, meaning the systems must continue to function as specified even if several components fail. Ensuring correct behavior, particularly when it comes to failures and asynchrony, makes programming such systems very challenging. Multiparty session types (MPSTs) is a typing discipline for concurrent processes that statically ensures desired properties, such as the absence of message reception errors and deadlocks. These properties can help developers implement correct asynchronous message-passing applications. However, existing MPSTs do not support the specification and verification of partial failure-handling or practical fault-tolerant protocols that handle and recover from partial failures. This fundamentally limits the applicability of MPSTs to asynchronous real-world distributed systems.

In this thesis we present our article "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems" [VCE+18], which is the first MPST formulation for *crash failure handling* in asynchronous distributed systems. This work features a lightweight coordinator modeled after widely used systems such as Apache ZooKeeper and Chubby. For this formulation we developed a typing discipline based on MPSTs that supports the specification and static verification of multiparty protocols with failure handling. The model preserves the distributed nature of MPSTs and interacts only with the lightweight coordinator for the purpose of critical decision-making around failure handling. The type system provides subject reduction despite the possibility of failures occurring at runtime. We implemented our formulation as a prototype in Scala, using Apache ZooKeeper for coordination, and used it to implement and verify a distributed logistic regression (LR) model. In the accompanying performance evaluation, the session type distributed LR model has a performance comparable to failure agnostics distributed LR models in the absence of failures.

We also present our article, "A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming" [VHEZ21], which combines ideas from the previous model with observations from fault-tolerant middleware systems. This work is the first formulation of MPSTs for practical *fault-tolerant distributed programming* of asynchronous

distributed systems. In this work, we give structure to communication patterns involving asynchronous communication and concurrent failures and integrate the features required to express practical fault-tolerant protocols involving dynamic replacement of failed parties and the retrying of failed protocol segments in the presence of imperfect failure detection (perfect failure detection is impossible in asynchronous distributed systems). Key to our approach is the development of the first model of *event-driven concurrency* for multiparty sessions to unify the session-typed handling of failures and regular I/O events. Moreover, the characteristics of our model allow us to prove a *global progress* property for well-typed processes engaged in multiple concurrent sessions. Global progress traditionally does not hold in MPST systems. To demonstrate its practicality, we implement our approach as a toolchain for Scala and use it to specify and implement a session-typed version of the cluster manager (CM) of the widely employed Apache Spark data analytics engine. Our session-typed CM integrates with other vanilla Spark components to give a functioning Spark runtime, i.e., it can execute existing unmodified third-party Spark applications. Measured on an industry-standard benchmark Apache Spark has an average performance overhead below 10% when using our session-typed CM instead of Spark's default CM, in the absence of failures.

The developed MPSTs typing disciplines and prototypes enable the specification and verification of practical distributed applications that handle partial failures. Thus, we enable the verification of desired properties and, in turn, help develop correct distributed applications.

Abstract

Das Internet und die Dienste, die es bereitstellt, sind ein omnipräsenter Teil unseres Lebens geworden, und asynchrone verteilte Systeme bilden die Basis dieser Dienste. Eine wichtige Eigenschaft von verteilten Systemen ist, dass sie robust mit partiellen Fehlern umgehen. Diese Eigenschaft ermöglicht den verteilten Systemen, trotz Fehlern in mehreren Komponenten, weiterhin den Anforderungen entsprechende Fortschritte zu machen. Das Programmieren von verteilten Systemen wird durch die Notwendigkeit mit partiellen Fehlern umzugehen, sehr anspruchsvoll und fehleranfällig. Insbesondere das Sicherstellen der Anforderungen, unter Berücksichtigung von Asynchronität und möglichen Fehlern, ist eine Herausforderung. Multiparty Session Types (MPSTs) ist eine Klasse von Typsystemen für parallele Prozesse, die wünschenswerte Eigenschaften, wie beispielsweise die Abwesenheit von Empfangsfehlern und Deadlocks, statisch sicherstellt. Daher können MPSTs das Programmieren von verteilten Anwendungen vereinfachen. Bisher existierende MPSTs-Ansätze können nicht auf eine signifikante Klasse von verteilten Systemen angewendet werden, da diese die Spezifikation und Verifikation von Crash-Stopp-Fehlerbehandlung nicht ermöglichen. Weiterhin bieten existierende MPSTs-Ansätze nicht die nötigen Abstraktionen zur Spezifikation und Verifikation von praxistauglichen Protokollen, die partielle Fehler behandeln und tolerieren.

Diese Ausarbeitung stellt unseren Artikel “A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems” [VCE+18] vor, der die erste MPSTs Formulierung für Crash-Stopp-Fehlerbehandlung in asynchronen verteilten Systemen ist. Diese Arbeit verwendet ein leichtgewichtiges Koordinatormodell, inspiriert durch die weit verbreiteten Systeme wie Apache ZooKeeper und Cubby. Für dieses Modell haben wir ein MPSTs-Typsystem entwickelt, dass das Spezifizieren und Verifizieren von Protokollen mit expliziter Fehlerbehandlung von partiellen Fehlern ermöglicht. Der Koordinator ist nur in der kritischen Behandlung von partiellen Fehlern involviert, daher bewahrt die Arbeit die verteilte Struktur von MPSTs. Wir zeigen, dass das Typsystem Subject-Reduction erfüllt, trotz des möglichen Auftretens von Fehlern während der Laufzeit. Wir haben einen Prototypen in Scala entwickelt, der Apache Zookeeper als Koordinator verwendet und demonstrieren damit die praktische Einsetzbarkeit unserer MPSTs Formulierung zur Entwicklung und Verifikation von verteilten Anwendungen, die robust mit partiellen

Fehlern umgehen können. In der durchgeführten Evaluation hat der entwickelte Prototyp außerhalb der Fehlerbehandlung keinen relevanten Performance Overhead.

Weiterhin stellen wir unseren Artikel "A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming" [VHEZ21] vor, dies ist die erste Formulierung von MPSTs für die praxistaugliche Programmierung von fehlertoleranten verteilten Anwendungen in asynchronen verteilten Systemen. Wir beantworten eine offene Frage im Bereich von Session Types; wie können wir Kommunikationsmustern, die Asynchronität und parallele Fehler beinhalten eine Struktur geben und gleichzeitig die nötigen Funktionen bereitstellen, die für praxistaugliche fehlertolerante Protokolle nötig sind. Diese Protokolle benötigen Funktionen, wie das dynamische Ersetzten von fehlerhaften Teilnehmern, das Wiederholen von Protokollsegmenten, die wegen Fehlern gescheitert sind und das Tolerieren von nicht perfekten Fehlerdetektoren (da eine perfekte Fehlererkennung in asynchronen System nicht möglich ist). Kern unseres Ansatzes ist, dass wir das erste Modell für eine eventbasierte Nebenläufigkeit für MPSTs entwickelt haben. Dieses Modell behandelt normale IO-Nachrichten und Fehlerbenachrichtigungen uniform. Dies ermöglicht es uns, nicht nur eine Vielzahl an Funktionen zu realisieren, sondern ist auch Grundlage für unser Global Progress Resultat, was traditionell nicht in MPSTs gilt. Um die praktische Einsetzbarkeit unseres Ansatzes aufzuzeigen, haben wir eine Toolchain in Scala entwickelt. In dieser haben wir einen Session typisierten Cluster Manager (CM) für Apache Spark entwickelt. Unsere Session typisierter CM interagiert mit Apache Spark Komponenten und ergibt eine funktionierende Apache Spark Laufzeitumgebung. Der CM kann existierende und unmodifizierte Spark Anwendungen, auch von Dritten, ausführen. In einem Standardbenchmark hat Apache Spark, wenn es den Session typisierten CM anstelle des Standard CM verwendet, einen durchschnittlich Overhead von unter 10%.

Die entwickelten neuartigen MPSTs basierten Typsysteme und Frameworks ermöglichen die Spezifikation und Verifikation von praxistauglichen verteilten Anwendungen, die partielle Fehler behandeln. Wir ermöglichen damit eine leichtgewichtige Verifikation, stellen wünschenswerte Eigenschaften sicher und unterstützen damit die Entwicklung von robusten verteilten Anwendungen.

Contents

List of Figures

1. Introduction

The Internet and the services that it provides have become an omnipresent part of our lives. Networked distributed systems form the basis of these services, e.g., web services and cloud-based data processing. Developing applications which execute across a set of physically separated nodes is a major challenge. A *distributed application* requires a correctly designed communication protocol that describes the asynchronous communication between distributed processes. And the processes comprising the distributed application must correctly implement the protocol in accordance with the role they play in the protocol.

Asynchrony is a crucial reason why developing distributed applications is challenging. This is because the processing speeds of different processes and the message transmission delays between processes are unbounded. Consequently, steps that would occur in a particular order on a single machine may not occur in the same order in a distributed application. The problem is amplified enormously when a distributed application has to be resilient against partial failure, so that, even if some processes fail, others must remain functional. Partial failures affect both the correctness and the progress of distributed applications. Once again, asynchrony is the reason why dealing with partial failures is so tricky. In general, asynchrony makes it impossible to distinguish a failed process from an extremely slow process, and the famous impossibility result [FLP85] states that when only one process can fail, it is impossible to reach an agreement in an asynchronous system. Furthermore, dealing with partial failures often requires additional logic such as the replacement of failed processes.

In many areas, type systems provide developers with a helpful tool that is accessible in the early stages of development. Type systems are among the most widely used lightweight verification tools. Many programming languages support static type checking. Four of the five most popular programming languages, based on the TIOBE index[1], offer static type checking. Type systems are usually closely integrated with the development process, provide early and rapid feedback to the developer, are almost free, and verify many relevant properties. Type systems can for example prevent the following errors: memory

[1] TIOBE Index for May 2021, `https://www.tiobe.com/tiobe-index/` (accessed May 21st of 2021)

errors [MI14], type mismatches, e.g., in method calls [IPW01], and leakage of sensitive information to untrusted parts [SM03]. Behavioral type systems are a kind of type system that can describe the behavior of a program. Intuitively, behavioral type systems describe *how* a calculation proceeds, whereas "classical" type systems describe *what* (result) a calculation computes [HLV+16] – allowing type systems to verify an even broader class of beneficial properties. Examples of behavioral types include: typestates [SY86, SNS+11] that restrict the operations that may be carried out on an entity depending on its current state; and type-and-effect systems [NN93] where the type describes what is calculated and the effect describes how it is calculated. See Hüttle et al. [HLV+16] for a recent and broad survey of behavioral types focusing on session types and behavioral contracts.

Behavioral type systems are an active and promising area towards addressing the challenges of correctly developing distributed applications. Session types (STs) and multiparty session types (MPSTs), in particular, are well-established behavior type systems for concurrent message-passing programs that statically ensure properties such as the absence of message reception errors and deadlocks. The original MPSTs paper [HYC08] recently won the Most Influential POPL Paper Award.[2] And the Journal of the ACM published an extended and revised version of the original MPSTs work [HYC16]. MPSTs generalize STs to more than two parties, i.e., MPSTs can type multiparty interactions. MPSTs constitute a promising technique for distributed systems: an essential concept in MPST theory is projection, which derives a decoupled (i.e., distributed) view of a protocol for each participant. STs and MPSTs were originally developed for the π-calculus [HVK98] but have been successfully applied to many practical languages, e.g., Java [HY16, SQZ+13], Scala [SDHY17a], Haskell [LM16, PT08], and OCaml [IYY17, Pad17]. We believe that behavioral type systems such as MPSTs can continue the success story of type systems for distributed systems.

Figure 1.1 provides an overview of the standard MPST framework. It starts with a user-provided protocol specification, the global type. The global type describes the asynchronous multiparty communication from a neutral perspective. The framework derives a protocol specification, the local type for each endpoint participating in the global type. The local types describe the localized behavior and allow endpoints to play their part in a protocol without global supervision. The framework uses the local type to *statically* type check the I/O actions of the implementations for the different endpoints. A well-typed system of session endpoint programs enjoys important safety and liveness properties, such as *no reception errors* (only expected message types are received) and *session progress* (absence of deadlocks).

[2] Most Influential POPL Paper Award `https://www.sigplan.org/Awards/POPL/` (accessed May 21st of 2021)

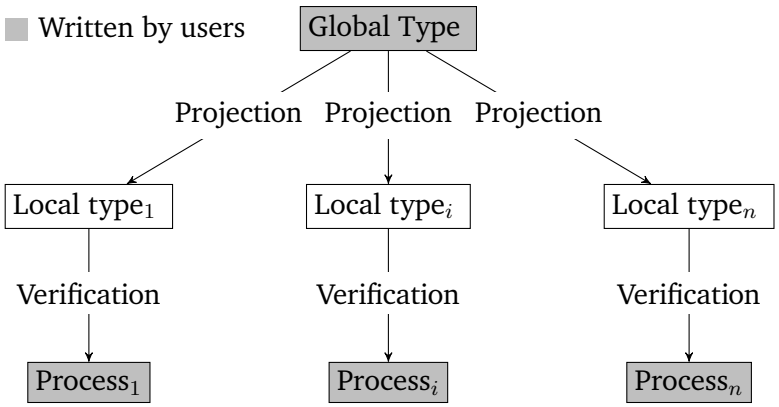

Figure 1.1.: Overview of the multiparty session types framework.

MPSTs are an active research area, and the message passing aspect of MPSTs makes them well suited to capturing the asynchronous behavior of distributed systems. Unfortunately, current MPST works cannot handle partial failures in an asynchronous distributed system. Partial failures refer to the situation in which some processes fail, by crashing, while others remain operational. However, failures are an inevitable part of distributed systems, and a broad class of distributed applications has to deal with partial failures. Not supporting partial failures fundamentally limits the applicability of MPSTs to asynchronous real-world distributed systems.

1.1. MPSTs for Fault-tolerant Distributed Applications

MPSTs are not ready for practical fault-tolerant distributed programming because no existing asynchronous MPSTs work (apart from the works presented here) considers crash-stop failures and provides the tools necessary to handle such failures. We will now discuss the obstacles that MPSTs must overcome before developers can benefit from the calm, order, and safety that MPSTs offer while implementing distributed applications that deal with all the chaos of failures. It should be noted that, to the best of our knowledge, no existing work (outside of this thesis) addresses these obstacles.

(Crash-stop) failures. Failures are a normal part of (asynchronous) distributed systems [Bur06], i.e., not considering failures means ignoring an essential class of distributed applications. Crash-stop failures[3], in particular, are a normal part of distributed

[3]Distributed systems can also suffer from other failures, such as network partitions [GL02] or Byzantine

systems. A crash-stop failure occurs when a process crashes and stays down. This can happen at any time. This behavior makes crash-stop failures hard to handle for the particular reason that the failed process cannot help with ensuring consistent failure handling.

The behavior of crash-stop failures is in stark contrast with that of application-level failures or interrupts, which are studied in MPSTs [CGY16, DHH+15], such as division by zero or out of memory exceptions. Application-level failures can be handled by the process, in which the failure occurred, i.e., the subject of the failure can handle the failure. This is not possible in the case of crash-stop failures.

Asynchrony. Typical distributed applications have to deal with asynchrony because the Internet and typical data centers are subjected to asynchrony. That makes implementing distributed applications challenging because they must consider that the processing speeds and message transmission delays are unbounded. Consequently, steps that would occur in a particular order in a single machine may not occur in the same order in a distributed application.

MPSTs are well equipped to capture asynchronous behavior in distributed applications that communicate over reliable channels such as TCP. However, asynchrony also vastly complicates dealing with partial failures, and, to the best of our knowledge, *no* MPST work (outside of this thesis) considers failures, such as crash-stop failures, in an asynchronous reduction semantic. However, asynchrony is the reason why it is impossible to reach an agreement even when only one process fails [FLP85] and why it is impossible to accurately detect a failed process [CT96].

Application specific failure handling. What to do when a failure occurs is highly application specific. Different distributed applications provide different means of remedying failures. For some distributed applications it may be sufficient to safely terminate when a failure occurs. But other distributed applications must remain operational even in the face of multiple failures. This means that they must replace failed processes, for example. Application specific failure handling, and therefore failures, can influence the communication pattern and behavior that occurs inside a distributed application.

Abstraction and programming features. Failures, asynchrony, and the resulting nondeterminism influence the ways in which fault-tolerant distributed applications are

failures [LSP82]. These are interesting subjects for future work. Note that tolerating network partitions requires sacrificing either consistency or availability [GL02]. And it has been argued that Byzantine failures are an excessively generic failure model (e.g., [Bir17]).

implemented.

It is very difficult to deal correctly with failures, so a lot of distributed applications use extremely reliable coordination services, such as ZooKeeper [Hun10] or Chubby [Bur06], as a tool for simplifying fault-tolerance [VCE+18]. But these services do not in any way render the implementation of fault-tolerant applications trivial.

Event-driven programming (EDP) or similar styles are commonly used programming and concurrency models for dealing with the nondeterminism in fault-tolerant applications. Apache Spark uses EDP, for example. EDP makes it possible to treat standard I/O actions and failures uniformly.

Failures and their handling is a cornerstone of the implementation of fault-tolerant distributed applications. However, practical applications must also deal with further aspects, such as replacing failed processes, concurrent tasks, and working with a variable number of processes depending on the workload.

Implementation. A safe typing discipline does not directly help developers implement distributed applications. It is vital to ensure that a typing discipline for failure handling can specify protocols and verify real distributed applications that deal with partial failures. Moreover, a framework for failure handling should not add too much performance overheads. Therefore, a prototype, the implementation of relevant use cases, and performance evaluation are required.

1.2. Problem Statement

MPSTs are a good foundation for the specification and verification of distributed applications, because they ensure properties like communication safety and progress. However, as highlighted in the previous section, we must address a large number of obstacles before MPSTs provide the means of verifying practical fault-tolerant distributed applications. We now discuss the high-level problems that we need to address before MPSTs can become a helpful lightweight verification tool to support the development of fault-tolerant distributed applications.

1. We need to extend asynchronous MPSTs with a concept of failure that is faithful to crash-stop failures. Such a concept can be explicit (modeling a crash) or implicit, but it must take into account the nondeterminism and unpredictability behind crash-stop failures. We need to study such failures in asynchronous MPSTs, because asynchrony is the key reason why failure handling in distributed systems is so difficult.

We have to bear in mind here that in general it is impossible to reach an agreement in an asynchronous system even if only one process fails [FLP85]. To overcome this impossibility, we need to make appropriate assumptions.

We need to work out assumptions that are both practical and faithful and also work in an MPST framework. We can, for example, consider concepts used in existing fault-tolerant distributed applications, such as coordination services or program-controlled crashes.

2. We need to extend the global types and local types with abstractions for application-specific failure handling. Furthermore, we need to provide a programming construct to implement the failure handling.

   Providing mechanisms for the specification of failure handling is the first step, because it allows us to specify and verify fault-tolerant distributed applications. We then need to enrich MPSTs with features that include concurrent tasks potentially involving different participant subsets, parameterization, restarting of failed tasks, and dynamic replacement of failed participants, because practical fault-tolerant applications rely on more than just failure handling.

   Furthermore, we should provide a programming model already used by existing systems, such as EDP, instead of the π-calculus that is the traditional foundation of MPSTs.

3. We need to verify the designed methods for dealing with the asynchronous and chaotic behavior of failures by proving desired properties. Here we need to consider that a consequence of asynchrony and failures is that failure handling itself is subject to both, i.e., failure handing can potentially be interrupted by new failures and therefore only partially executed. Or different participants can react to different failures.

   An important aspect to consider in ensuring the desired properties is that MPSTs define a very rigid invariant that describes how different processes relate to each other (cf., coherence and consistency [HYC16, CDYP16]). MPSTs ensure this invariant for the entire execution, and this rigid invariant is the foundation for establishing subject reduction. This invariant does not take failure into account. We therefore need to modify this invariant so that it can deal with partial failures. We need to relax the invariant to allow for inconsistencies that can occur because of failures, e.g., a process that waits for a message from a failed process or a message in transit where the receiver failed. At the same time, we need to ensure that the invariant remains rigid enough to prove the desired properties.

4. We need to implement our formalization to verify that it is implementable and in practices expressive enough to verify relevant distributed applications. In addition, we need to evaluate the implementation to ensure that it does not introduce an undue performance overhead.

This thesis presents two typing disciplines that extend MPSTs to deal with crash-stop failures in asynchronous reduction semantics (addressing Problem 1). Both extend MPSTs to offer application-specific failure handling (addressing Problem 2). And we establish and prove relevant properties (addressing Problem 3). Lastly, we provide prototype implementations for both formalizations and show that they provide reasonable performance (addressing Problem 4). To add some perspective, we will now discuss the work of Adameit et al. [APN17]. To the best of our knowledge, this is the only other MPSTs work that studies distributed system failures as we do in this thesis. We will then discuss further related works. See Section 2.2 for a detailed discussion of related work.

Adameit et al. [APN17] study link failures in synchronous MPSTs. They introduce an optional block that allows an enclosed session fragment to be executed as standard. But the block may also abruptly discard the ongoing execution non-deterministically. In both cases, the session outside of the optional block continues, using either the results calculated within the optional block or default values if the optional block has been discarded, i.e., the optional block expresses failure masking rather than failure handling. Failure masking needs to be distinguished from failure handling. Both are important concepts that are used in the implementation of fault-tolerant distributed applications. Besides targeting a different set of fault-tolerant distributed applications (failure masking rather than failure handling), the work uses *synchronous* MPST semantics. However, asynchrony is a crucial reason why dealing with partial failures is challenging. That said, the authors convincingly argue that the main example, a rotating coordinator algorithm, can be considered as a distributed asynchronous process; because of its structure, e.g., it uses no choice and outputs have no continuation different than 0 [APN16]. Lastly, the work does not offer an implementation.

Besides the work of Adameit et al. [APN17], Fowler et al. [FLMD19] add an exception handling process primitive for failures in (binary) STs. Type-level treatment of failure handling behaviors between the remaining (and new) peers, as in this thesis, cannot be studied in a binary setting. Several other works study application-level exception or interrupt handling (e.g., [DHH$^+$15, CGY16]). We discuss related work in detail in Section 2.2.

1.3. Thesis Statement

This thesis extends MPSTs to handle partial failures in asynchronous distributed systems and combines failure handling with several practical and advanced features that enable specification and verification of practical fault-tolerant distributed protocols and their fault-tolerant implementations. We propose and implement a novel framework that extends MPSTs with a coordination service inspired by practical and widely used coordination services, creating the first asynchronous MPST framework that supports partial failure handling for distributed applications in an asynchronous distributed system. We combine ideas from the first model with observations from fault-tolerant middleware systems and derive the first event-driven formulation of MPSTs. This novel model allows the specification and verification of practical fault-tolerant protocols and their implementations, has strong formal properties such as global progress, can deal with false failure suspicions, and is expressive enough to build components that integrate with an existing and widely used fault-tolerant middleware system, namely Apache Spark.

1.4. Outlook and Summary

We now provide a preview of the rest of this chapter and the rest of the thesis. We summarize the two main technical sections, i.e., Chapters 3 and 4. We then list the contributions of this thesis and conclude with a list of publications and how they relate to this thesis. Chapter 2 presents explicitly optional background, related to the two main technical chapters, and a discussion of related work. Chapter 3 presents our article "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems", which appeared at the European Symposium on Programming 2018 [VCE+18]. Chapter 4 presents our work "A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming", which appeared at Proceedings of the ACM on Programming Languages 2021 [VHEZ21]. Both Chapters 3 and 4 are self-contained and can be read in isolation. Chapter 5 concludes the thesis and provides a short discussion of future work.

Appendix A provides the appendix for Chapter 3. It provides additional examples, omitted definitions and the proof details. Appendix B provides the appendix for Chapter 4. It provides additional examples, more details on the main use case and the system model, omitted definitions and the proof details.

1.4.1. A typing discipline for statically verified crash failure handling in distributed systems

Chapter 3, which presents our article, "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems" [VCE+18], introduces a formal model for crash failure handling in asynchronous distributed systems featuring a lightweight coordinator, modeled along the lines of widely used systems such as Apache ZooKeeper [Hun10] and Chubby [Bur06].

For this model we develop a typing discipline based on MPSTs that supports the specification and static verification of multiparty protocols with explicit failure handling. We thereby make partial failure handling in MPSTs available for a widely used class of distributed systems, e.g., many systems in the family of Apache middleware systems use ZooKeeper [Hun10] by default or as an option for fault-tolerance, such as Apache Mesos [HKZ+11], Apache Hadoop YARN [VMD+13a], and Apache Spark [ZCD+12b].

We introduce a try-handle construct in the global type, local type, and the processes for the specification and implementation of failure handling. In a global type a *try-handle* $\left(G_1 \blacktriangleright H^{\uparrow}\right)^{\kappa}.G_2$, describes a "failure-atomic" protocol unit: all *live* (i.e., non-crashed) roles will eventually reach a consistent protocol state, despite any concurrent and asynchronous role crashes. The try-block G_1 defines the *default* protocol flow, and $H^{\uparrow}$ is a *handling environment*. Each element of $H^{\uparrow}$ maps a *handler signature* F, that specifies a *non-empty* set of *failed* roles $\{p_i\}_{i \in I}$, to a *handler body* specified by a G. The handler body G specifies how the live roles should proceed given the failure of roles F. The protocol then proceeds (for live roles) according to the continuation G_2 after the default block G_1 or failure handling defined in $H^{\uparrow}$ has been completed as appropriate.

We introduce a coordination process Ψ. The coordinator plays a crucial role in ensuring consistent partial failure handling. Importantly, processes interact with the coordination service via asynchronous message passing, and the coordination does not know immediately when a failure occurs. Furthermore, the coordination is not involved in normal message exchange between participants; it is only involved in the coordination to ensure consistent partial failure handling. The last point is crucial in practice, because interactions with the coordinator are expensive in comparison with normal interactions between participants, and it is therefore important not to involve it in the normal asynchronous interactions.

We show that our type system ensures subject reduction in the presence of failures. Our published article [VCE+18] provides a progress property for our formal model. In other words, in a well-typed system, even if some participants fail during execution, the system is guaranteed to progress in a consistent manner with the remaining participants.

Lastly, the chapter presents a prototype implementation and a performance evaluation.

The prototype is based on the formal model, uses Apache ZooKeeper for coordination, and is written in Scala. The evaluation shows that a session type logistic regression model has a runtime performance comparable to that of failure-agnostic baseline implementations.

1.4.2. A multiparty session typing discipline for fault-tolerant event-driven distributed programming

Chapter 4, which presents our article, "A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming" [VHEZ21], develops an MPST-based theory for *practical* programming of *fault-tolerant* distributed applications. The work draws inspiration from one of the major paradigms for distributed programming in practice, *event-driven programming* (EDP), and introduces an event-driven reduction semantics to MPSTs. Furthermore, the work adds several advanced features needed for practical fault-tolerant protocols. In addition, the proposed formalization is designed to tolerate false failure suspicions. This enables MPST-based specification and verification of practical fault-tolerant multiparty asynchronous protocols that deal with *crash-stop* process failures to ensure the key properties of communication safety and progress. Moreover, it enables us to prove a *global progress* property for well-typed processes engaged in multiple concurrent sessions, which does not hold in traditional MPST systems.

We introduce two new core abstractions, *role sets*, and failure-aware *subprotocols* to global and local types. Role sets provide an abstraction for a *set* of participant processes *capable* of the same behavior. E.g., a set of worker processes where worker processes are selected during execution to perform different tasks or a set of master processes to provide redundancy for potential failures. Role sets provide a practical, lightweight form of parameterization. Subprotocols provide an abstraction for concurrent failure-aware tasks. We now illustrate our concept of a *failure-aware concurrent task*:

$$g(r_1, ..., r_n; r_p; R_1, ..., R_m) = G \text{ with } r_p@r_i \,.\, G$$

It specifies task g whose participants are the individual processes playing roles $r_1, ..., r_n$ and r_p, combined with all member processes of the role sets $R_1, ..., R_n$. The arguments are separated into three groups separated by ';'. Each role, e.g., r_1, is implicitly associated with a role set. The role arguments $r_1, ..., r_n$ (first group) specifies participants that occur in the subprotocol; the r_p in the second argument group specifies that this participant should be *dynamically* assigned from its role set when this subprotocol is spawned, and the last argument group specifies role sets that occur in the protocol.

The subprotocol definition has two parts: the *normal activity* written before the with, and the *failure handling activity* after the with. The $r_p@r_i$ clause specifies that, in this subsession,

r_i is responsible for *monitoring* r_p for failure. By default, the subsession proceeds following the interactions in the normal activity. However, if r_i *suspects* r_p of failure, r_i will switch to the failure handling and inform the other participants about the failure; it may do so at any point during the subsession.

The work introduces a novel event-driven reduction semantic and concurrency model. Instead of using a parallel composition of sequential processes in the context of the π-calculus, which is the typical style used in MPSTs. Participant processes are written as an *event loop* that contains a set of *event handlers*. For instance, the runtime calls event-handlers for receiving a message once that message is present in the queue. Event handlers have the following form:

$$[L]\lambda x \,.\, P$$

The above event-handler has two parts. The function $\lambda x \,.\, P$ is the behavior of the event handler, where x is the session channel variable and P is the event handler body. $[L]$ describes the action performed on the session channel x in P. The runtime system calls an event handler when the session is in a state that is compatible with L and the required events are ready, e.g., if P receives a message on x then the handler is fired if the message is present in the queue.

We show that our type system ensures subject reduction, session progress, and global progress in the presence of failures. It should be noted that global progress does not generally hold in MPSTs. In other words, in a well-typed system, even if some participants fail during execution, the system is guaranteed to progress in a consistent manner with the remaining participants.

To demonstrate the practicality of our approach, we implement our system as a toolchain for fault-tolerant distributed programming in Scala and use it to specify and implement a session-typed version of Spark's cluster manager (CM). Our session-typed CM is compatible with other vanilla Spark components and provides a functioning Spark runtime to execute existing third-party Spark applications *without* any code modifications. We use a Spark implementation of the industry-standard TPC-H benchmark suite[4] to test and evaluate the runtime performance of our session-typed CM. Measuring the time it takes from submitting a Spark application to its completion, the results show our prototype implementation incurs an average overhead below 10%, in the absence of failures.

[4]Transaction Processing Performance Council (TPC), `http://www.tpc.org/tpch/` (accessed May 20, 2021)

1.5. Contributions

In this dissertation, we introduce and implement two novel typing frameworks that extend MPSTs to enable partial failure handling in asynchronous distributed systems.

1.5.1. A typing discipline for statically verified crash failure handling in distributed systems

Chapter 3, which presents our article "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems" [VCE+18], introduces a new typing discipline for safe specification and implementation of distributed programs prone to process crash failures based on MPSTs in a model using a coordination service. The following summarizes the contributions of that work.

Multiparty session calculus with coordination service. We develop an extended multiparty session calculus as a formal model of processes prone to crash-stop failures in asynchronous message passing systems. Unlike standard session calculi that reflect only "minimal" networking infrastructures, our model introduces a practically-motivated *coordinator* artifact and explicit, asynchronous messages for run-time crash notifications and failure handling.

MPSTs with explicit failure handling. We introduce new global and local type constructs for *explicit failure handling*, designed for the specification of protocols that tolerate partial failures. Our type system carefully reworks many of the key elements in standard MPSTs to manage the intricacies of handling crash failures. These include the well-formedness of failure-prone global types, and the crucial *coherence* invariant of MPST typing environments to reflect the concept of system consistency in the presence of crash failures and the resulting errors. We show safety for a well-typed MPST session despite potential failures.

Prototype and performance evaluation. Our prototype, which is implemented in Scala, uses Apache ZooKeeper as a coordination service. We compare the performance of a session typed logistic regression model that uses our prototype with failure-agnostic baseline implementations. The session typed version offers a performance similar to the failure-agnostic baseline implementations in the absence of failures.

1.5.2. A multiparty session typing discipline for fault-tolerant event-driven distributed programming

Chapter 4, which presents our article, "A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming" [VHEZ21], combines the ideas of Chapter 3 with observations from fault-tolerant middleware systems. When compared with Chapter 3, it uses a more practical and flexible failure and system model, by, e.g., allowing for false suspicions and not requiring a coordination artifact. Furthermore, it adds features that are required for the expression of many practical *fault-tolerant* distributed applications. Lastly, we demonstrate its practicality by realizing a complex use case and show that it has a negligible performance overhead. The following summarizes our contributions.

Practical fault-tolerance. We develop the first session typing discipline that supports and integrates a range of novel features needed to specify *fault-tolerant* multiparty interactions, including *failure-aware subprotocols*, *participant parameterization* and *dynamic role assignment*.

EDP. We present the first MPST-based system for *event-driven concurrency*. We develop an endpoint projection and type system for distributed processes implemented and executed as components driven by asynchronous I/O events. The central abstractions are *session-typed event handlers* and *event loops*

MPST properties for practical fault-tolerance. We formalize our system and prove the key properties of *communication safety* and *global progress* for an entire system of *multiple, concurrent sessions*. Our framework offers global progress, which generally does not hold in MPSTs. And integrating the range of features needed to meet our aims in a tractable formalism is a significant challenge for MPSTs. We achieve this by exploiting EDP to unify handling of regular I/O and failure handling events.

Implementation and evaluation. We implement our framework as a toolchain for safe programming of fault-tolerant distributed applications in Scala. To evaluate our approach we specify and implement a session-typed version of Spark's cluster manager (CM) component, which is compatible with the other existing Spark runtime components, and supports the execution of existing Spark application code without modification.

1.6. Publications and Thesis-related Correspondence

[CVB+16] Tzu-Chun Chen, **Malte Viering**, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A Type Theory for Robust Failure Handling in Distributed Systems. In Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, volume 9688, pages 96–113. Springer, 2016

Declaration of originality. Tzu-Chun Chen was the technical lead and lead author of that work. I contributed to that work with writing, technical discussion, and technical modifications and extensions to many parts of the technical development. All authors contributed to the work in technical discussions, writing, and refinement of the presentation of the paper.

Thesis correspondence. This work does not directly appear in this thesis. The proofs in Chapters 3 and 4 follow, modify, and extend the proof schemes that we used and developed in that work.

[VCE+18] **Malte Viering**, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, volume 10801 of Lecture Notes in Computer Science, pages 799–826. Springer, 2018.

Declaration of originality. I was the technical lead and lead author of this work. The work was created in close cooperation with Tzu-Chun Chen, however. Tzu-Chun Chen headed the initial development of the type system and proving of the properties. Furthermore, she contributed technical modifications and extensions to other parts of the formal model. Raymond Hu contributed the initial version of the Three-Buyer and Streaming examples. Furthermore, all my co-authors contributed to the work in technical discussions, writing, and refinement of the presentation of the paper.

Thesis correspondence. Chapter 3 mostly presents that work *directly*, i.e., the chapter (including its supplement) is for significant parts a verbatim copy of that work. The chapter slightly revises the calculus and adds additional explanation, examples, and a new evaluation. Furthermore, I reuse parts of the writing in other parts of the thesis.

[VHEZ21] **Malte Viering**, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. In Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA, October 2021.

Declaration of originality. I am the technical lead and lead author on this work. Raymond Hu created the first version of Figures 4.1 and 4.4 to 4.6. My coauthors contributed to the work in technical discussions, writing and refinement of the presentation of the paper.

Thesis correspondence. Chapter 4 mostly presents that work *directly*, i.e., the chapter (including its supplement) is for significant parts a verbatim copy of that work. Furthermore, I reuse parts of the writing in other parts of the thesis.

1.6.1. Additional publications

In addition to the papers mentioned above, I co-authored the following papers during my PhD studies. These papers are not part of the present PhD thesis.

[BDV+19] Andi Bejleri, Elton Domnori, **Malte Viering**, Patrick Eugster, and Mira Mezini. Comprehensive multiparty session types. The Art, Science, and Engineering of Programming, 3(3):6, 2019.

[BVSE17] Marcel Blöcher, **Malte Viering**, Stefan Schmid, and Patrick Eugster. The grand CRU challenge. In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems, HotConNet@SIGCOMM 2017, pages 7–11. ACM, 2017.

2. Background

This chapter provides general background information on MPSTs and a discussion of relevant related work. In the first part, Section 2.1, we provide an informal description of MPSTs and explain the user view and underlying framework of MPSTs and the methodology for establishing properties. This provides optional background information for Chapters 3 and 4. In Section 2.2, we discuss related work with a focus on MPSTs.

2.1. Multiparty Session Types

In this section, we present optional background information for Chapters 3 and 4. We begin by explaining MPSTs. In particular, we explain the programming model and user view, the typing system, and a methodology for establishing type system properties. For this purpose, we present an adaptation of the works of Coppo et al. [CDYP16] and our previous work [CVB+16]. Our adaptation mostly follows the work of Coppo et al. [CDYP16], but in particular, for the section about establishing properties we adopt parts of our work [CVB+16]. We present an adaptation of these two works, because this allows us to align this section closely with Chapter 3. This chapter also provides background for Chapter 4, but in that chapter we adopt MPSTs for EDP, which is novel.

Multiparty session types (MPSTs) are a behavioral type system for (distributed) applications that use channels as communication primitives [HYC16, CDYP16]. MPSTs define the usage of channel primitives and ensure their safe usage. In particular, they ensure, *communication safety*, *fidelity*, and *progress*. Communication safety guarantees that there is never a mismatch between the received message type and the expected type. Fidelity ensures that processes follow a defined protocol specification. Lastly, progress ensures that a non-terminated application can always perform a reduction step.

MPSTs structures communication in sessions and typically follow a top-down approach, as illustrated in Figure 2.1 (left side). The top-down approach starts from a user-specified global type, a protocol that describes the entire communication in a session from a neutral perspective. The global type (top-level) is used to type check the processes that implement

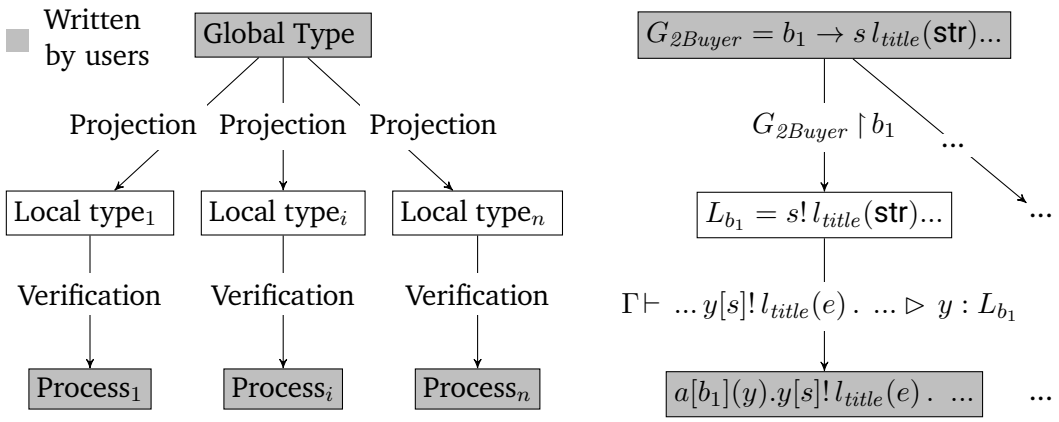

Figure 2.1.: Overview of the multiparty session types framework (left side) and an illustration based on the running example (right side).

the (distributed) application (bottom level). The user writes the global type and the implementation, whereas the steps in between are part of the MPST framework.

MPSTs [HYC08] were developed as a generalization of session types (STs) [HVK98], i.e., they generalized typing from two-party communications to multiparty communications. STs and MPSTs were originally introduced for the π-calculus [HVK98, HYC08], and subsequently applied successfully to a broad range of practical languages, e.g., Java [HY16, SQZ+13], Scala [SDHY17a], Haskell [LM16, PT08], OCaml [IYY17, Pad17].

Remark. *Throughout this thesis, we occasionally refer to standard or classic MPSTs. We are then referring to the works of Honda et al. [HYC16] and Coppo et al. [CDYP16].*

Figure 2.1 provides an overview of the MPST framework, which follows the typical top-down approach (left part), and illustrates it in accordance with the running example in this section (right part). A software architect specifies a global type, i.e., the protocol, which describes the communication between all, say n, participants of a distributed application. The right-hand side shows the beginning of the two-buyer global type (see Figure 2.2 for the full global type). The global type gets projected onto a local type for each of the n participants. On the right-hand side $G_{2Buyer} \upharpoonright b_1$ denotes the projection of the two-buyer global type onto the participant b_1 (see Figure 2.3). A local type describes the communication that one participant performs in the protocol, i.e., the messages it sends and receives. $L_{b_1} = s!\, l_{title}(\mathsf{str})...$ (right-hand side) specifies that b_1 will send a message with label l_{title} and type str to s, which is the beginning of the local type of participant b_1. The type system uses the local types to type the different participant implementations

$$
\begin{aligned}
G_{2Buyer} = \quad & b_1 \to s\, l_{title}(\mathsf{str}).\ s \to b_1\, l_{price}(\mathsf{dbl}).\ s \to b_2\, l_{price}(\mathsf{dbl}). \\
& b_1 \to b_2\, l_{share}(\mathsf{dbl}).\ b_2 \to s\{ \\
& \qquad\qquad\qquad\qquad\qquad\qquad l_{addr}(\mathsf{str}).\ s \to b_2\, l_{date}(\mathsf{str}).\mathsf{end}, \\
& \qquad\qquad\qquad\qquad\qquad\qquad l_{quit}(\mathsf{str}).\ \mathsf{end} \\
& \}
\end{aligned}
$$

Figure 2.2.: Global type of the two-buyer protocol [HYC16].

and ensures that the session channels are used as described by the local types. Lastly, developers implement the processes that implement the different protocol participants. The right-hand side shows the beginning of the process ($a[b_1](y).y[s]!\, l_{title}(e)\ .\ ...$) that implements participant b_1. The process starts with an initiation action that starts a new session followed by sending a message with label l_{title} and a value of type str to s (see Figure 2.4 for the full implementation).

We now use examples to give an intuition on the global types, local types, and processes, and then define these terms more precisely. For the purpose of definition, we use the two-buyer protocol as an example, following the specification proposed by Honda et al. [HYC16]. Note that this example is simple and intentionally only uses a subset of the MPST features.

Global types in a nutshell. Before specifying and describing the two-buyer protocol, we explain the two interaction primitives that are required. The first is a (regular) interaction $a \to b\, l(S).G$ which denotes that a sends a message carrying a value of type S and a label l to b; the protocol then continues as G. The second is a choice $a \to b\{l_1(S_1).G_1, ..., l_n(S_n).G_n\}$ which denotes that a sends a message to b and chooses a label from $l_1, ..., l_n$; the protocol will continue with the G_i corresponding to the selected label.

Figure 2.2 specifies the two-buyer protocol, which involves two buyers (b_1 and b_2) and a seller (s). It describes the following behavior. The first buyer b_1 sends the title of the book they want to buy to the seller s. Then the seller sends the book price to both buyers. After that, the first buyer b_1 sends the second buyer b_2 the amount they are willing to contribute. The second buyer then decides whether they are willing to pay the rest or not. They either send the shipping address to the seller, who replies with the shipping date, or inform the seller that they will not buy the book.

Local types in a nutshell. The two-buyer's local types use the send type $a!\, l(S).L$ and receive type $a?\, l(S).L$. The send type describes sending a message that carries a value of type S and label l to a; after that, the protocol continues as L. The receive type is the

$$
\begin{aligned}
G_{2Buyer} \upharpoonright s &= b_1?\, l_{title}(\mathsf{str}).\ b_1!\, l_{price}(\mathsf{dbl}).\ b_2!\, l_{price}(\mathsf{dbl}). \\
&\quad\ b_2?\{ \\
&\qquad\qquad l_{addr}(\mathsf{str}).\ b_2!\, l_{date}(\mathsf{str}).\mathsf{end}, \\
&\qquad\qquad l_{quit}(\mathsf{str}).\ \mathsf{end} \\
&\quad\ \} \\
G_{2Buyer} \upharpoonright b_1 &= s!\, l_{title}(\mathsf{str}).\ s?\, l_{price}(\mathsf{dbl}).\ b_2!\, l_{share}(\mathsf{dbl}).\ \mathsf{end} \\
G_{2Buyer} \upharpoonright b_2 &= s?\, l_{price}(\mathsf{dbl}).\ b_1?\, l_{share}(\mathsf{dbl}). \\
&\quad\ s!\,\{ \\
&\qquad\qquad l_{addr}(\mathsf{str}).\ s?\, l_{date}(\mathsf{str}).\mathsf{end}, \\
&\qquad\qquad l_{quit}(\mathsf{str}).\ \mathsf{end} \\
&\quad\ \}
\end{aligned}
$$

Figure 2.3.: Local types for the two-buyer protocol.

receiving counterpart to the send type. The protocol further uses selection and branching types. The selection type $a!\,\{l_1(S_1).\,L_1, ..., l_n(S_n).\,L_n\}$ describes sending a message to a; the protocol continues with the local type corresponding to the send label, e.g., with L_1 if l_1 is sent. Lastly, the branching type $a?\,\{l_1(S_1).\,L_1, ..., l_n(S_n).\,L_n\}$ is the receiving counterpart to the selection type.

See Figure 2.3 for the local types of the two-buyer protocol. We use $G_{2Buyer} \upharpoonright s$ to state that G_{2Buyer} is projected to s, i.e., the local type of s is calculated from G_{2Buyer}. The different local types should be relatively self-explanatory. We, therefore, only explain the local type of the seller ($G_{2Buyer} \upharpoonright s$). The seller receives the book title and sends the book price to both buyers. The seller then waits to receive either the shipping address, in which case they will send the second buyer the expected delivery date, or a quit, signaling that the buyers do not want to buy the book.

Processes in a nutshell. The processes implementing the two-buyer protocol all start with an initialization instruction, e.g., $a[b_1](y).P$. A initialization instruction $a[p](y).P$ states that this process looks for other participants over the shared name a to start a session. In that session, this process wants to play participant p. During initialization, the session variable y gets replaced with the session channel. The initializing instruction has no matching type. Furthermore, the implementations use the send instruction $y[p]!\, l(e).P$, which sends both label l and the result of the expression e over the channel y to p and then continues as P. The receiving counterpart is $y[p]?\, l(x).P$, which expects label l and a value that gets substituted for x in P. The branching statement $y[p]?\{l_1(x_1).P_1, ..., l_n(x_n).P_n\}$

$$
\begin{aligned}
P_s \;=\;& a[s](y).y[b_1]?\, l_{title}(x_{book}).\\
& y[b_1]!\, l_{price}(price).\\
& y[b_2]!\, l_{price}(price).\\
& y[b_2]?\{\\
& \quad l_{addr}(x_{addr}).\ y[b_2]!\, l_{date}(date).0,\\
& \quad l_{quit}(x_{quit}).0\\
& \}
\end{aligned}
\qquad
\begin{aligned}
P_{b_1} \;=\;& a[b_1](y).\\
& y[s]!\, l_{title}(\text{"MPSTs"}).\\
& y[s]?\, l_{price}(x_{price}).\\
& y[b_2]!\, l_{share}(x_{price}/2).0
\end{aligned}
$$

$$
\begin{aligned}
P_{b_2} \;=\;& a[b_2](y).y[s]?\, l_{price}(x_{price}).\ y[b_1]?\, l_{share}(x_{share}).\mathsf{if}\ (x_{price} - x_{share} < 100)\\
& \qquad y[s]!l_{addr}(addr).\ y[s]?\, l_{date}(x).0\\
& \quad \mathsf{else}\\
& \qquad y[s]!l_{quit}(quit).0
\end{aligned}
$$

Figure 2.4.: The implementation of the two buyers and the seller from the two-buyer protocol.

differs from the receive statement in that the computation continues in P_i depending on the received label, e.g., P_1 if label l_1 was received. Lastly, the implementations use a condition statement if (e) P_1 else P_2 where depending on whether e evaluates to true or false, the computation continues as either P_1 or P_2. The condition statement allows the execution of different interaction primitives, i.e., sending different labels, depending on the evaluation of e.

The implementation of the three processes, in Figure 2.4, is quite self-explanatory. We will explain the second buyer's (b_2) implementation as it implements a selection. The process P_{b_2} of the second buyer starts with an initialization, stating that this process will play the participant b_2 and awaits other processes through shared name a. After session initialization, it receives the book price from seller s and the amount the first buyer (b_1) is willing to contribute. The following condition checks whether this process (b_2) is willing to pay the rest. If the price after subtracting the other buyer's contribution is below 100, this process (b_2) is willing to pay the rest; it (b_2) sends the shipping address to the seller and receives the shipping date from the seller. Otherwise, the price is too high, and this process (b_2) informs the seller and aborts the transaction. The send instructions in the different condition branches, i.e., $y[s]!l_{addr}(addr)$ in the *then* branch and $y[s]!l_{quit}(quit)$ in the *else* branch, implement the selection type $s!\,\{l_{addr}(\mathsf{str}).\ s?\, l_{date}(\mathsf{str}).\mathsf{end}, l_{quit}(\mathsf{str}).\ \mathsf{end}\}$ from b_2's local type.

$$
\begin{array}{rcll}
G & ::= & p \to q\{l_i(S_i).G_i\}_{i\in I} & \text{Branching} \\
 & \| & \mu t.G \mid t & \text{Recursion} \\
 & \| & \mathsf{end} & \text{End} \\
S & ::= & \mathsf{bool} \mid \mathsf{str} \mid \mathsf{int} \mid \ldots & \text{Sorts}
\end{array}
$$

Figure 2.5.: Syntax of global types.

$$
\begin{array}{rcll}
L & ::= & p!\{l_i(S_i).L_i\}_{i\in I} & \text{Selection} \\
 & \| & p?\{l_i(S_i).L_i\}_{i\in I} & \text{Branching} \\
 & \| & \mu t.L \mid t & \text{Recursion} \\
 & \| & \mathsf{end} & \text{End}
\end{array}
$$

Figure 2.6.: Syntax of local types.

2.1.1. Global types

Figure 2.5 defines the global types we use in this chapter. A global type consists of branching types $p \to q\{l_i(S_i).G_i\}_{i\in I}$, which define that p sends a label l_i from $\{l_i\}_{i\in I}$ and a value of type S_i to q; the communication will then continue as described in the selected branching case, i.e., G_i. The value type is a primitive type such as bool or str. If $\{l_i\}_{i\in I}$ contains only one label, i.e., we have a singleton label set, we may write $p \to q\, l(S).\, G$ as a shorthand. Note that we introduce the shorthand as a separated construct in the "in a nutshell" section to make these sections easier to read.

$\mu t.G$ is a recursive type, and t is a type variable. We assume type variables are guarded in the standard way [CDYP16], meaning that type variables only appear under some interaction prefix. We use an equi-recursive approach to recursion i.e., $\mu t.G$ is equal to $G\{\mu t.G/t\}$. Lastly end expresses the end of a global type, which we sometimes omit from examples.

Figure 2.2 shows the two-buyer protocol expressed in this global type.

2.1.2. Local types

We now describe local types and projection, the process of extracting local types from global types (see Figure 2.1 for the top-down view of MPSTs).

Figure 2.6 depicts the local type syntax. The selection type $p!\{l_i(S_i).L_i\}_{i\in I}$ states that the process sends a message to p that carries a label l_i ($l_i \in \{l_i\}_{i\in I}$) and a value of type S_i; the processes communication continues as described in the selected branching case, i.e., L_i. The branching type $p?\{l_i(S_i).L_i\}_{i\in I}$ is the receiving counterpart. We may write

$p!\,l(S).\ L$ and $p?\,l(x).\ L$ for a selection type or branch type over a singleton label set. $\mu t.L$ is a recursive type, and t a type variable. As in the global type, recursion must be guarded, meaning type variables only appear under some interaction prefix, and we take an equi-recursive approach. Lastly, end expresses the end of a local type, which we sometimes omit from examples.

Figure 2.3 provides the local types for the seller s and the two buyers (b_1 and b_2) of the two-buyer protocol (Figure 2.2).

Projection

We now explain projection, i.e., the process of extracting local types from a global type.

Definition 1 (Projection ($g\upharpoonright p$)). *The projection of a global type G to a participant p is defined as follows:*

$$p_1 \to p_2\{l_i(S_i).G_i\}_{i\in I}\upharpoonright p = \begin{cases} p_2!\{l_i(S_i).G_i\upharpoonright p\}_{i\in I} & \textit{if } p = p_1 \\ p_1?\{l_i(S_i).G_i\upharpoonright p\}_{i\in I} & \textit{if } p = p_2 \\ G_1\upharpoonright p & \textit{if } \forall i,j \in I.G_i\upharpoonright p = G_j\upharpoonright p \end{cases}$$

$$(\mu t.G)\upharpoonright p = \{\mu t.(G\upharpoonright p) \ \ \textit{if } G\upharpoonright p \neq t \quad \textit{end} \ \ \textit{otherwise}\} \quad t\upharpoonright p = t \quad \textit{end}\upharpoonright p = \textit{end}$$

Otherwise it is undefined.

The first case handles branching types. If we project a branching type to the sender ($p = p_1$), we create a selection type and project all branching cases. Respectively, if we project to the receiver ($p = p_2$), we create a branching type. If we project neither to the sender nor the receiver, the projection of all branching cases must be identical ($\forall i,j \in I.G_i\upharpoonright p = G_j\upharpoonright p$), and we project the first branching case. The cases in the second line handle recursion and end. Lastly, if no rule is applicable, projection is undefined.

Figure 2.3 shows the result of projecting the two-buyer protocol (see Figure 2.2) to the two buyers and the seller.

2.1.3. Endpoint processes

Figure 2.7 provides the process syntax, which ranges over $P, P_1, \ldots$ Labels range over $l, l_1, \ldots$, expressions $e, e_i, ..$ can be values $v, v_i, \ldots$, variables $x, x_i, \ldots$, and standard operations.

An initialization process ($a[p](y).P$) states that this process wants to initialize a new session via the shared name a and play participant p; in P, the process uses the channel variable y, which gets replaced by a session channel during initialization for the session

P(Process)	$::=$	$a[p](y).P$	Initialization	a, b			Shared name
	$\parallel$	$c[p]!l(e).P$	Selection	x			Value Var
	$\parallel$	$c[p]?\{l_i(x_i).P_i\}_{i\in I}$	Branching	y, z			Channel Var
	$\parallel$	if e P else P	Condition	s			Session name
	$\parallel$	def D in P	Recursion	p, q			Participant id
	$\parallel$	$X\langle e, c\rangle$	Call	l			Label
	$\parallel$	0	Inaction	$s[p]$			Conc. Channel
	$\parallel$	$P \mid P$	Parallel	c	$::=$	$y \parallel s[p]$	Channel
	$\parallel$	$s : h$	Queue	m	$::=$	$\langle p, q, l(v)\rangle$	Message
	$\parallel$	$(\nu s)P$	Session hiding	v			Value
D	$::=$	$X(x, y) = P$	Declaration	e			Expression
E	$::=$	$[] \parallel P \mid E \parallel E \mid P$	Eval. context	h	$::=$	$h \cdot m \parallel \emptyset$	Queue
	$\parallel$	$(\nu s)E \parallel$ def D in E					

Figure 2.7.: Process syntax.

communication. A selection process $c[p]!l(e).P$ evaluates an expression e, sends the result together with label l to participant p over channel c, and continues as P. A channel is either a channel variable (y) before initialization or a named channel ($s[p]$). A branching process $c[p]?\{l_i(x_i).P_i\}_{i\in I}$ receives a value and a label from p over channel c; then continues as the P_i that corresponds to the received label; the received value is accessible via the value variable x_i.

A condition if e P_1 else P_2 evaluates an expression e to a boolean. If the result is true, the process proceeds as P_1, otherwise as P_2. A recursion def D in P_1 defines a declaration $X(x, y) = P_1$ over a value and a channel variable. A process call $X\langle e, c\rangle$ calls a declaration. 0 expresses an inactive process. Parallel composition $P_1 \mid P_2$ states both processes run in parallel.

A queue $s : h$ stores messages ($\langle p, q, l(v)\rangle$) in transit and a session hiding $(\nu s)P$ hides a session channel. Queues and hidings exist only during execution.

Reduction

Figure 2.8 shows selected reduction rules. Processes are considered modulo structural equivalence, and the reduction relies on the evaluation context E; we omit these details here and focus on a selection of reduction rules.

Remark. *We present fewer details from this section onwards and only explain the general concepts. We believe that focusing on the concepts is suffice as general background. In particular, as Chapters 3 and 4 are self-contained*

$$a[p_1](y_1).P_1 \mid ... \mid a[p_n](y_n).P_n \rightarrow (\nu s)(P_1\{s[p_1]/y_1\} \mid ... \mid P_n\{s[p_n]/y_n\} \mid s : \emptyset) \qquad \{p_1, .., p_n\} = \text{pid}(G) \qquad \textbf{(Link)}$$

$$s[p][q]!\, l(e).P \mid s : h \rightarrow P \mid s : h \cdot \langle p, q, l(v)\rangle \qquad e \Downarrow v \qquad \textbf{(Snd)}$$

$$s[p][q]?\, \{l_i(x_i).P_i\}_{i\in I} \mid s : \langle q, p, l_k(v)\rangle \cdot h \rightarrow P_k\{v/x_k\} \mid s : h \qquad k \in I \qquad \textbf{(Rcv)}$$

Figure 2.8.: Selected reduction rules.

The **(Link)** rule initializes a new session where the processes P_1 to P_n play the participants p_1 to p_n. The processes agree on shared name a, obeying to some global type that involves the participants p_1 to p_n. Together they start a private session s; this replaces the variable y_i in P_i with the channel $s[p_i]$ and creates a global queue $s : \emptyset$.

The **(Snd)** rule reduces a sending process $s[p][q]!\, l(e).\ P$ to P by emitting a message $\langle p, q, l(v)\rangle$ to the global queue $s : h$. The **(Rcv)** rule reduces a receiving process $s[p][q]?\, \{l_i(x_i).P_i\}_{i\in I}$ to P_k, if a message with label l_k from q is present at the head of the global queue; the message value gets substituted for x_k.

Example 1. *We now discuss a partial reduction of the two-buyer protocol. We start with the processes in Figure 2.4.*

$$\begin{aligned} P_s \mid P_{b_1} \mid P_{b_2} \xrightarrow{\textbf{(Link)}} \; & (\nu s)(s[s][b_1]?\, l_{title}(x).\ s[s][b_1]!\, l_{price}(220).\ s[s][b_2]!\, l_{price}(220).\ P'_s \\ & \mid s[b_1][s]!\, l_{title}(\text{"MPSTs"}).\ s[b_1][s]?\, l_{price}(x_{price}). \\ & \qquad s[b_1][b_2]!\, l_{share}(x_{price}/2).0 \\ & \mid s[b_2][s]?\, l_{price}(x_{price}).\ s[b_2][b_1]?\, l_{share}(x_{share}).\ P'_{b_2} \\ & \mid s : \emptyset) \end{aligned} \tag{2.1}$$

In Equation (2.1) *the three processes start a new session s, the first process plays the participant s, i.e. the seller, and its channel variable y is replaced by the channel $s[s]$, and similarly for the other two processes.*

$$\begin{aligned} (2.1) \xrightarrow{\textbf{(Snd)}} \; & (\nu s)(s[s][b_1]?\, l_{title}(x).\ s[s][b_1]!\, l_{price}(220).\ s[s][b_2]!\, l_{price}(220).\ P'_s \\ & \mid s[b_1][s]?\, l_{price}(x_{price}).\ s[b_1][b_2]!\, l_{share}(x_{price}/2).0 \\ & \mid s[b_2][s]?\, l_{price}(x_{price}).\ s[b_2][b_1]?\, l_{share}(x_{share}).\ P'_{b_2} \\ & \mid s : \langle b_1, s, l_{title}(\text{"MPSTs"})\rangle) \end{aligned} \tag{2.2}$$

In Equation (2.2) *the first buyer (b_1) sends the book title to the seller (s) and the corresponding*

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \rhd \Delta, \{y : G\lceil p\}}{\Gamma \vdash a[p](y).P \rhd \Delta} \qquad \frac{k \in I \quad \Gamma \vdash e : S_k \quad \Gamma \vdash P \rhd \{c : L_k\}, \Delta}{\Gamma \vdash c[p]!\, l_k(e).P \rhd \{c : p!\, \{l_i(S_i).L_i\}_{i \in I}\}, \Delta} \quad \lfloor \texttt{T-ini/T-snd} \rfloor$$

$$\frac{\Gamma \vdash P_1 \rhd \Delta_1 \quad \Gamma \vdash P_2 \rhd \Delta_2 \quad dom(\Delta_1) \cap dom(\Delta_2) = \emptyset}{\Gamma \vdash P_1 \mid P_2 \rhd \Delta_1, \Delta_2} \qquad \frac{\Gamma \vdash P \rhd \Delta \quad \Delta_s \text{ coherent}}{\Gamma \vdash (\nu s)P \rhd \Delta \setminus \Delta_s} \quad \lfloor \texttt{T-par/T-s} \rfloor$$

Figure 2.9.: Selected typing rules.

message is added into the queue ($s : \langle b_1, s, l_{title}(\text{"MPSTs"}) \rangle$).

$$(2.2) \quad \xrightarrow{\textbf{(Rcv)}} \quad \begin{array}{l} (\nu s)(s[s][b_1]!\, l_{price}(220).\ s[s][b_2]!\, l_{price}(220).\ P'_s\{\text{"MPSTs"}/x\} \\ \mid\ s[b_1][s]?\, l_{price}(x_{price}).\ s[b_1][b_2]!\, l_{share}(x_{price}/2).0 \\ \mid\ s[b_2][s]?\, l_{price}(x_{price}).\ s[b_2][b_1]?\, l_{share}(x_{share}).\ P'_{b_2} \\ \mid\ s : \emptyset) \end{array} \tag{2.3}$$

In Equation (2.3) *the seller receives a message carrying the book title and substitutes the title for* x*. The remaining reductions are similar, so we omit them.*

2.1.4. Type system

We now discuss type checking in MPSTs. Figure 2.9 shows selected typing rules, which have the following form

$$\Gamma \vdash P \rhd \Delta$$

stating P is well-typed by Δ under Γ. Γ is a shared environment, which tracks process variables ($X : S\,L$), content variables ($x : S$), and shared names ($a : G$). The session environment Δ tracks endpoint types ($c : L$) and queue types ($s : \mathtt{h}$).

We first discuss typing rules for endpoint processes, i.e., $\lfloor \texttt{T-snd} \rfloor$ and $\lfloor \texttt{T-ini} \rfloor$. After that, we explain typing rules for processes, i.e., $\lfloor \texttt{T-s} \rfloor$ and $\lfloor \texttt{T-pa} \rfloor$.

The typing rule $\lfloor \texttt{T-snd} \rfloor$

$$\frac{k \in I \quad \Gamma \vdash e : S_k \quad \Gamma \vdash P \rhd \{c : L_k\}, \Delta}{\Gamma \vdash c[p]!\, l_k(e).P \rhd \{c : p!\, \{l_i(S_i).L_i\}_{i \in I}\}, \Delta}$$

states that we can type a send process $c[p]!\, l(e).\ P$ on channel c if we can type the continuation, P, ($\Gamma \vdash P \rhd \{c : L_k\}, \Delta$) and message value expression e; with a selection type added, as a prefix, to the channel type c coming from typing the continuation ($c : L_k$),

i.e., $c : p! \{l_i(S_i).L_i\}_{i \in I}$. The selection types can contain cases not part of the send process since multiple send primitives implement selections, i.e., multiple send statements, in different branches of a condition statement, provide the implementation for a selection type (cf, P_{b_2} in Figure 2.4). The general idea of this typing rule and other typing rules for interaction processes is to type the continuations and add the process type as a prefix to the type derived from typing the continuations.

The typing rule $\lfloor$`T-ini`$\rfloor$ connects endpoint types with the corresponding global type. In detail, we can type an initialization process $a[p](y).P$, if P uses the channel variables y as defined by its local type, i.e., the local type derived by projecting the global type G associated with the shared name a to the participant p.

We now discuss the typing rules for processes. The first rule $\lfloor$`T-pa`$\rfloor$ types a parallel composition if we can type both processes ($\Gamma \vdash P_{1/2} \triangleright \Delta_{1/2}$) with session environments (Δ_1 and Δ_2) having disjointed domains. The last check is crucial, as it ensures the linear usage of session channels. The last typing rule $\lfloor$`T-s`$\rfloor$ types a session hiding $(\nu s)P$. It requires that we can type P ($\Gamma \vdash P \triangleright \Delta$) where the session environment, containing the endpoint types and queue type of s, is coherent.

Coherence, also called consistency [CDYP16], is the central typing invariant in MPSTs. We will discuss coherence after explaining the concepts on which it relies. The general idea is that two endpoint types in the same session have matching inputs and outputs, e.g., if one has an output, the other must have an input matching the output. Coherence must consider messages in transit, and that local types describe interactions towards multiple participants.

Queue types. MPSTs have typing rules for messages and queues, and their typing is straightforward. A message $\langle p, q, l(v) \rangle$, has the type $\langle p, q, l(S) \rangle$ where v has the type S. A queue $s : h$ has the type $s : \mathtt{h}$, where each message in the queue has a corresponding message type in the queue type.

Coherence

We now explain the required concepts for coherence. These include: extracting from a local type the interactions towards a specific participant; selecting message types from the queue type that describe messages between specific participants; calculating the effect messages have on a type; checking that inputs and outputs in one type match the outputs and inputs in another type based on the duality relation.

Partial projection ($L \upharpoonright p$). Partial projection, projects a local type L to a participant p by removing all selection and branching types from L which do not interact with p. For example, the first buyer's local type describes interactions with the participants b_2 and s.

$$L_{b_1} = s!\, l_{title}(\mathsf{str}).\ s?\, l_{price}(\mathsf{dbl}).\ b_2!\, l_{share}(\mathsf{dbl})$$

The partial projection to s ($L_{b_1} \upharpoonright s$) is $s!\, l_{title}(\mathsf{str}).\ s?\, l_{price}(\mathsf{dbl})$ and to b_2 is $b_2!\, l_{share}(\mathsf{dbl})$.

Heading-to ($(\mathtt{h})_{p \to q}$). Heading-to is similar to partial projection but for queue types. Given a queue type and participants p and q, heading-to removes all message types that do not describe messages sent from p and heading to q. For example, consider the queue type $\mathtt{h} = \langle p_1, p_2, l(S)\rangle \cdot \langle p_3, p_1, l'(S')\rangle \cdot \langle p_4, p_1, l'(S')\rangle$ then $(\mathtt{h})_{p_3 \to p_1}$ returns $\langle p_3, p_1, l'(S')\rangle$, the only message sent by p_3 and heading to p_1.

Session remainder $L - \mathtt{h}$. The session remainder calculates the effect message types have on a type; it subtracts branching prefixes from the type for which a message type is present. For example, assume the type $s?\, l_{price}(\mathsf{dbl}).\ \mathsf{end}$ of b_1 and a queue type containing $\langle s, b_1, l_{price}(\mathsf{dbl})\rangle$. The session remainder will return end; the branching type $s?\, l_{price}(\mathsf{dbl})$ consumes the message $\langle s, b_1, l_{price}(\mathsf{dbl})\rangle$.

Duality ($s[p] : L \bowtie s[p] : L$). The duality relation checks that two endpoint types in a session s have matching inputs and outputs. A selection type is dual to a branching type (and vice versa) if both use the same branching labels and data types, and all branching cases are dual. For example, the following selection type (of $s[b_2]$) $s[b_2] : s!\,\{l_{addr}(\mathsf{str}).\ s?\, l_{date}(\mathsf{str}).\mathsf{end}, l_{quit}(\mathsf{str}).\ \mathsf{end}\}$ is dual to the branching (of $s[s]$) $s[s] : b_2?\{l_{addr}(\mathsf{str}).\ b_2!\, l_{date}(\mathsf{str}).\mathsf{end}, l_{quit}(\mathsf{str}).\ \mathsf{end}\ \}$, since both use the same labels and data types ($l_{addr}(\mathsf{str})$, and $l_{quit}(\mathsf{str})$) and the branching cases are dual. Because, $s?\, l_{date}(\mathsf{str})$ is dual to $b_2!\, l_{date}(\mathsf{str})$ as they have the same label and data type, and describe no further interactions (end is dual to end).

Coherence. We say that a session environment Δ is coherent for a session s if every endpoint type is dual to every other endpoint type, taking into consideration partial projection, heading-to and session remainder. More formally, Δ is coherent for the session s if $s : \mathtt{h} \in \Delta$ and for all $s[p] : L, s[q] : L' \in \Delta$ the following holds:

$$s[p] : L \upharpoonright q - (\mathtt{h})_{q \to p} \bowtie s[q] : L' \upharpoonright p - (\mathtt{h})_{p \to q}$$

We now provide an example of a coherent session environment.

Example 2. *Assume a session environment*

$$\Delta = \left\{ \begin{array}{rl} s : \mathtt{h} & \textit{where } \mathtt{h} = \langle p_3, p_1, l_1(S_1) \rangle, \\ s[p_1] : L_1 & \textit{where } L_1 = p_3?\, l_1(S_1).\ p_2?\, l_2(S_2).\ \mathsf{end}, \\ s[p_2] : L_2 & \textit{where } L_2 = p_1!\, l_2(S_2).\ \mathsf{end}, \\ s[p_3] : L_3 & \textit{where } L_3 = \mathsf{end} \end{array} \right\}$$

Taking into account partial projection, heading-to, and session remainder we have

$$\begin{array}{lclcl}
s[p_1] : L_1 \downharpoonleft p_2 - (\mathtt{h})_{p_2 \to p_1} & = & s[p_1] : p_2?\, l_2(S_2) - \emptyset & = & s[p_1] : p_2?\, l_2(S_2) \\
s[p_1] : L_1 \downharpoonleft p_3 - (\mathtt{h})_{p_3 \to p_1} & = & s[p_1] : p_3?\, l_1(S_1) - \langle p_3, p_1, l_1(S_1) \rangle & = & s[p_1] : \mathsf{end} \\
 & & & & \\
s[p_2] : L_2 \downharpoonleft p_1 - (\mathtt{h})_{p_1 \to p_2} & = & s[p_2] : p_1!\, l_2(S_2) - \emptyset & = & s[p_2] : p_1!\, l_2(S_2) \\
s[p_2] : L_2 \downharpoonleft p_3 - (\mathtt{h})_{p_3 \to p_2} & = & s[p_2] : \mathsf{end} - \emptyset & = & s[p_2] : \mathsf{end} \\
 & & & & \\
s[p_3] : L_3 \downharpoonleft p_1 - (\mathtt{h})_{p_1 \to p_3} & = & s[p_3] : \mathsf{end} - \emptyset & = & s[p_3] : \mathsf{end} \\
s[p_3] : L_3 \downharpoonleft p_2 - (\mathtt{h})_{p_2 \to p_3} & = & s[p_3] : \mathsf{end} - \emptyset & = & s[p_3] : \mathsf{end}
\end{array}$$

Comparing the endpoint types, taking into account partial projection, heading-to, and session remainder, we have for the pair p_1 and p_2 that the branching type $p_2?\, l_2(S_2)$ is dual to the selection type $p_1!\, l_2(S_2)$. For any other pair we have that end *is dual to* end*. Therefore, the session environment Δ is coherent.*

2.1.5. Properties

The three main properties that MPSTs provide are: subject reduction, which provides communication safety as a corollary; fidelity; and progress.

Subject reduction Subject reduction ensures that a process that is well-typed in a coherent typing environment remains well-typed in a – potentially different – coherent typing environment after performing a reduction step.

The coherence invariant ensures that there is never a type mismatch between received messages and expected messages, i.e., it ensures communication safety.

Fidelity The global type accounts for any reduction that a well-typed process performs.

Progress A well-typed process can perform a reduction step or it has reached end. Progress typically forbids session interleaving [CDYP16].

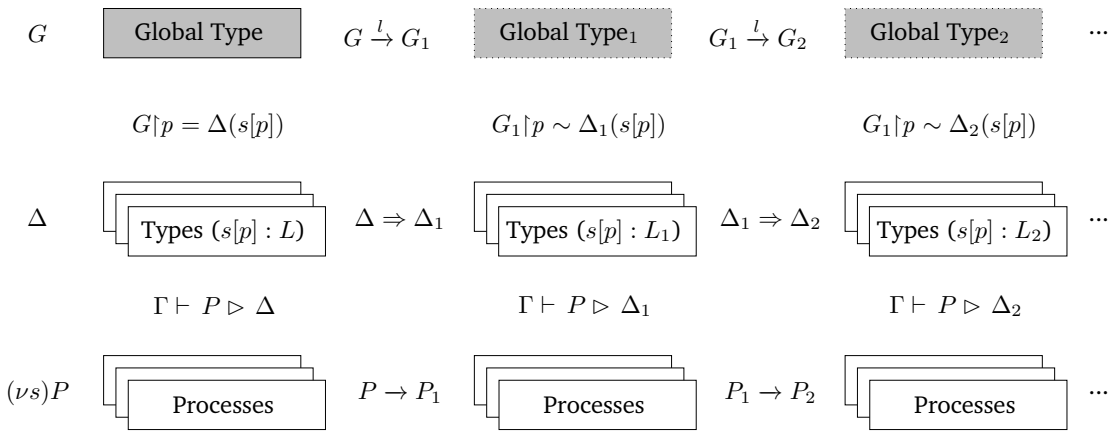

Figure 2.10.: Overview of the relation between different reduction relations in multiparty session types.

We now discuss the general idea of how to establish these properties. We rely on a typing environment reduction relation ($\Delta \Rightarrow \Delta'$) to establish subject reduction. The typing environment reduction describes the endpoint type and queue type changes that occur alongside a process reduction.[1] Similarly, to establish fidelity and progress, we rely on a global type reduction relation. Figure 2.10 visualizes the different reductions and their connection.

In Figure 2.10 (first column) we assume a process just after initialization of a new session s which follows a global type G. The process has the form $(\nu s)P$, i.e., it is a session hiding processes. The process P is well-typed in a session environment Δ and the endpoint type of a participant p is $s[p] : G\upharpoonright p$, i.e., projecting G gives us the endpoint types. By nature of its construction the initial session environment Δ is coherent (Performing projection and partial projection on a global type results in dual types, i.e., $s[p] : G\upharpoonright p \downharpoonleft q \bowtie s[q] : G\upharpoonright q \downharpoonleft p$).

To establish subject reduction we must ensure that a process performing a reduction, e.g., sending a message, remains well-typed in a coherent session environment. To do this, we rely on the typing environment reduction, which, e.g., reduce a selection type to one of its branching cases and adds a message type into the queue type. For any process reduction we show that: we can perform a typing environment reduction alongside the process reduction; the typing environment reduction preserves coherence; and the result

[1]The typing environment reduction relation is only required for the proofs. It is not part of a runtime reduction.

of the typing environment reduction types new processes.

In Figure 2.10 we see this in the bottom two rows. Initially P is well-typed by Δ under Γ, then we reduce P to P_1 by any reduction rule. We show that, regardless of the reduction rule used, we can perform a typing environment reduction from Δ to Δ_1 and that P_1 is well-typed by Δ_1 under Γ. Similarly, from P_1, respectively Δ_1, and so on.

The proofs in Chapters 3 and 4 follow this general idea and use structural induction over $P \to P$ to establish subject reduction.

Example 3. *We now demonstrate the concept of typing environment reduction with a concrete reduction of the two-buyer protocol. We start from* $P = s[b_1][s]!\,l_{title}(\text{"MPSTs"}).\ P'_{b_1} \mid s : \emptyset \mid P_{b_2} \mid P_s$ *which we can type* $\Gamma \vdash P \rhd \Delta$ *with* $\Delta = \{s[b_1] : s!\,l_{title}(\mathsf{str}).L'_{b1}, s : \emptyset\}, \Delta'$. *Assume* $P \to P_1$, *where* b_1 *sends the book title to the seller. Then* $P_1 = P'_{b_1} \mid s : \langle b_1, s, l_{title}(\text{"MPSTs"})\rangle \mid P_{b_2} \mid P_s$.

We can perform $\Delta \Rightarrow \Delta_1$ *where* $\Delta_1 = \{s[b_1] : L'_{b1}, s : \langle b_1, s, l_{title}(\mathsf{str})\rangle\}, \Delta'$. *It is easy to see that we have* $\Gamma \vdash P_1 \rhd \Delta_1$.

The coherence invariant is insufficient to establish fidelity and progress. For those following Coppo et al. [CDYP16], we introduce a labeled global type reduction relation ($G \xrightarrow{l} G'$). The global type reduction uses a slightly extended global type, e.g., it contains a fired interaction type, i.e., a type describing an interaction that sent a message that is not yet received. To establish fidelity, we show that when we make a typing environment reduction, we can perform a matching global type reduction. Projection connects the global type with the typing environment. The queue types need special consideration. Figure 2.10 shows this in the top two rows.

Once fidelity is established, progress follows more or less as a matter of course. The prefix interaction (branching type or fired interaction type) in the global type ensures that there is either a sending process, or a branching process and a message for the branching process is in the queue. In both cases, the process can do a step

2.2. Related Work

Failure, exceptions, and interrupts in session types. In the literature on MPSTs, there is one closely related work on failures. Adameit et al. [APN17] extend MPSTs with *optional blocks* to model communication link failures. Optional blocks allow an enclosed session fragment to be executed as standard, but they may also abruptly discard the ongoing execution non-deterministically. From the perspective of the protocol, optional blocks model a form of failure masking. Link failures are modeled by encapsulating the relevant interactions in appropriate optional blocks, as a kind of design pattern. Safety is attained by first requiring default values for input operations that may be skipped. Secondly, whenever execution discards the block of some role, the only action permitted of the peer roles is to discard their corresponding blocks. This is achieved by using *synchronous* channels, which may limit applicability to real-world distributed systems that are asynchronous. The authors argue that the main example, a rotating coordinator algorithm, can be considered as a distributed asynchronous process; because of its structure, e.g., it uses no choice and outputs have no continuation different than 0 [APN16]. The work does not provide an implementation nor a practical evaluation.

Fowler et al. [FLMD19] add an exception handling process primitive for failures in binary sessions. *Type*-level treatment of failure handling behaviors between the remaining (and new) peers, as in this thesis, cannot be studied in a binary setting. Structured interactional exceptions [CHY08] study exception handling for binary sessions. The work extends session types with a *try-catch* construct and a *throw* instruction, allowing participants to raise runtime exceptions. Global escape [CGY16] extends previous works on exception handling in binary session [CHY08] types to MPSTs. It supports nesting and sequencing of try-catch blocks with restrictions. Reduction rules for exception handling are of the form $\Sigma \vdash P \to \Sigma' \vdash P'$, where Σ is the *exception environment*. This central environment at the core of the semantics is updated synchronously and atomically. Furthermore, the reduction of a try-catch block to its continuation is done in a synchronous reduction step involving all participants in a block. Lastly, this work can only handle exceptions, i.e., explicitly raised application-level failures. These do not affect communication channels [CGY16], unlike participant crashes. Similarly, our previous work [CVB$^+$16] only deals with exceptions. An interaction $p \to q : S \vee F$ defines that p can send a message of type S to q. If F is not empty then, instead of sending a message, p can throw F. If a failure is thrown only participants that have casual dependencies on that failure are involved in the failure handling. No concurrent failures are allowed, so all interactions which can raise failures are executed in a lock step fashion. As a consequence, the model cannot be used to deal with crash-stop failures. Demangeon et al. [DHH$^+$15] study interrupts in MPSTs. This work introduces an interruptible block $\{|G|\}^c\langle l \text{ by } \mathtt{r}\rangle; G'$ identified by c; here the protocol G can be interrupted

by a message l from $\mathtt{r}$ and is continued by G' after either normal or interrupted completion of G. An interrupt is a control flow instruction like an exception, rather than an actual failure handling construct, and the semantics cannot model participant crashes. Miu et al. [MFYZ21] studies MPSTs for web programming, targeting TypeScript, in particular React.js[2] for the front end and Node.js[3] for the back end. The implementation presented requires the user to provide a cancellation handler that handles local exceptions and global session cancellation exceptions such as a disconnects [MFYZ21]. Such handlers neither provide partial failure handling, nor are they part of the formal model.

Neykova and Yoshida [NY17] show that MPSTs can be used to calculate safe global states for a safe recovery in Erlang's *let it crash* model [Arm03]. That work is well suited for recovery of lightweight processes in an actor setting. However, while it allows for elaborate failure handling by connecting (endpoint) processes with runtime monitors, the model does not address the fault-tolerance of runtime monitors themselves. As monitors can be interacting in complex manners, replication does not seem straightforwardly applicable, at least not without potentially hampering performance (just as with *straightforward* replication of entire applications).

Events in session types. Hu et al. [HKP+10] present a binary session calculus that allows encodings of EDP patterns. They add a non-blocking primitive for polling channels for messages and a typecase construct for sessions based on dynamic typing [ACPP91], whereas we model (in Chapter 4) event loops and handlers as first-class concepts. Their operational semantics maintains runtime types for channels for the typecase, not dissimilar to our concept of subprotocol state; unlike their dynamic channel typing, however, our subprotocol states are used to generalize the dynamic dispatch of event occurrences (e.g., in the special but not uncommon case of systems that dispatch solely on unique input labels, subprotocol states are unnecessary). Their approach is based on encoding event-driven behaviors in terms of these primitives, as opposed to our work that models first-class event loops. Their work does not consider failures. And specification of fault-tolerant, multiparty communication patterns, as we propose in Chapters 3 and 4, cannot be studied in a binary setting. Their work does not consider any of the technical challenges that we address for MPSTs, e.g., (partial) projections, coherence or fidelity. Cano et al. [CAP17] present a reactive binary calculus (without types) for reactive sessions.

Progress in MPSTs. Coppo et al. [CDYP16] develop an *additional* interaction type system on top of MPSTs to analyze global progress; the system that we propose in Chapter 4

[2]React.js. `https://reactjs.org`.
[3]Node.js `https://nodejs.org`.

offers global progress for concurrent (sub)sessions within MPSTs.

Subsessions in MPSTs. Our subprotocol and subsession concepts, which we use in Chapter 4, are inspired by the nested protocols of Demangeo and Honda [DH12]. We exploit subsessions to (i) incorporate a lightweight and practical facility for participant parametericity, and dynamic assigning of participants, and (ii) reason about the runtime structure of failure monitoring between subsessions. Their work does not consider event-driven concurrency nor failures, and their progress property is restricted to a single session (cf. the "simple" condition on their session typing environment).

Parameterization in MPSTs. Yoshida et al. [YDBH10, DYBH12] and Deniélou et al. [YDBH10] present a purely theoretical approach to parameterization in MPSTs. These works realize role parameterization via dependent types. Ng et al. [NdFCY15] use parameterized MPST [DYBH12] to generate C code for Message Passing Interface (MPI) programs. Their work produces completely centralized programs [CHJ+19]. Charalambides et al. [CDA16], present a parameterization for multi-actor computation extension to MPSTs, with a focus on repeated behavior, such as sliding window protocols. The work does not support role parametric protocols in the sense that two roles with different indexes can exchange messages (cf, projection [CDA16]). This forbids, for example, typical parameterized protocols, such as pipelines or rings. Castro et al. [CHJ+19] present a work for parameterized protocols with indexed roles. They support role-parametricity in MPST while maintaining both decidability and modularity of MPSTs. They provide a toolchain for MPST-based programming in Go. Unlike our work (in Chapter 4), none of the above works deals with partial failure or supports EDP. Furthermore, their treatment of parameterization tends to be quite technical, e.g., by relying on dependent types or SAT solving, whereas the lightweight parameterization in Chapter 4 does not rely on additions such as these.

Code generation for type checking of session types. Hu and Yoshida [HY16] proposed the use of global types in MPSTs to generate a type direct channel API to allow a hybrid verification of MPSTs in Java. Hybrid refers to the underlying compiler, here the Java compiler, that statically verifies the behavior part of session typing whereas the linearity part of session typing is performed dynamically. The general idea is to use a finite state machine (FSM) representation of the endpoint behavior in the protocol [DY12]. For each state, the framework generates a channel class with methods for the outgoing transitions, and those methods return a new channel for the state to which the transition leads and perform the corresponding session action, e.g., sending a message. This approach was used,

for example, in session type implementation in F# [NHYA18] and Go [CHJ+19]. Zhou et al. [ZFH+20] modify this approach to ensure linearity statically. Instead of exposing the channels to the user program, they require the user program to implement callback functions that are executed in the transition from one state to another. These callback functions, e.g., return a sent message but do not directly interact with the underlying channel. The underlying channel is never exposed to the user program, so the approach can ensure linear channel usage. The idea has similarities with the code generation approach of Ng et al. [NdFCY15], which generates centralized programs for MPI in C99. That work generates backbone MPI code and links it with user written sequential code. The work uses aspect-oriented programming to safely merge sequential user code with the generated MPI code.

There are further methods, besides code generation, that may be used to verify session types. These may be based on monads [OY17], monitors [CBD+11], or compiler extensions [HKP+10], for example.

The prototype implementation in Chapter 3 performs dynamic type checking using a monitor, similar to the work of Chen et al. [CBD+11]. The monitor checks that performed session actions are permissible. In Chapter 4, we build on the ideas of Hu and Yoshida [HY16] and extend them to EDP.

Typing invariant in MPSTs. Different approaches to compare the channel types exist in the MPST literature to ensure that processes can interact safely. The work of Coppo et al. [CDYP16] defines the consistent invariant (described as coherent in their conference publication [BCD+08]), for session channel typing environments. In essence, consistency is a generalization of duality from STs [HVK98]. STs have a concept of *co-types*. The co-type of a type T is written $\overline{T}$, where T is a channel type. E.g., the co-type of a sending type $!(S);T$ is a receiving type $?(S);\overline{T}$, similarly defined for other types. The main idea behind the generalization of duality is to introduce a second projection relation. This extracts – from a generalized type that consists of message types and a channel type – the types related to a specific participant, which is then related via duality. Honda et al. [HYC08, HYC16] use the coherent invariant. The main idea of this is that there must be a coherent global type[4] G. And for all participants in the global type G, the channel type of each participant equals the local type derived by projecting G onto that participant. Scalas and Yoshida [SY19] introduce a safety property for typing context that relates channel types via MPST-based behavioral type properties. This thesis builds on the concepts proposed by Coppo et al. [CDYP16].

[4] "We say G is coherent if it is linear and $G \upharpoonright p$ is well-defined for each $p \in \text{pid}(G)$" [HYC16](DEFINITION 4.2 (COHERENCE))

Branching restriction MPSTs. The projection of branching types in standard MPSTs [CDYP16, HYC16] is somewhat limiting. A participant, which is neither sender nor receiver, may not occur in any continuation or must perform the exact same behavior in all continuations of a branching statement. The merge operator [CYH09a, YDBH10] enables more flexible interaction patterns. Scalas et al. [SDHY17b] provide a careful treatment of the merge operator and fix an error in the classic definition. Hu and Yoshida [HY17] propose a model checking approach for the validation of safety and progress properties in MPSTs that also enables branching patterns not expressible in classic MPSTs. We use restricted projection since expressing more flexible branching patterns is an interesting but orthogonal research question.

Verification of distributed systems outside of session types Failure handling is studied in several process calculi and communication-centered programming languages without typing disciplines. The conversation calculus [VCS08] models exception behavior in abstract service-based systems with message-passing-based communication. The work does not use channel types but studies the behavioral theory of bisimilarity. Error recovery is studied in a concurrent object setting [XRR+95]; interacting objects are grouped into coordinated atomic actions (CAs), enabling safe error recovery. CAs cannot be nested, however. PSYNC [DHZ16] is a domain-specific language based on the *heard-of* model of distributed computing [CBS09]. Programs written in PSYNC are structured into rounds that are executed in a lock step manner. PSYNC comes with a state-based verification engine that enables the checking of safety and liveness properties; programmers have to define non-trivial inductive invariants and ranking functions.

Model-checking can automatically verify or partially verify distributed systems and protocols. However, complete verification is often not feasible because of underlying undecidabilities, i.e., model checkers may find errors but usually can not guarantee the absence of those errors. They can, for example, transparently check implementations [YCW+09] or be combined with a programming model tailored to the model checker [KAB+07]. Konnov et al. [KLVW17] introduce a model checker for a restricted class of distributed protocols, expressed as threshold automata. For those protocols, they can automatically verify safety and liveness properties.

Mechanized verification is used to establish properties for distributed systems. Examples include Verdi [WWP+15] and IronFleet [HHK+15]. Verdi [WWP+15] is a framework for implementing and verifying distributed systems in Coq. It provides the possibility of verifying the system against different network models. Verdi enables the verification of properties in an idealized fault model and then transfers the guarantees to more realistic fault models by applying transformation functions. Verdi supports safety properties but

not liveness properties. IronFleet [HHK+15] enables the implementation of distributed systems and the process of proving them correct. They use a combination of TLA-style state-machine refinement and Hoare-logic verification – the first for protocols including concurrency and ignoring the complexity of the implementation, the latter for verification of the implementation while ignoring the concurrency. They can establish both safety and liveness properties of distributed systems. More recent works on verification for distributed systems, e.g., Sergey et al. [SWT18] and Taube et al. [TLM+18], try to reduce the burden on engineers through the use of modularity. However, the burden of creating proofs remains huge. Sergey et al. [SWT18] still require a proof of almost $3,000$ lines of code for their two-phase commit protocol. And Taube et al. [TLM+18] state that the verification of the Raft protocol, which is well understood, took them approximately 3 person-months. We believe that complete verification is an auspicious direction for future research, but the amount of proof engineering required is still prohibitive in many cases.

3. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed System

This chapter is based on our work, "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems", which appeared at the European Symposium on Programming 2018 [VCE+18]. The chapter presents for significant parts that work *directly*, i.e., the chapter is for significant parts a verbatim copy of our article. It introduces a formal model for crash failure handling in asynchronous distributed systems. This model features a lightweight coordinator modeled along the lines of widely-used systems such as Apache ZooKeeper and Chubby. We develop, for this model, a typing discipline based on multiparty session types that supports the specification and static verification of multiparty protocols with explicit failure handling. We show that our type system ensures subject reduction in the presence of failures. The publication on which this chapter is based, provides a progress property for our formal model. In other words, in a well-typed system, even if some participants crash during execution, the system is guaranteed to progress in a consistent manner with the remaining participants. Furthermore, we present a prototype based on the formal model that uses ZooKeeper for coordination and is written in Scala. The accompanying performance evaluation shows no real performance impact in the absence of failures.

3.1. Introduction

Networked distributed systems - like web services and cloud-based data analytics - are becoming omnipresent.

3.1.1. Distributed programs, partial failures, and coordination

Nevertheless, developing programs that execute across a set of physically remote, networked processes is challenging. The correct operation of such a *distributed program* requires correctly designed protocols, which govern the asynchronous interaction of concurrent processes, and processes, which are correctly implemented in accordance with their roles in the protocols. This becomes particularly challenging when distributed programs have to be resilient to *partial failures*, where some processes crash while others remain operational. Partial failures affect both *safety* and *liveness* of applications. Asynchrony is the key issue in the handling of such failures, resulting in the inability to distinguish slow processes from failed ones. In general, this makes it impossible for processes to reach agreement, even when only a single process may crash [FLP85].

In practice, such impasses are overcome by making appropriate assumptions about the considered infrastructure and applications. One common approach is to assume the presence of a highly available *coordination service* [Hun10] – realized using a set of replicated processes large enough to survive common rates of process failures (e.g., 1 out of 3, 2 out of 5) – and delegating critical decisions to this service. While this *coordinator model* has been in widespread use for many years (cf. *consensus service* [GS01]), the advent of cloud computing has recently brought it further into the mainstream, via instances such as Chubby [Bur06] and Apache ZooKeeper [Hun10]. Such systems are used not only by end user applications but also by a variety of frameworks and middleware systems across the layers of the protocol stack [CDG+06, GGL03, KNR11, SKRC10].

3.1.2. Typing disciplines for distributed programs

Typing disciplines for distributed programs constitute a promising and active research area towards addressing the challenges in the correct development of distributed programs. See Hüttel et al. [HLV+16] for a broad survey. *Session types* are one of the established typing disciplines for message passing systems. Originally developed in the π-calculus [HVK98], these have subsequently been successfully applied to a broad range of practical languages, e.g., Java [HY16, SQZ+13], Scala [SDHY17a], Haskell [LM16, PT08], and OCaml [IYY17, Pad17].

Multiparty session types (MPSTs) [CDYP16, HYC16] is a generalization of session types to more than two participants. In a nutshell, a standard MPST framework takes (1) a specification of the whole multiparty message protocol as a *global type*, from which (2) *local types*, describing the protocol from the perspective of each participant, are derived; these are in turn used to (3) statically *type check* the I/O actions of endpoint programs implementing the session participants. A well-typed system of session endpoint programs

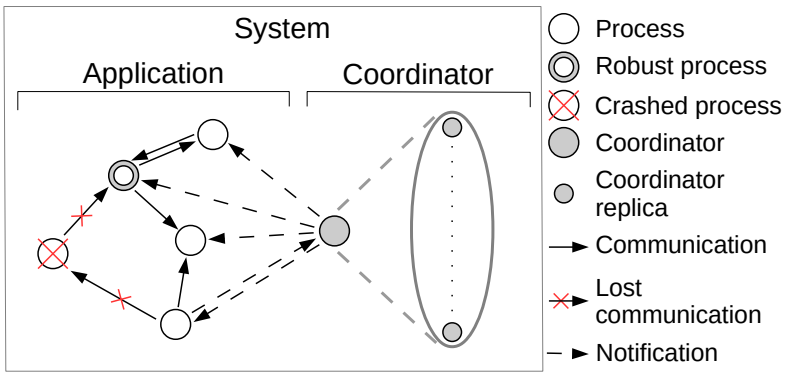

Figure 3.1.: Coordinator model for asynchronous distributed systems. The coordinator is implemented by replicated processes (internals omitted).

enjoys important safety and liveness properties, such as *no reception errors* (only expected messages are received) and *session progress*. A basic intuition behind MPSTs is that the design (i.e., restrictions) of the type language constitutes a class of distributed protocols for which these properties can be statically guaranteed by the type system.

Unfortunately, *no* MPST work supports protocols for asynchronous distributed programs dealing with *partial failures due to process crashes*, so the aforementioned properties no longer hold in such an event. Several MPST works have treated communication patterns based on *exception messages* (or *interrupts*) [CGY16, CHY08, DHH+15]. In these works, such messages may convey exceptional states in an *application* sense; from a protocol compliance perspective, however, these messages are the same as any other messages communicated during a *normal* execution of the session. This contrasts with *process* failures, which may invalidate messages which are already in-transit (*orphan* messages), and where the task of agreeing on the concerted handling of a crash failure is itself prone to such failures.

Outside of session types and other type-based approaches, there have been a number of advances in the verification of fault tolerant distributed protocols and applications (e.g., based on model checking [KAB+07] or proof assistants [WWP+15]); however, little work exists on providing direct compile-time support for *programming* such applications in the spirit of MPSTs.

3.1.3. Contributions and challenges

This chapter puts forward a new typing discipline for safe specification and implementation of distributed programs prone to process crash failures based on MPSTs. The following summarizes the key challenges and contributions.

Multiparty session calculus with coordination service. We develop an extended multiparty session calculus as a formal model of processes prone to crash failures in asynchronous message passing systems. Unlike standard session calculi that reflect only "minimal" networking infrastructures, our model introduces a practically-motivated *coordinator* artifact and explicit, asynchronous messages for run-time crash notifications and failure handling.

MPSTs with explicit failure handling. We introduce new global and local type constructs for *explicit failure handling*, designed for the specification of protocols that tolerate partial failures. Our type system carefully reworks many of the key elements in standard MPSTs to manage the intricacies of handling crash failures. These elements include the well-formedness of failure-prone global types, and the crucial *coherence* invariant for MPST typing environments to reflect the notion of system consistency in the presence of crash failures and the resulting errors. We show safety for a well-typed MPST session despite potential failures.

Prototype and performance evaluation. We develop a prototype based on our model in Scala that uses Apache ZooKeeper as its coordination service. We realize a session-type logistic regression (LR) model in it. In our evaluation, the session-type logistic regression (LR) has a performance similar to the non-failure-aware baselines in the non-failure cases.

To apply our model in practice, we introduce programming constructs similar to well-known and intuitive exception handling mechanisms, for handling concurrent and asynchronous process crash failures in sessions. These constructs serve to integrate user-level session control flow in endpoint processes and the underlying communications with the coordination service, which is used by the target applications of our work to outsource critical failure management decisions (see Figure 3.1). It is important to note that the coordinator does *not* magically solve all problems. Key design challenges are to ensure that communication with the coordinator is fully asynchronous as in real-life, and that it is involved only in a "minimal" fashion. Thus we treat the coordinator as a first-class, asynchronous network artifact, as opposed to a convenient but impractical global "oracle" (cf. [CGY16]), and our operational semantics of multiparty sessions remains primarily

$$\left(\begin{array}{ll} \mu t. & \{w_1\} : \mu t_1.drv{\rightarrow}w_2\ l_{pointsw_1}(S). \\ \quad drv{\rightarrow}w_1\ l_{points}(S).drv{\rightarrow}w_2\ l_{points}(S). \blacktriangleright & \qquad w_2{\rightarrow}drv\ l_{gradw_1}(S').t_1, \\ \quad w_1{\rightarrow}drv\ l_{grad}(S').w_2{\rightarrow}drv\ l_{grad}(S').t & \{w_2\} : ..., \{w_1, w_2\} : \mathsf{end} \end{array}\right)^1$$

Figure 3.2.: A top-level global type $[drv]G$ that describes a distributed logistic regression model with failure handling capabilities.

choreographic in the original spirit of distributed MPSTs, unlike works that resort to a centralized *orchestrator* to conduct all actions [CP16, CLM+16]. As depicted in Figure 3.1, application-specific communication does not involve the coordinator. Our model lends itself to common practical scenarios, in which processes monitor each other in a peer-based fashion to detect failures, and rely on a coordinator only to establish agreement on which processes have failed, and when.

3.1.4. Example

Figure 3.2 provides a communication specification, as a global type, for a logistic regression (LR) model. The model has a structure similar to many other big data tasks: Namely, divide data into chunks, distribute data chunks, calculate partial results, aggregate partial results, and optionally repeat previous steps. For example, distributed algorithms, such as word count, distributed π, or K-means clustering, also follow this scheme. The distributed LR model is an iterative algorithm, and in every iteration, it performs the following: the driver splits the data points into chunks and sends the chunks and model parameter to the workers; the workers calculate the partial model update and send the result to the driver; the driver updates the model parameter. The global type in Figure 3.2 involves a driver, drv, and two workers, $w_{1,2}$. We want to focus on the novel failure handling, so we intentionally keep the global type simple. In particular: We use only two workers; We do not specify the concrete message types; We assume that the driver has built-in fault tolerance mechanisms, i.e., we consider the driver, drv, to be *robust*, denoted by the annotation $[drv]$; The workers, however, may fail individually. The LR model specification can be easily extended to more than two workers or a non-robust driver.

We now explain the global type in more detail. In the *try-handle* construct $\big((a) \blacktriangleright (b)\big)^1$, the *try-block* (a) gives the *normal* (i.e., failure-free) flow of the protocol, and (b) contains the explicit *handlers* for potential failures. In the try-block, the global type specifies that the driver sends the model parameter and data points to the individual workers ($drv{\rightarrow}w_i\ l_{points}(S)$), and then each worker sends the calculated update back to the driver ($w_1{\rightarrow}drv\ l_{grad}(S')$). If a worker fails by crashing, the corresponding failure handling

($\{w_i\}$: ...) will take over. In it the non-failed worker performs the computation for both workers, enabling the system to produce a valid result in spite of the failure. If both workers fail (by any interleaving of their concurrent failures), the global type specifies that the driver should safely terminate its role in the session.

We shall refer to this basic example, that focuses on the new failure handling constructs, in explanations in later sections. See Appendix A.1 for examples of larger protocols featuring multiparty choices and recursion with explicit failure handling. We also give many further examples throughout the following sections. These illustrate the potential session errors due to failures exposed by our model and the way in which our framework resolves the errors in order to recover MPST safety.

3.1.5. Roadmap

Section 3.2 describes the adopted system and failure model. Section 3.3 introduces global types for guiding failure handling. Section 3.4 introduces our process calculus with failure handling capabilities and a coordinator. Section 3.5 introduces local types, derived from global types by projection. Section 3.6 describes typing rules, and defines *coherence* of session environments with respect to endpoint crashes. Section 3.7 states the properties of our model. Section 3.8 presents our prototype implementation. Section 3.9 discusses related work. Section 3.10 draws conclusions.

3.2. System and Failure Model

In distributed systems, care is required to prevent partial failures from adversely affecting the liveness (e.g., waiting for messages from crashed processes) or safety (e.g., when processes manage to communicate with some peers but not others before crashing, thus leading to inconsistencies) properties of applications. Based on the nature of the infrastructure and application, appropriate *system and failure models* are chosen along with judiciously made assumptions to overcome such impasses in practice.

We pinpoint the key characteristics of our model, according to our practical motivations and standard distributed systems literature, that shape the design choices we make later regarding the process calculus and types. As is common practice, we augment our system with a *failure detector (FD)* to allow for distinguishing slow and failed processes. The advantages of the FD are that: (i) in terms of reasoning, it concentrates all assumptions to solve given problems, and (ii) in terms of implementation, it yields a single main module in which time-outs are set and used.

In our work, we make the following concrete assumptions about failures, the system, and FDs:

(1) **Crash-stop failures**: Application processes fail by crashing (halting), and do not recover.

(2) **Asynchronous system**: Application processes and the network are asynchronous, meaning that there are no upper bounds limiting processes' relative speeds or message transmission delays.

(3) **Reliable communication**: Messages transmitted between correct (i.e., non-failed) participants are eventually received.

(4) **Robust coordinator**: The coordinator (coordination service) is permanently available.

(5) **Asynchronous reliable failure detection**: Application processes have access to local FDs which eventually detect all failed peers and do not falsely suspect peers.

Items (1), (2) and (5) are standard in the literature on fault-tolerant distributed systems [FLP85]. Note that processes can still recover but will not do so *within* sessions (or will not be re-considered for those sessions). Other failure models, e.g., crash-recovery [ACT00][1], network partitions [GL02] or Byzantine failures [LSP82], are interesting subjects of future work. Note that network partitions are not tolerated by ZooKeeper et al., and that Byzantine failures have often been argued to be a too generic failure model (e.g., [Bir17]).

Assumption (4) about the coordinator is in line with a large number of applications. For example, the data analytics frameworks Hadoop [Apa06] and Spark [ZCD+12a], the non-relation distributed database HBase [Apa08], or the resource managements Mesos [HKZ+11] and YARN [VMD+13a] use the ZooKeeper [Hun10] coordination service by default or as an option for coordination/fault tolerance. The assumption that the coordinator is robust implicitly means that the number of concomitant failures among the coordinator replicas is assumed to remain within a minority and that failed replicas are replaced in time to tolerate further failures. Without loss of validity, the coordinator internals can be treated as a black box.

The final Item (4) regarding failure detection is backed in practice by the concept of *program-controlled* crash [CHTCB96], which consists in communicating decisions to disregard supposedly failed processes also to those very processes, prompting them to reset

[1] This is different from simply assuming that crashed processes transparently recover without disrupting an application.

$$
\begin{array}{llll}
\text{(Basic type)} & S & ::= & \mathsf{bool} \mid \mathsf{str} \mid \mathsf{int} \\
\text{(Global type)} & G & ::= & p \to q\{l_i(S_i).G_i\}_{i\in I} \mid \mu t.G \mid t \mid \mathsf{end} \mid \left(G \blacktriangleright H^{\uparrow}\right)^{\kappa}.G \\
\text{(Failure block)} & H^{\uparrow} & ::= & F : G\,,\ H^{\uparrow} \mid \emptyset \qquad \text{(Failure set)}\ F ::= \{p_i\}_{i\in I}
\end{array}
$$

Figure 3.3.: Syntax of global types with explicit handling of partial failures.

themselves upon false suspicion. In practice, systems can be configured to minimize the probability of such events, by a "two-level" membership that involves evicting processes from *individual* sessions (cf. recovery above) more quickly than from the system as a whole; several authors have also proposed network support to avoid false suspicions altogether (e.g., [LWH+11]).

These assumptions do not make handling of failures trivial, let alone mask failures. For instance, the network can arbitrarily delay messages and thus reorder them with respect to their real sending times. Different processes can therefore detect failures at different points in time and in a different order. This means that our approach has to overcome the difficulty that processes have to agree on which failures to handle, and when and how to handle them.

3.3. Global Types for Explicit Handling of Partial Failures

Based on the foundations of MPSTs, we develop *global types* to formalize specifications of distributed protocols with explicit handling of *partial failures due to role crashes*, simply referred to as *failures*. We present global types before introducing the process calculus to provide a high degree of insight into how failure handling works in our model.

3.3.1. Global types syntax

The syntax of *global types* is depicted in Figure 3.3. We use the following base notation: $p, q, \ldots$ for *role* (i.e., participant) names; $l_1, l_2, \ldots$ for message *labels*; and $t, t', \ldots$ for type variables. *Base types* $S, S_1, \ldots$ include primitive types such as bool or int.

Global types are denoted by G. We first summarize the constructs from standard MPSTs ([CDYP16, HYC16]). A *branch* type $p \to q\{l_i(S_i).G_i\}_{i\in I}$ means that p can send to q *one* of the messages of type S_k with label l_k, where k is a member of the non-empty index set I. The protocol then proceeds according to the continuation G_k. When I is a singleton, we may simply write $p{\to}q\ l(S).G$. We use t for type variables and take an equi-recursive view, i.e., $\mu t.G$ and its unfolding $[\mu t.G/t]$ are equivalent. We assume type

variable occurrences are bound and guarded (e.g., $\mu t.t$ is not permitted). end is for termination.

We now introduce our extensions for partial failure handling. A *try-handle* $\left(G_1 \blacktriangleright H^{\uparrow}\right)^{\kappa}.G_2$ describes a "failure-atomic" protocol unit: all *live* (i.e., non-crashed) roles will eventually reach a consistent protocol state, despite any concurrent and asynchronous role crashes. The try-block G_1 defines the *default* protocol flow, and $H^{\uparrow}$ is a *handling environment*. Each element of $H^{\uparrow}$ maps a *handler signature* F, that specifies a *non-empty* set of *failed* roles $\{p_i\}_{i \in I}$, to a *handler body* specified by a G. The handler body G specifies how the live roles should proceed given the failure of roles F. The protocol then proceeds (for live roles) according to the continuation G_2 after the default block G_1 or failure handling defined in $H^{\uparrow}$ has been completed as appropriate. We allow nesting of try-handles, if the inner try-handle occurs in the default block of outer try-handles, but we do not allow try-handles to occur inside a handling environment.

To simplify later technical developments, we annotate each try-handle term in a given G by a unique $\kappa \in \mathbb{N}$ that lexically identifies the term within G. These annotations may be assigned mechanically. As a shorthand, we refer to the try-block and handling environment of a particular try-handle by its annotation; e.g., we use κ to stand for $\left(G_1 \blacktriangleright H^{\uparrow}\right)^{\kappa}$. In the running examples, if only one try-handle exists, we may omit κ for simplicity.

We define the global context $\mathcal{G}$ as follows:

$$\begin{array}{rcl} \mathcal{G} & ::= & [\,] \mid \left(\mathcal{G} \blacktriangleright H^{\uparrow}\right)^{\kappa}.G \mid \left(G \blacktriangleright F : \mathcal{G}, H^{\uparrow}\right)^{\kappa}.G' \mid \\ & & \left(G \blacktriangleright H^{\uparrow}\right)^{\kappa}.\mathcal{G} \mid p \to q\{l_i(S_i).G_i\}_{i \in I} \cup \{l_j(S).\mathcal{G}\} \mid \mu t.\mathcal{G} \end{array}$$

We define the global context in the standard way, i.e., a global context is a global type with a hole. In more detail a context is a hole $[\,]$, a default block context $\left(\mathcal{G} \blacktriangleright H^{\uparrow}\right)^{\kappa}.G$, a handler body context $\left(G \blacktriangleright F : \mathcal{G}, H^{\uparrow}\right)^{\kappa}.G'$, a try-handle continuation context $\left(G \blacktriangleright H^{\uparrow}\right)^{\kappa}.\mathcal{G}$, a branching context $p \to q\{l_i(S_i).G_i\}_{i \in I} \cup \{l_j(S).\mathcal{G}\}$, or a recursion context $\mu t.\mathcal{G}$. Such contexts allow us to reason about parts of global types. Based on these contexts, we define a containment relation on global types as follows:

Definition 2 ($G' \in G$)**.** *If* $\exists \mathcal{G}$ *s.t.* $G = \mathcal{G}[G']$*, then* $G' \in G$

$G' \in G$ means G' is a part of global type G. Analogous to $G' \in G$, we write $G \in H^{\uparrow}$ if the handling environment $H^{\uparrow}$ contains G; $\kappa \in G$ if G contains κ (remember κ is shorthand for $\left(G_1 \blacktriangleright H^{\uparrow}\right)^{\kappa}$); $\kappa \in \kappa'$ if the try-handle κ contains the try-handle κ'; $l \in G$ if the label l appears inside G; and $l \in \mathcal{G}$ if the label l appears inside $\mathcal{G}$. We use a lookup function $outer_G(\kappa)$ for the set of all try-handles in G that enclose a given κ (including κ itself).

Definition 3 ($outer_G(\kappa)$)**.** $outer_G(\kappa) = \{\kappa' \mid \kappa \in \kappa' \wedge \kappa' \in G\}$.

Top-level global types and robust roles We use the term *top-level* global type to mean the source protocol specified by a user, following a typical top-down interpretation of MPST frameworks [CDYP16, HYC16]. We allow top-level global types to be optionally annotated $[\tilde{p}]G$, where $[\tilde{p}]$ specifies a set of *robust* roles—i.e., roles that can be assumed to never fail. In practice, a participant may be robust if it is replicated or is made inherently fault tolerant by other means (e.g., the participant that represents the driver in Figure 3.2).

3.3.2. Well-formedness

Not every syntactically correct global type describes a safe communication protocol. Therefore, the first stage of validation in standard MPSTs is to check that the top-level global type satisfies the supporting criteria used to ensure the desired properties of the type system. MPSTs require global types to be projectable and they validate global types in projection [HYC16, CDYP16]. For example, MPSTs reject the following branching pattern, since it defines an unsafe branching pattern.

$$a \to b\{l_1(S).\, c{\to}a\; l(S),\; l_2(S).\, c{\to}a\; l'(S)\}$$

The issue is that c does not know if a selects l_1 or l_2 but c must send l to a if a selects l_1, or l' if a selects l_2. Therefore, this protocol gets rejected during projection in standard MPSTs [HYC16, CDYP16] and in this work (see Definition 10 branching case). We add well-formedness criteria for failure handling that is not present in standard MPSTs.

We now define well-formedness below:

Definition 4 (Well-formedness)**.** *Let κ stand for $\left(G_1 \blacktriangleright H^{\uparrow}\right)^{\kappa}$, and κ' for $\left(G_1' \blacktriangleright H^{\uparrow'}\right)^{\kappa'}$. A top-level $[\tilde{p}]G$ is* well-formed *if it fulfills* all *the following conditions. For all $\kappa \in G$:*

1. *For any two separate handler signatures of a handling environment of κ, there is always a handler whose handler signature matches the union of the respective failure sets. This handler is either inside the handling environment of κ itself, or in the handling environment of an outer try-handle:*

$$\forall F_1 \in \mathit{dom}(H^{\uparrow}).\forall F_2 \in \mathit{dom}(H^{\uparrow}).\exists \kappa' \in \mathit{outer}_G(\kappa) \text{ s.t. } F_1 \cup F_2 \in \mathit{dom}(H^{\uparrow'})$$

2. *If the handling environment of a try-handle κ contains a handler for F, then there is no outer try-handle κ' with a handler for F' such that $F' \subseteq F$:*

$$\nexists F \in \mathit{dom}(H^{\uparrow}).\exists \kappa' \in \mathit{outer}_G(\kappa).\exists F' \in \mathit{dom}(H^{\uparrow'}) \text{ s.t. } \kappa' \neq \kappa \wedge F' \subseteq F$$

3. *All κ in G are unique. In addition, all labels which appear inside a default try-body or any handler body do not occur outside of the default block/the handler body:*

$$G = \mathcal{G}[(G_1 \blacktriangleright H^{\uparrow})^{\kappa}.G_2] \Rightarrow \forall l \in G_1.l \notin \mathcal{G}, G_2, H^{\uparrow} \text{ and } \\ \forall F, F' \in H^{\uparrow}. \text{ s.t. } F \neq F' \; \forall l' \in H^{\uparrow}(F).l' \notin \mathcal{G}, G_1, G_2, H^{\uparrow}(F')$$

4. *A role does not appear in the handling activity of its own failure:*

$$\forall F \in dom(H^{\uparrow}).\forall p \in F \Rightarrow p \notin roles(H^{\uparrow}(F))$$

5. *All branching types of non-robust participants ($p \notin \tilde{p}$) must be handled in G, i.e., must be enclosed by try-handles which can handle the potential failures of p:*

$$\forall q \to q'\{l_i(S_i).G_i\}_{i \in I} \in G.\forall p \in \{q, q'\}.p \notin \tilde{p} \Rightarrow \\ \exists \kappa' \in G.q \to q'\{l_i(S_i).G_i\}_{i \in I} \in \kappa' \wedge \{p\} \in dom(H^{\uparrow'})$$

Condition 1 ensures that if roles are active in different handlers of the same try-handle, there is a handler whose signature corresponds to the union of the signatures of those different handlers. Example 7 and Example 8 in Section 3.4 show why this condition is needed. The reason for condition 2 is that, in the case of nested try-handles, the operational semantic (see **(TryHdl)** in Section 3.4, Figure 3.8) allows multiple try-handles to start failure handling; eventually the outermost try-handle will perform the handling and may interrupt failure handling at an inner try-handle. The reason for condition 3 is that a try-handle is our handling basis, and we shall not confuse normal messages or done notifications from different try-handles. Condition 4 is straightforward. Condition 5 is also straightforward, because we can only handle failures occurring in a try-handle, so interaction that involves non-robust roles needs to be inside try-handles.

We use the following examples to illustrate non-well-formed global types.

Example 4. $G_{ko} = ((G' \blacktriangleright \{p_1\} : G_1)^2 \blacktriangleright \{p_1\} : G_1')^1$ *violates condition 2 because there are two different handling activities for $\{p_1\}$ at different nesting levels. Since the outer try-handle will eventually take over, it is not sensible to have the handling activity $\{p_1\}$ at the inner nesting level.*

Example 5. $G_{ko} = (p_1 \to p_2 \; l_1(S_1) \blacktriangleright \{p_1\} : p_1 \to p_2 \; l_2(S_2))$ *violates condition 4. It is not well-formed since the handling activity of $\{p_1\}$ contains $p_1 \to p_2 \; l_2(S_2)$ in which p_1 is expected to output a message, yet p_1 would have failed at that point.*

Example 6. $G_{ko} = (p_1 \to p_2 \; l_1(S_1) \blacktriangleright \{p_1\} : p_2 \to p_3 \; l_3(S_3)).p_1 \to p_3 \; l_2(S_2)$ *violates condition 5, since the non-robust p_1 is expected to perform a message send $p_1 \to p_3 \; l_2(S_2)$ which is not enclosed by any try-handle. This is not safe because, if p_1 crashes before $p_1 \to p_3 \; l_2(S_2)$ completes, then p_3 will get stuck.*

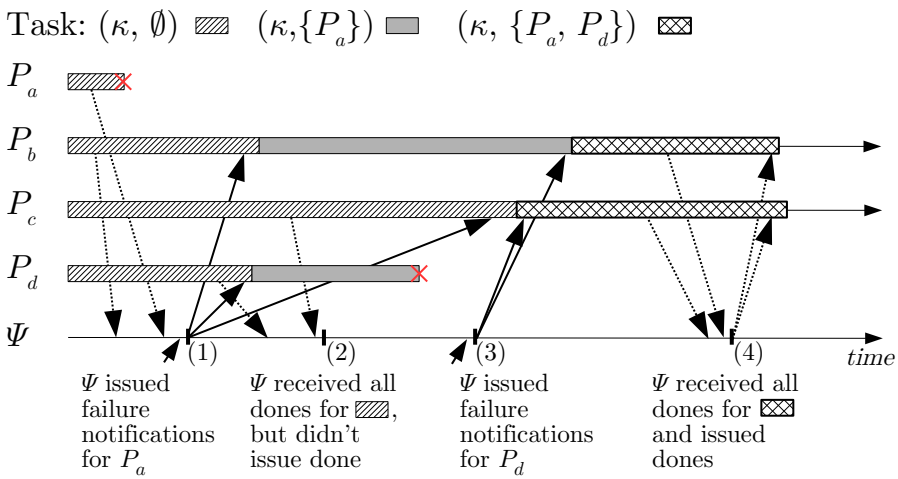

Figure 3.4.: Challenges under pure asynchronous interactions with a coordinator. Between time (1) and time (2), the task $\phi = (\kappa, \emptyset)$ is interrupted by the crash of P_a. Between time (3) and time (4), due to asynchrony and multiple crashes, P_c starts handling the crash of $\{P_a, P_d\}$ without handling the crash of $\{P_a\}$. Finally after (4) P_b and P_c finish their common task.

3.4. A Process Calculus for Coordinator-based Failure Handling

In this section we introduce our calculus for describing processes implementing our global types with failure handling. Figure 3.4 depicts a scenario that can occur in practical asynchronous systems with coordinator-based failure handling through frameworks such as ZooKeeper (Section 3.2). Using this scenario, we first illustrate challenges, then formally define our model, and finally develop a safe type system.

3.4.1. Scenario

The scenario corresponds with a global type of the form $(G) \blacktriangleright (\{P_a\} : G_a, \{P_a, P_d\} : G_{ad}, ...)^{\kappa}$, with processes $P_{a..d}$ and a coordinator Ψ. We define a *task* to mean a series of interactions, which includes failure handling behaviors. Initially all processes are collaborating on a task ϕ, which we label $(\kappa, \emptyset)$ (identifying the task context, and the set of failed processes). The shaded boxes signify the tasks on which each process is working. Dotted arrows represent notifications between processes and Ψ related to task completion, and solid arrows represent failure notifications from Ψ to processes. During the scenario, P_a first fails, then P_d fails: the execution proceeds through failure handling for $\{P_a\}$ and

$\{P_a, P_d\}$.

(I) When P_b reaches the end of its part in ϕ, the application has P_b notify Ψ. P_b then remains in the context of ϕ (the continuation of the box after notifying) in consideration of other non-robust participants still working on ϕ – P_b may yet need to handle their potential failure(s).

(II) The processes of synchronizing on the completion of a task or performing failure handling *are themselves subject to failures* that may arise concurrently. In Figure 3.4, all processes reach the end of ϕ (i.e., four dotted arrows from ϕ), but P_a fails. Ψ determines this failure and initiates failure handling at time (1), while *done* notifications for ϕ continue to arrive asynchronously at time (2). The failure handling for crash of P_a is itself interrupted by the second failure at time (3).

(III) Ψ can receive notifications that are no longer relevant. For example, at time (2), Ψ has received all *done* notifications for ϕ, but the failure of P_a has already triggered failure handling from time (1).

(IV) Due to multiple concurrent failures, interacting participants may end up in different tasks: around time (2), P_b and P_d are in task $\phi' = (\kappa, \{P_a\})$, whereas P_c is still in ϕ (and asynchronously sending or receiving messages with the others). Moreover, P_c never executes ϕ' because of delayed notifications, so it goes from ϕ directly to $(\kappa, \{P_a, P_d\})$.

3.4.2. Syntax

Figure 3.5 defines the grammar of processes and networks (distributed applications). Expressions $e, e_i, ..$ can be values $v, v_i, ...$, variables $x, x_i, ...$, and standard operations.

(Application) processes and statements. (Application) processes are denoted by $P, P_i, ...$. An initialization $a[p](y).P$ agrees to play role p via shared name a and takes actions defined in P; actions are executed on a session channel $c : \eta$, where c ranges over $s[p]$ (session name and role name) and session variables y; η represents action statements.

A try-handle $\left(\eta \blacktriangleright H\right)^{\phi}$ attempts to execute the local action η, and can handle failures occurring therein as defined in the handling environment H, analogously to global types. H thus also maps a handler signature F to a handler body η defining how to handle F. Annotation $\phi = (\kappa, F)$ is composed of two elements: an identity κ of a *global* try-handle, and an indication of the *current* handler signature which can be empty. $F = \emptyset$ means that the default try-block is executing, whereas $F \neq \emptyset$ means that the handler body for F is

(Expression)	e	$::=$	$v \mid x \mid e+e \mid -e \mid \ldots$	(Channel)	c	$::=$	$s[p] \mid y$
(Process)	P	$::=$	$a[p](y).P \mid c:\eta$	(Level)	ϕ	$::=$	(κ, F)
(Statement)	η	$::=$	$(\eta \blacktriangleright H)^{\phi}.\eta \mid \underline{0} \mid 0 \mid p!l(e).\eta$	(Declaration)	D	$::=$	$X(x)=\eta$
		$\mid$	$p?\{l_i(x_i).\eta_i\}_{i\in I} \mid X\langle e\rangle$	(Handling)	H	$::=$	$F:\eta,\ H \mid \emptyset$
		$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ \eta \mid \mathsf{if}\ e\ \eta\ \mathsf{else}\ \eta$	(Queue)	h	$::=$	$\emptyset \mid h\cdot m$
(Network)	N	$::=$	$P \mid \Psi \blacklozenge N \mid s:h$	(Context)	E	$::=$	$(E \blacktriangleright H)^{\phi}.\eta$
		$\mid$	$N \mid N \mid (\nu s)N \mid 0$			$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ E \mid [\,]$
(Message)	m	$::=$	$\langle p,q,l(v)\rangle \mid \llparenthesis p,F \rrparenthesis \mid \langle p,q\rangle^{\phi}$	(Coordinator)	Ψ	$::=$	$G :_s (F,h)$

Figure 3.5.: Grammar for processes, applications, systems, and evaluation contexts.

executing. Term $\underline{0}$ only occurs in a try-handle during runtime. It denotes a *yielding* for a *notification* from a *coordinator* (introduced shortly).

Other statements are similar to those defined in Coppo et al. [CDYP16]. Term 0 represents an *idle* action. Following convention, we sometimes omit 0 at the end of a statement. Action $p!\ l(e).\eta$ represents a sending action that sends p a label l with content e, then it continues as η. Branching $p?\{l_i(x_i).\eta_i\}_{i\in I}$ represents a receiving action from p with several possible branches. When label l_k is selected, the transmitted value v is saved in x_k, and $\eta_k\{v/x_k\}$ continues. For convenience, when there is only one branch, the curly brackets are omitted, e.g., $c : p?l(x).P$ means there is only one branch $l(x)$. $X\langle e\rangle$ is for a statement variable with one parameter e, and $\mathsf{def}\ D\ \mathsf{in}\ \eta$ is for recursion, where declaration D defines the recursive body that can be called in η. The conditional statement is standard.

The structure of processes ensures that failure handling is not interleaved between different sessions. However, we note that in standard MPSTs [CDYP16, HYC16], session interleaving must in any case be prohibited for the basic progress property. Our model does allow parallel sessions at the top-level, whose actions may be concurrently interleaved during execution.

Network A *network* in our framework is a composition of processes (describing a distributed application) and a coordinator (cf. Figure 3.1). A network consists of processes P; a *system* $\Psi \blacklozenge N$, which consists of (robust) coordinator services $\Psi = G :_s (F, h_d)$, which coordinates the session s following the global type G and a network which is coordinated by the coordinator. The coordinator stores in (F, h_d) the failures F that occurred in the application, and in h_d done notifications sent to the coordinator. We may omit s from a coordinator if it is clear from the context. The coordinator is denoted by ψ when viewed as a role. The job of the coordinator is to ensure that even in the presence of failures there is consensus on whether all participants in a given try-handle completed

$$
\begin{array}{ll}
P_{drv} = & a[drv](y).y : \\
& \left(\begin{array}{ll}
\text{def } X(x_w) = & w_1 : \text{def } X_{w_1}(x_w) = \\
\quad w_1!\ l_{points}(x_w, getPoints(w_1)). & \quad w_2!\ l_{pointsw_1}(x_w, getPoints(w_1, w_2)). \\
\quad w_2!\ l_{points}(x_w, getPoints(w_2)). & \quad w_2?\ l_{gradw_1}(x_{res1}). \\
\quad w_1?\ l_{grad}(x_{res1}). & \quad X_{w_1}\langle calcW(x_w, x_{res1})\rangle \text{ in} \\
\quad w_2?\ l_{grad}(x_{res2}). & X_{w_1}\langle x_{wi}\rangle \\
\quad X\langle calcW(x_w, x_{res1}, x_{res2})\rangle \text{ in} & w_2 : ... \\
X\langle x_{wi}\rangle & w_1, w_2 : 0
\end{array}\right)^{(1,\emptyset)}
\end{array}
$$

Figure 3.6.: A driver process that implements the driver role, drv, from the LR model (Figure 3.2).

their respective local actions, or whether failures need to be handled, and which ones; global queues $s : h$ carry a sequence of messages m sent by participants in session s. A message is either a regular message $\langle p, q, l(v)\rangle$ with label l and content v sent from p to q or a *notification*. A notification may contain the role of a coordinator. There are *done* and *failure* notifications. The done notifications: $\langle p, \psi\rangle^{\phi}$ notifies ψ that p has finished its local actions of the try-handle ϕ; $\langle \psi, p\rangle^{\phi}$ is sent from ψ to notify p that ψ has received all done notifications for the try-handle ϕ so that p shall end its current try-handle and move to its next task. For example, in Figure 3.4 at time (4) the coordinator will inform P_b and P_c via $\langle \psi, P_b\rangle^{(\kappa,\{P_a,P_d\})}.\langle \psi, P_c\rangle^{(\kappa,\{P_a,P_d\})}$ that they can finish the try-handle $(\kappa, \{P_a, P_d\})$. Note that the appearance of $\langle \psi, p\rangle^{\phi}$ implies that the coordinator has been informed that all participants in ϕ have completed their local actions. The *failure* notifications: $\langle\!\langle \psi, F\rangle\!\rangle$ notifies ψ that F occurred (e.g., $\{q\}$ means q has failed); and $\langle\!\langle p, F\rangle\!\rangle$ is sent from ψ to notify p about the failure F for possible handling. We write $\langle\!\langle \widetilde{p}, F\rangle\!\rangle$, where $\widetilde{p} = p_1, ..., p_n$ short for $\langle\!\langle p_1, F\rangle\!\rangle \cdot ... \cdot \langle\!\langle p_n, F\rangle\!\rangle$; similarly for $\langle \psi, \widetilde{p}\rangle^{\phi}$; $N \mid N'$ composes two networks in parallel; $(\nu s)N$ hides the session s in N; and 0 is an inactive network.

Processes of LR example

Figures 3.6 and 3.7 present an implementation of the LR model (see Figure 3.2 for the global type). Figure 3.6 presents the implementation of the driver role, drv, and Figure 3.7 presents the implementations of the worker roles, w_1 and w_2. We take the liberty of using non-defined functions for local calculations that involve *no* communication. The first process P_{drv} (see Figure 3.6) implements the driver, starting with an initialization $a[drv](y).y$ which states that the process will play drv via the shared name a. The following statement is a try-handle annotated by $(1, \emptyset)$ where 1 matches the try-handle annotation from the global type (see Figure 3.2) and $\emptyset$ states that the try-handle has no active failure

$$
\begin{array}{ll}
P_{w_1} = & a[w_1](y).y : \\
& \left(\begin{array}{l} \textsf{def } X(x) = \\ \quad drv?\ l_{points}(x_w, x_{points}). \\ \quad drv!\ l_{grad}(calcG(x_w, x_{points})). \\ \quad X\langle 0\rangle \\ \textsf{ in } X\langle 0\rangle \end{array} \blacktriangleright \begin{array}{l} w_2 : \textsf{def } X_{w_2}(x) = \\ \quad drv?\ l_{pointsw_2}(x_w, x_{points}). \\ \quad drv!\ l_{gradw_2}(calcG(x_w, x_{points})). \\ \quad X_{w_2}\langle 0\rangle \\ \textsf{ in } X_{w_2}\langle 0\rangle \\ \ldots \end{array} \right)^{(1,\emptyset)} \\
P_{w_2} = & a[w_2](y).y : \\
& \left(\begin{array}{l} \textsf{def } X(x) = \\ \quad drv?\ l_{points}(x_w, x_{points}). \\ \quad drv!\ l_{grad}(calcG(x_w, x_{points})). \\ \quad X\langle 0\rangle \\ \textsf{ in } X\langle 0\rangle \end{array} \blacktriangleright \begin{array}{l} w_1 : \textsf{def } X_{w_1}(x) = \\ \quad drv?\ l_{pointsw_1}(x_w, x_{points}). \\ \quad drv!\ l_{gradw_1}(calcG(x_w, x_{points})). \\ \quad X_{w_1}\langle 0\rangle \\ \textsf{ in } X_{w_1}\langle 0\rangle \\ \ldots \end{array} \right)^{(1,\emptyset)}
\end{array}
$$

Figure 3.7.: Worker processes that implement the worker roles, w_1 and w_2, from the LR model (Figure 3.2).

handling. In the default block, the driver contains a recursive definition (def $X(x_w) =$ ... in ...) that defines the communication for one iteration of the LR model. The initial call to the recursive definition $X\langle x_{wi}\rangle$ provides the initial model parameter x_{wi}. Inside the recursive definition, the driver sends the point information and model parameter to all workers, e.g., $w_1!\ l_{points}(x_w, getPoints(w_1))$ for worker w_1. The driver then waits until it receives the result of the model update calculations from all workers, e.g., $w_1?\ l_{grad}(x_{res1})$ for worker w_1. It calculates the LR model parameter update and repeats the previous steps ($X\langle calcW(x_w, x_{res1}, x_{res2})\rangle$). Furthermore, the implementation provides failure handling for the following failures $\{w_1\}$, $\{w_2\}$, and $\{w_1, w_2\}$. The handler body for $\{w_1\}$ and $\{w_2\}$ follows the default block with the difference that a handler body only involves one worker, namely the non-failed worker. The handler body for $\{w_1, w_2\}$ is just 0, i.e., ending the LR model since all workers have failed.

The implementations of both workers (P_{w_1} and P_{w_2} in Figure 3.7) have a structure which is similar to the implementation of the driver. After the initialization statement, they have a try-handle annotated with $(1, \emptyset)$ matching the try-handle level from the global type. In the default activity, they contain a recursive definition in which they expect to receive the point information and model parameter from the driver, then calculate the gradient and send the result to the driver. In the failure handling, if the other worker has failed, the general behavior is the same as in the default block; the only difference is that the worker has to process more points in the gradient calculation.

$$\frac{a : G \qquad \Psi = G :_s (\emptyset, \emptyset) \qquad \{p_1, .., p_n\} = pid(G)}{a[p_1](y_1).P_1 \mid ... \mid a[p_n](y_n).P_n \to (\nu s)(\Psi \blacklozenge P_1\{s[p_1]/y_1\} \mid ... \mid P_n\{s[p_n]/y_n\} \ \mid \ s : \emptyset)} \quad \textbf{(Link)}$$

$$s[p] : E[q!\, l(e).\eta] \ \mid \ s : h \to s[p] : E[\eta] \ \mid \ s : h \cdot \langle p, q, l(v)\rangle \qquad e \Downarrow v \quad \textbf{(Snd)}$$

$$\frac{l_k \in \{l_i\}_{i \in I}}{s[p] : E[q?\{l_i(x_i).\eta_i\}_{i \in I}] \ \mid \ s : \langle q, p, l_k(v_k)\rangle \cdot h \to s[p] : E[\eta_k\{v_k/x_k\}] \ \mid \ s : h} \quad \textbf{(Rcv)}$$

$$s[p] : E[\mathsf{def}\ X(x) = \eta\ \mathsf{in}\ X\langle e\rangle] \to s[p] : E[\mathsf{def}\ X(x) = \eta\ \mathsf{in}\ \eta\{v/x\}] \quad e \Downarrow v \quad \textbf{(Rec)}$$

$$\frac{N_1 \equiv N_3 \to N_4 \equiv N_2}{N_1 \to N_2} \qquad \frac{N_1 \to N_2}{N_1 \mid N \to N_2 \mid N} \qquad \frac{N_1 \to N_2}{\Psi \blacklozenge N_1 \to \Psi \blacklozenge N_2} \qquad \frac{N \to N'}{(\nu s)N \to (\nu s)N'} \quad \begin{array}{l}\textbf{(Str, Par)}\\ \textbf{(Sys, New)}\end{array}$$

$$N \ \mid \ s : h \to N \setminus s[p] : \eta \ \mid \ s : \mathit{remove}(h, p) \cdot \langle\!\langle \psi, \{p\}\rangle\!\rangle \quad s[p] : \eta \text{ non-robust} \quad \textbf{(Crash)}$$

Figure 3.8.: Operational semantics of distributed applications.

3.4.3. Basic dynamic semantics for processes

Figure 3.8 shows the operational semantics of applications. The processes and networks are considered modulo structural equivalence, denoted by $\equiv$, and defined by the rules in Definition 5 along with α-renaming. We use evaluation contexts as defined in Figure 3.5. Context E is either a hole $[\,]$, a default context $\big(E \blacktriangleright H\big)^{\phi}.\eta$, or a recursion context $\mathsf{def}\ D\ \mathsf{in}\ E$. We write $E[\eta]$ to denote the action statement obtained by filling the hole in $E[\cdot]$ with η.

Rule **(Link)** says that (local) processes who agree on shared name a, obeying some protocol (global type), playing certain roles p_i represented by $a[p_i](y_i).P$, will together start a private session s; this will result in replacing every variable y_i in P_i and, at the same time, creating a new global queue $s : \emptyset$, and appointing a coordinator $G :_s (\emptyset, \emptyset)$, which is novel in our work.

Rule **(Snd)** in Figure 3.8 reduces a sending action $q!\, l(e)$ by emitting a message $\langle p, q, l(v)\rangle$ to the global queue $s : h$. Rule **(Rcv)** reduces a receiving action if the message arriving at its end is sent from the expected sender with an expected label. Rule **(Rec)** is for recursion. When the recursive body, defined inside η, is called by $X\langle e\rangle$ where e is evaluated to v, it reduces to the statement $\eta\{v/x\}$ which will again implement the recursive body. Rule **(Str)** says that a process can do a step, if a process that is structurally concurrent to that process can do a step. Structural concurrence is unsurprising and we define it after discussing the remaining rules. Rule **(Par)** states that a parallel composition has a reduction if its sub-application can reduce. Rule **(Sys)** states that a system has a reduction if the network has a reduction, and **(New)** says a reduction can proceed under a session.

Rule **(Crash)** states that a process on channel $s[p]$ can fail at any point in time. **(Crash)** also

adds a notification $\langle\!\langle \psi, F \rangle\!\rangle$ which is sent to ψ (the coordinator). This is an abstraction for the failure detector described in Section 3.2 system model assumption (4). The notification $\langle\!\langle \psi, F \rangle\!\rangle$ is the first such notification issued by a participant based on its local failure detector. Adding the notification into the global queue instead of making the coordinator immediately aware of it models that failures are only detected eventually. Note that a failure is not annotated with a level because failures transcend all levels, and asynchrony makes it impossible to identify "where" exactly they occurred. As a failure is permanent it can affect multiple try-handles. The **(Crash)** rule does not apply to participants which are robust, i.e., that conceptually cannot fail (e.g., drv in Figure 3.2). Rule **(Crash)** removes channel $s[p]$ (the failed process) from application N, and removes messages and notifications delivered from, or heading to, the failed p by function *remove*(h, p). Function *remove*(h, p) returns a new queue after removing all regular messages and notifications that contain p, e.g., let $h = \langle p_2, p_1, l(v)\rangle \cdot \langle p_3, p_2, l'(v')\rangle \cdot \langle p_3, p_4, l'(v')\rangle \cdot \langle p_2, \psi\rangle^{\phi} \cdot \langle\!\langle p_2, \{p_3\}\rangle\!\rangle \cdot \langle \psi, p_2\rangle^{\phi}$ then *remove*$(h, p_2) = \langle p_3, p_4, l'(v')\rangle$. Messages are removed to model that in a real system send and receive does *not* constitute an atomic action.

We now detail structural congruence used in rule **(Str)**

Structural congruence Structural congruence is defined as follows:

Definition 5 (Structural Congruence)**.**

$$h \equiv \emptyset \cdot h \equiv h \cdot \emptyset \qquad \frac{m \cdot m' \curvearrowright m' \cdot m}{h \cdot m \cdot m' \cdot h' \equiv h \cdot m' \cdot m \cdot h'} \qquad \frac{h \equiv h'}{s : h \equiv s : h'}$$

$$\textsf{def } D \textsf{ in } 0 \equiv 0 \qquad \begin{array}{c} \textsf{def } D \textsf{ in } (\textsf{def } D' \textsf{ in } \eta) \equiv \textsf{def } D' \textsf{ in } (\textsf{def } D \textsf{ in } \eta) \\ \textsf{if } dpv(D) \cap (dpv(D') \cup fpv(\eta)) = dpv(D') \cap (dpv(D) \cup fpv(\eta)) = \emptyset \end{array}$$

$$\Psi \blacklozenge 0 \equiv 0 \qquad \frac{N \equiv N'}{\Psi \blacklozenge N \equiv \Psi \blacklozenge N'} \qquad (N_1 \mid N_2) \mid N_3 \equiv N_1 \mid (N_2 \mid N_3) \qquad (N_1 \mid N_2 \equiv N_2 \mid N_1)$$

$$N \mid 0 \equiv N \qquad (\nu s)(\nu s')N \equiv (\nu s')(\nu s)N \qquad (\nu s)N \mid N' \equiv (\nu s)(N \mid N') \quad \textsf{if } s \notin fn(N')$$

In Definition 5, the rules in the first lines allow the permutation of messages, modeling asynchrony. $m \cdot m' \curvearrowright m' \cdot m$ means that the order of $m \cdot m'$ can be switched to $m' \cdot m$. Permutation is detailed below. The rules in the next two lines give structural congruence over recursions. Function $dpv(D)$ gives the set of process variables in declarations, and $fpv(\eta)$ gives the set of process variables which occur freely in η. The final two lines of rules state structural congruence of networks where: the first two rules defines structural congruence of a network N combined with a coordinator Ψ; the next three rules define structural congruence for parallel composition of network; and the last rules define structural concurrent for restriction and scope extension in the standard way. Function $fn(N)$ on the last line gives the set of free names in N.

We now detail the permutation of messages. Permutation is possible, when messages and notifications have different sources or destinations, or done notifications have different levels (e.g., $\phi \neq \phi'$).

Definition 6 (Permutable Messages)**.** *We define $m_i \cdot m_j \curvearrowright m_j \cdot m_i$, $i \neq j$, saying $m_i \cdot m_j$ can be permuted to $m_j \cdot m_i$, if* none *of the following conditions holds:*

- $m_i = \langle p, q, l(v) \rangle$ *and* $m_j = \langle p, q, l'(v') \rangle$ *for some* l, l', v, v'.
- $m_i = \langle \psi, q \rangle^{\phi}$ *and* $m_j = \llbracket q, F \rrbracket$ *for some* ϕ, F.

For example the following messages are permutable: $\langle p, q, l(v) \rangle \cdot \langle p, q', l(v) \rangle$ if $q \neq q'$ and $\langle p, q, l(v) \rangle \cdot \langle \psi, p \rangle^{\phi}$ and $\langle p, q, l(v) \rangle \cdot \llbracket q, F \rrbracket$. But $\langle \psi, p \rangle^{\phi} \cdot \llbracket p, F \rrbracket$ is not permutable, because both have the same sender and receiver (ψ is the sender of $\llbracket p, F \rrbracket$).

3.4.4. Handling at processes

Failure handling, defined in Figure 3.9, is based on the observations that (i) a process that fails stays down, and (ii) multiple processes can fail. As a consequence, a failure can trigger multiple failure handlers either because these handlers are in different (subsequent) try-handles or because of additional failures. Therefore a process needs to retain the information of *who* failed. For simplicity we do not model state at processes, but instead processes read but do not remove failure notifications from the global queue. We define $Fset(h, p)$ to return the union of failures for which there are notifications heading to p, i.e., $\llbracket p, F \rrbracket$, issued by the coordinator in queue h *up to the first done notification heading to* p:

Definition 7 (Union of Existing Failures $Fset(h, p)$)**.**

$$Fset(\emptyset, p) = \emptyset \quad Fset(h, p) = \begin{cases} F \cup Fset(h', p) & \text{if } h = \llbracket p, F \rrbracket \cdot h' \\ \emptyset & \text{if } h = \langle \psi, p \rangle^{\phi} \cdot h' \\ Fset(h', p) & \text{otherwise } h = m \cdot h' \end{cases}$$

In short, if the global queue is $\emptyset$, then naturally there are no failure notifications. If the global queue starts with a failure notification sent from the coordinator, say $\llbracket p, F \rrbracket$, we collect the failure and process the tail. If the global queue starts with a done notification $\langle \psi, p \rangle^{\phi}$ sent from the coordinator, then $Fset(h, p)$ stops collecting failure notifications. Otherwise, $Fset(h, p)$ ignores the first message and processes the tail.

Our failure handling semantics, **(TryHdl)**, allows a try-handle $\phi = (\kappa, F)$ to handle different failures or sets of failures by allowing a try-handle to switch between different

$$\frac{F' = \cup\{A \mid A \in \mathit{dom}(H) \wedge F \subset A \subseteq \mathit{Fset}(h,p)\} \quad F' : \eta' \in H}{s[p] : E[\big(\eta \blacktriangleright H\big)^{(\kappa,F)}.\eta''] \;\mid\; s : h \rightarrow s[p] : E[\big(\eta' \blacktriangleright H\big)^{(\kappa,F')}.\eta''] \;\mid\; s : h} \quad \textbf{(TryHdl)}$$

$$s[p] : E[\big(0 \blacktriangleright H\big)^{\phi}.\eta] \;\mid\; s : h \rightarrow s[p] : E[\big(\underline{0} \blacktriangleright H\big)^{\phi}.\eta] \;\mid\; s : h \cdot \langle p, \psi\rangle^{\phi} \quad \textbf{(SndDone)}$$

$$\frac{\langle \psi, p\rangle^{\phi} \in h}{s[p] : E[\big(\underline{0} \blacktriangleright H\big)^{\phi}.\eta] \;\mid\; s : h \rightarrow s[p] : E[\eta] \;\mid\; s : h \setminus \{\langle \psi, p\rangle^{\phi}\}} \quad \textbf{(RcvDone)}$$

$$s[p] : E[\eta] \;\mid\; s : \langle q, p, l(v)\rangle \cdot h \rightarrow s[p] : E[\eta] \;\mid\; s : h \quad l \notin \mathit{labels}(E[\eta]) \quad \textbf{(Cln)}$$

$$\frac{\langle \psi, p\rangle^{\phi} \in h \quad \phi \notin E[\eta]}{s[p] : E[\eta] \;\mid\; s : h \rightarrow s[p] : E[\eta] \;\mid\; s : h \setminus \langle \psi, p\rangle^{\phi}} \quad \textbf{(ClnDone)}$$

Figure 3.9.: Operational semantics of distributed applications, for endpoint handling.

handlers. F thus denotes the current set of handled failures. For simplicity we refer to this as the *current(ly handled) failure set*. This is a slight abuse of terminology for the purpose of brevity, as failures are obviously only detected with a certain time lag. The handling strategy for a process is to handle the — currently — largest set of failed processes that this process has been informed of and is able to handle. This largest set is calculated by $\cup\{A \mid A \in \mathit{dom}(H) \wedge F \subset A \subseteq \mathit{Fset}(h,p)\}$, that selects all failure sets which are larger than the current one ($A \in \mathit{dom}(H) \wedge F \subset A$) if they are also triggered by known failures ($A \subseteq \mathit{Fset}(h,p)$). Condition $F' : \eta' \in H$ in **(TryHdl)** ensures that there is a handler for F'. The following example shows how **(TryHdl)** is applied to switch handlers.

Example 7. *Take h such that $\mathit{Fset}(h,p) = \{p_1\}$ and $H = \{p_1\} : \eta_1, \{p_2\} : \eta_2, \{p_1, p_2\} : \eta_{12}$ in process $P = s[p] : \big(\eta_1 \blacktriangleright H\big)^{(\kappa,\{p_1\})}$, which indicates that P is handling failure $\{p_1\}$. Assume now one more failure occurs and results in a new queue h' such that $\mathit{Fset}(h',p) = \{p_1, p_2\}$. By* **(TryHdl)***, the process acting at $s[p]$ is handling the failure set $\{p_1, p_2\}$ such that $P = s[p] : \big(\eta_{12} \blacktriangleright H\big)^{(\kappa,\{p_1,p_2\})}$ (also notice the η_{12} inside the try-block). A switch to only handling $\{p_2\}$ does not make sense, since, e.g., η_2 can contain p_1. Figure 3.2 shows a case in which the handling strategy differs according to the number of failures.*

In Section 3.3 we formally define well-formedness conditions, which guarantee that if there are two handlers for two different handler signatures in a try-handle, then a handler exists for their union. The following example demonstrates why such a guarantee is needed.

Example 8. *Assume a slightly different P compared to the previous examples (no handler for the union of failures): $P = s[p] : E[\big(\eta \blacktriangleright H\big)^{(\kappa,\emptyset)}]$ with $H = \{p_1\} : \eta_1, \{p_2\} : \eta_2$. Assume also that $\mathit{Fset}(h,p) = \{p_1, p_2\}$. Here* **(TryHdl)** *will not apply since there is no failure handling for*

$\{p_1, p_2\}$ in P. If we were to allow a handler for either $\{p_1\}$ or $\{p_2\}$ to be triggered we would have no guarantee that other participants involved in this try-handle will all select the same failure set. Even with a deterministic selection, i.e., all participants in that try-handle selecting the same handling activity, there needs to be a handler with handler signature $= \{p_1, p_2\}$ since it is possible that p_1 is involved in η_2. Therefore, the type system will ensure that there is a handler for $\{p_1, p_2\}$ either at this level or at an outer level.

Before discussing the next rule, we revisit observation (I), which explains that a process finishing its default action (P_b) cannot leave its current try-handle $(\kappa, \emptyset)$ immediately because other participants may fail (P_a failed). Equation 3.1 below also shows this issue from the perspective of semantics:

$$s[p] : \big(0 \blacktriangleright F : q!l(10).q?l'(x)\big)^{(\kappa,\emptyset)}.\eta' \;\mid\; s[q] : \big(p?l(x').p!l'(x'+10) \blacktriangleright H^{\uparrow}\big)^{(\kappa,F)}.\eta'' \mid\; s : \langle\!\langle q, F\rangle\!\rangle \cdot \langle\!\langle p, F\rangle\!\rangle \cdot h \quad (3.1)$$

In Equation 3.1 the process acting on $s[p]$ ended its try-handle (i.e., the action is 0 in the try-block), and if $s[p]$ finishes its try-handle the participant acting on $s[q]$ which started handling F would be stuck. To solve the issue, we use **(SndDone)** and **(RcvDone)** for completing a local try-handle with the help of a coordinator.

The rule **(SndDone)** sends out a done notification $\langle p, \psi\rangle^{\phi}$ if the current action in ϕ is 0 and sets the action to $\underline{0}$, indicating that a done notification from the coordinator is needed for ending the try-handle. The send out done notification of a process indicates that the process wants to finish the try-handle, but has no further effect on the reduction until the coordinator processes it. Assume process on channel $s[p]$ finished its local actions in the try-block (i.e., as in Equation 3.1 above), then by **(SndDone)**, we have

$$(3.1) \rightarrow\; s : \langle\!\langle q, F\rangle\!\rangle \cdot \langle\!\langle p, F\rangle\!\rangle \cdot \langle p, \psi\rangle^{(\kappa,\emptyset)} \cdot h \;\mid\; s[p] : \big(\underline{0} \blacktriangleright F : q!l(10).q?l'(x)\big)^{(\kappa,\emptyset)}.\eta' \;\mid\; s[q] : \big(p?l(x').p!l'(x'+10) \blacktriangleright H^{\uparrow}\big)^{(\kappa,F)}.\eta''$$

where notification $\langle p, \psi\rangle^{(\kappa,\emptyset)}$ is added to inform the coordinator. Now the process on channel $s[p]$ can still handle failures defined in its handling environment. This is similar to the case described in observation (II).

Rule **(RcvDone)** is the counterpart of **(SndDone)**. Once a process receives a done notification for ϕ from the coordinator it can finish the try-handle ϕ and reduces to the continuation η. Consider Equation 3.2 below, which is similar to Equation 3.1, except that take a case where the try-handle can be reduced with **(RcvDone)**. In Equation 3.2 **(SndDone)** is applied:

$$s[p] : \big(\underline{0} \blacktriangleright F : q!l(10).q?l'(x)\big)^{(\kappa,\emptyset)}.\eta' \;\mid\; s[q] : \big(\underline{0} \blacktriangleright F : p?l(x').p!l'(x'+10)\big)^{(\kappa,\emptyset)}.\eta'' \;\mid\; s : h \quad (3.2)$$

With $h = \langle\psi, q\rangle^{(\kappa,\emptyset)} \cdot \langle\psi, p\rangle^{(\kappa,\emptyset)} \cdot \llbracket q, F\rrbracket \cdot \llbracket p, F\rrbracket$ both processes can apply **(RcvDone)** and safely terminate the try-handle $(\kappa, \emptyset)$. Note that $Fset(h, p) = Fset(h, q) = \emptyset$ (by Definition 7), i.e., rule **(TryHdl)** cannot be applied since a done notification suppresses the failure notification. Further, the done and the failure notifications cannot be permuted. Thus Equation 3.2 will reduce to:

$$(3.2) \rightarrow^* s[p] : \eta' \ \mid \ s[q] : \eta'' \ \mid \ s : \llbracket q, F\rrbracket \cdot \llbracket p, F\rrbracket$$

It is possible that η' or η'' have handlers for F. Note that once a queue contains $\langle\psi, p\rangle^{(\kappa,\emptyset)}$, all non-failed processes in the try-handle $(\kappa, \emptyset)$ have sent done notifications to ψ (i.e., applied rule **(SndDone)**). The coordinator which will be introduced shortly ensures this.

Before explaining rule **(Cln)** we define the function $labels(\eta)$ (see Definition 8), which returns all labels of receiving actions in η which are able to receive messages now or possibly later. The only receiving actions which cannot and never will be able to receive messages in η are those in a handler with signature F in a try-handle (κ, F') where $F \subseteq F'$ (**(TryHdl)** cannot switch the handling to F).

Definition 8 (Extracting Reachable Labels in η)**.** *We define $labels(\eta)$, which is the over-approximation of the set of all labels reachable in η, as the fix point of:*

$$\begin{array}{rcl}
labels(p!l(e).\eta) & = & labels(\eta) \\
labels(p?\{l_i(e_i).\eta_i\}_{i\in I}) & = & \bigcup_{i\in I}(\{l_i\} \cup labels(\eta_i)) \\
labels(\big(\eta \blacktriangleright H\big)^{(\kappa,F)}.\eta') & = & labels(\eta) \cup labels(\eta') \bigcup_{(F':\eta'')\in H \wedge F\subset F'} labels(\eta'') \\
labels(X\langle e\rangle) & = & labels(\eta) \quad \textit{where def } X(x) = \eta \textit{ in } \eta' \textit{ (the called)} \\
labels(\textsf{def } X(x) = \eta \textsf{ in } \eta') & = & labels(\eta') \\
labels(\textsf{if } e\ \eta \textsf{ else } \eta') & = & labels(\eta) \cup labels(\eta') \\
\multicolumn{3}{c}{labels(0) = \emptyset \qquad labels(\underline{0}) = \emptyset}
\end{array}$$

In detail, $labels()$ collects labels in η based on the following cases: for a send $p!l(e).\eta$ it collects all labels in the continuations; for a branching $p?\{l_i(e_i).\eta_i\}_{i\in I}$ it collects all branching labels and all labels in all continuations; for a try-handle $\big(\eta \blacktriangleright H\big)^{(\kappa,F)}.\eta'$ it collects all labels in the statements η and η', and all labels in the handler bodies with handling signatures larger ($\subset$) than the current handler signature; for an end 0 or an end waiting for a done $\underline{0}$ it collects no labels; for a statement variable $X\langle e\rangle$ it collects the labels of the called definition; for a definition def $X(x) = \eta$ in η' it collects the labels of η'; for a condition statement if $e\ \eta$ else η' it collects the label in both cases.

Rule **(Cln)** removes a normal message from the queue if the label in the message does not exist in the target process, which can happen when a failure handler was triggered. This

$$\frac{\tilde{p} = \mathit{roles}(G) \setminus F' \quad F' = F \cup \{p\} \quad m = \langle\!\langle \tilde{p}, \{p\} \rangle\!\rangle}{G : (F, h_d) \blacklozenge N \ \mid \ s : \langle\!\langle \psi, \{p\} \rangle\!\rangle \cdot h \rightarrow G : (F', h_d) \blacklozenge N \ \mid \ s : h \cdot m} \quad \textbf{(F)}$$

$$\frac{h'_d = h_d \cdot \langle p, \psi \rangle^{\phi}}{G : (F, h_d) \blacklozenge N \ \mid \ s : \langle p, \psi \rangle^{\phi} \cdot h \rightarrow G : (F, h'_d) \blacklozenge N \ \mid \ s : h} \quad \textbf{(CollectDone)}$$

$$\frac{\mathit{roles}(h_d, \phi) \supseteq \mathit{roles}(G, \phi) \setminus F \quad \forall F' \in \mathit{hdl}(G, \phi).(F' \not\subseteq F) \quad \widetilde{ps} = \mathit{roles}(G, \phi) \setminus F}{G : (F, h_d) \blacklozenge N \ \mid \ s : h \rightarrow G : (F, \mathit{remove}(h_d, \phi)) \blacklozenge N \ \mid \ s : h \cdot \langle \psi, \widetilde{ps} \rangle^{\phi}} \quad \textbf{(IssueDone)}$$

Figure 3.10.: Operational semantics for the coordinator.

removal based on the syntactic process is safe because in a global type separate branch types *not* defined in the same default block or handler body must have disjoint sets of labels (c.f., Section 3.3). Let $\phi \in P$ if try-handle ϕ appears inside P. Rule **(ClnDone)** removes a done notification of ϕ from the queue if no try-handle ϕ exists, which can happen in the case of nesting when a handler of an outer try-handle is triggered.

3.4.5. Handling at coordinator

Figure 3.10 defines the semantics of the coordinator. The rule **(F)**, states that the coordinator collects and removes a failure notification $\langle\!\langle \psi, p \rangle\!\rangle$ heading to it, retains this notification by $G : (F', h_d)$, $F' = F \cup \{p\}$, where F represents the failures that the coordinator is already aware of, and issues failure notifications to all non-failed participants. $\mathit{roles}(G)$ extracts the set of *all* roles appearing in G.

Rules **(CollectDone, IssueDone)**, in short, instruct all participants in $\phi = (\kappa, F)$ to finish their try-handle ϕ if the coordinator has received sufficient done notifications of ϕ and did not send out failure notifications that interrupt the task (κ, F) (e.g., see observation (III)). Rule **(CollectDone)** collects done notifications, i.e., $\langle p, \psi \rangle^{\phi}$, from the queue and retains these notifications which are used in **(IssueDone)**. Before introducing **(IssueDone)**, we first introduce $\mathit{hdl}(G, (\kappa, F))$ to return a set of handler signatures which can be triggered with respect to the current handler:

Definition 9. $\mathit{hdl}(G, (\kappa, F)) = \mathit{dom}(H^{\uparrow}) \setminus \mathcal{P}(F)$ *if* $\left(G_0 \blacktriangleright H^{\uparrow}\right)^{\kappa} \in G$ *where* $\mathcal{P}(F)$ *represents a powerset of* F.

Also, we abuse the function *roles* to collect the non-coordinator roles of ϕ in h_d, written $\mathit{roles}(h_d, \phi)$. Note h_d contains only done notifications sent by participants. Similarly, we write $\mathit{roles}(G, \phi)$ where $\phi = (\kappa, F)$ to collect the roles appearing in the try-handle κ without the failed roles, i.e. $\mathit{roles}(\kappa) \setminus F$.

Rule **(IssueDone)** is applied for some ϕ when conditions $\forall F' \in hdl(G, \phi).(F' \not\subseteq F)$ and $roles(h_d, \phi) \supseteq roles(G, \phi) \setminus F$ are both satisfied, where F contains all failures, of which the coordinator is aware. Intuitively, these two conditions ensure that (1) the coordinator only issues done notifications to the participants in the try-handle ϕ if it did not send failure notifications which will trigger a handler of the try-handle ϕ; (2) the coordinator has received all done notifications from all non-failed participants of ϕ.

We further explain both conditions in the following examples, starting from condition $\forall F' \in hdl(G, \phi).(F' \not\subseteq F)$, which ensures that no handler in ϕ can be triggered based on the failure notifications F sent out by the coordinator.

Example 9. *Assume a process playing role p_i is $P_i = s[p_i] : \left(\eta_i \blacktriangleright H_i\right)^{\phi_i}$ where $i \in \{1, 2, 3\}$, $H_i = \{p_2\} : \eta_{i2},\ \{p_3\} : \eta_{i3},\ \{p_2, p_3\} : \eta_{i23}$; the coordinator is $G : (\{p_2, p_3\}, h_d)$ ($F = \{p_2, p_3\}$) where $\left(G' \blacktriangleright H^{\uparrow}\right)^{\kappa} \in G$, $dom(H^{\uparrow}) = dom(H_i)$ for any $i \in \{1, 2, 3\}$, and $h_d = \langle p_1, \psi\rangle^{(\kappa, \{p_2\})} \cdot \langle p_1, \psi\rangle^{(\kappa, \{p_2, p_3\})} \cdot h'_d$. For any ϕ in h_d, the coordinator checks whether it has issued a failure notification that could trigger a new handler of ϕ:*

1. *For $\phi = (\kappa, \{p_2\})$ the coordinator issued failure notifications that can interrupt a handler since*

$$hdl(G, (\kappa, \{p_2\})) = dom(H^{\uparrow}) \setminus \mathcal{P}(\{p_2\}) = \{\{p_3\}, \{p_2, p_3\}\}$$

   and $\{p_2, p_3\} \subseteq \{p_2, p_3\}$. That means the failure notifications issued by the coordinator, i.e., $\{p_2, p_3\}$, can trigger the handler with signature $\{p_2, p_3\}$. Thus the coordinator will not issue done notifications for $\phi = (\kappa, \{p_2\})$. A similar case is depicted in Figure 3.4 at time (2).

2. *For $\phi = (\kappa, \{p_2, p_3\})$ the coordinator did not issue failure notifications that can interrupt a handler, because*

$$hdl(G, (\kappa, \{p_2, p_3\})) = dom(H^{\uparrow}) \setminus \mathcal{P}(\{p_2, p_3\}) = \emptyset$$

   so that $\forall F' \in hdl(G, (\kappa, \{p_2, p_3\})).(F' \not\subseteq \{p_2, p_3\})$ is true. The coordinator will issue done notifications for $\phi = (\kappa, \{p_2, p_3\})$.

Another condition $roles(h_d, \phi) \supseteq roles(G, \phi) \setminus F$ states that only when the coordinator sees sufficient done notifications (in h_d) for ϕ, does it issue done notifications to *all* non-failed participants in ϕ, i.e., $\langle \psi, roles(G, \phi) \setminus F\rangle^{\phi}$. Recall that $roles(h_d, \phi)$ returns all roles which have sent a done notification for the handling of ϕ and $roles(G, \phi)$ returns all roles involved in the handling of ϕ. Intuitively one might expect the condition to be $roles(h_d, \phi) = roles(G, \phi)$; the following example shows why this would be wrong.

Example 10. *Consider a process P acting on channel $s[p]$ ($\{q\} \notin dom(H)$):*

$$P = s[p] : \big(...\big(... \blacktriangleright \{q\} : \eta, H'\big)^{\phi'} \blacktriangleright H\big)^{\phi}$$

Assume P has already reduced to:

$$P = s[p] : \big(\underline{0} \blacktriangleright H\big)^{\phi}$$

We show why roles$(h_d, \phi) \supseteq$ roles$(G, \phi) \setminus F$ is necessary. We start with the simple cases and then move to the more complicated ones.

(a) Assume q did not fail, the coordinator is $G : (\emptyset, h_d)$, and all roles in ϕ issued a done notification. Then roles$(h_d, \phi) =$ roles(G, ϕ) and $F = \emptyset$.

(b) Assume q failed in the try-handle ϕ', the coordinator is $G : (\{q\}, h_d)$, and all roles except q in ϕ issued a done notification. roles$(h_d, \phi) \neq$ roles(G, ϕ) however roles$(h_d, \phi) =$ roles$(G, \phi) \setminus \{q\}$. Cases like this are the reason why **(IssueDone)** *only requires done notifications from non-failed roles.*

(c) Assume q failed after it had issued a done notification for ϕ (i.e., q finished try-handle ϕ') and the coordinator collected it (by **(CollectDone)***), so we have $G : (\{q\}, h_d)$ and $q \in$ roles(d, ϕ). Then roles$(h_d, \phi) \supset$ roles$(G, \phi) \setminus \{q\}$. I.e.* **(IssueDone)** *needs to consider done notifications from failed roles.*

Thus rule **(IssueDone)** *has the condition roles$(h_d, \phi) \supseteq$ roles$(G, \phi) \setminus F$ because of cases like (b) and (c).*

The interplay between the issuing of done notifications **(IssueDone)** and the issuing of failure notifications **(F)** is non-trivial. The following proposition clarifies that the participants in the same try-handle ϕ will never get confused with handling failures or completing the try-handle ϕ.

Proposition 1. *Given $s : h$ with $h = h' \cdot \langle \psi, p \rangle^{\phi} \cdot h''$ and $Fset(h, p) = F_s$, the rule* **(TryHdl)** *is not applicable for the try-handle ϕ at the process playing role p.*

Proof. Without loss of generality we assume that the coordinator is $\Psi = G :_s (F_q, h_d)$ and $\phi = (\kappa, F'_s)$. The failure and done notifications sent by Ψ to p are not permutable by Definition 6. $Fset(h, p)$ stops collecting failure notifications when it reaches the first done notification.

Assume we have $F \in hdl(G, \phi)$ and $F = \cup\{A \mid A \in dom(H) \wedge {F_s}' \subset A \subseteq Fset(h, p)\}$. Since $Fset(h, p) \subseteq F_q$, immediately rule **(IssueDone)** is violated. So we have a contradiction. □

Example 11 (Example reduction for the LR model)**.** *This example provides a partial reduction for the processes illustrated in Figures 3.6 and 3.7. The reduction covers interactions from the default behavior of the LR model. A failure of w_1 will interrupt the task and all live processes will switch to the failure handling for w_1.*

In the reductions below, we write over the arrow the main rule that is used. For the sake of brevity, we may leave parts of the network unspecified. We will now start sketching the reduction.

$$P_{drv} \mid P_{w_1} \mid P_{w_2} \xrightarrow{\textbf{(Link)}} \underbrace{(\nu s)\left(\begin{array}{l} G :_s (\emptyset,\emptyset) \blacklozenge s[drv] : \left(\eta_{drv} \blacktriangleright H_{drv}\right)^{(1,\emptyset)} \mid \\ s[w_1] : \left(\eta_{w_1} \blacktriangleright H_{w_1}\right)^{(1,\emptyset)} \mid s[w_2] : \left(\eta_{w_2} \blacktriangleright H_{w_2}\right)^{(1,\emptyset)} \mid s : \emptyset \end{array}\right)}_{N_1} \tag{3.3}$$

In Equation (3.3) *the* **(Link)** *reduction initializes the system N_1.* **(Link)** *creates a new session s, hides the new session (νs), replaces the session variables with the new session name and role names, e.g., $s[drv]$ for the driver, creates the session queue $s : \emptyset$, and assigns a coordinator $G :_s (\emptyset,\emptyset)$, where G is the global type from Figure 3.2.*

$$N_1 \xrightarrow{\textbf{(Rec)}}\xrightarrow{\textbf{(Rec)}}\xrightarrow{\textbf{(Rec)}}\xrightarrow{\textbf{(Snd)}}\xrightarrow{\textbf{(Snd)}} N_2 \quad N_2 = (\nu s)(G :_s (\emptyset,\emptyset) \blacklozenge N_2') \text{ where}$$

$$N_2' = \begin{array}{l} s[drv] : \underbrace{\left(\begin{array}{l} \textsf{def } X(x_w) = \ldots \textsf{ in} \\ \quad w_1?\ l_{grad}(x_{res1}). \\ \quad w_2?\ l_{grad}(x_{res2}). \\ \quad X\langle calcW(x_w, x_{res1}, x_{res2})\rangle \end{array} \blacktriangleright H_{drv}\right)^{(1,\emptyset)}}_{\eta'_{drv}} \mid s[w_1] : \eta'_{w_1} \mid s[w_2] : \eta'_{w_2} \mid \\ s : \langle drv, w_1, l_{points}(v_1)\rangle \cdot \langle drv, w_2, l_{points}(v_2)\rangle \end{array} \tag{3.4}$$

In the reductions in Equation (3.4)*, the driver and the workers call their recursive definitions (X). Furthermore, the driver sends $\langle drv, w_1, l_{points}(v_1)\rangle$ to w_1, and $\langle drv, w_2, l_{points}(v_2)\rangle$ to w_2.*

$$N_2 \xrightarrow{\textbf{(Crash)}} N_3 \text{ where}$$
$$N_3 = (\nu s)(G :_s (\emptyset,\emptyset) \blacklozenge s[drv] : \eta'_{drv} \mid s[w_2] : \eta'_{w_2} \mid s : \langle drv, w_2, l_{points}(v_2)\rangle \cdot \langle\!\langle \psi, \{w_1\}\rangle\!\rangle) \tag{3.5}$$

In the reduction in Equation (3.5)*, the participant w_1 fails by crashing. This reduction removes the process of w_1, removes all messages in the queue heading to or from w_1, and adds a failure notification heading to the coordinator. The message removal models that reliable message delivery is only ensured for live participants. Furthermore, the added failure notification*

models that, the failure will only be observed eventually (when the coordinator receives the notification).

$$
\begin{array}{c}
N_3 \xrightarrow{\textbf{(Rcv)}} \xrightarrow{\textbf{(Snd)}} N_4 \quad N_4 = (\nu s)(G :_s (\emptyset, \emptyset) \blacklozenge N_4') \textit{ where} \\
N_4' = s[drv] : \eta_{drv}' \mid s[w_2] : \underbrace{\left(\begin{array}{l} \textit{def } X(x_w) = \ldots \textit{ in} \\ \quad X\langle 0 \rangle \end{array} \blacktriangleright H_{w_2} \right)^{(1,\emptyset)}}_{\eta_{w_2}''} \mid \\
s : \langle\!\langle \psi, \{w_1\} \rangle\!\rangle \cdot \langle w_2, drv, l_{grad}(v_g) \rangle
\end{array}
\tag{3.6}
$$

In Equation (3.6) *worker* w_2 *receives the message* $\langle drv, w_2, l_{points}(v_2) \rangle$*, calculates the model update, and sends the result to the driver* drv *(*$\langle w_2, drv, l_{grad}(v_g) \rangle$*). These reductions show that even if a failure occurs other participants progress as normal (until they are aware of the failure – via a failure notification from the coordinator).*

$$
\begin{array}{c}
N_4 \xrightarrow{\textbf{(F)}} N_5 \textit{ where} \\
N_5 = \begin{array}{l} (\nu s)(G :_s (\{w_1\}, \emptyset) \blacklozenge s[drv] : \eta_{drv}' \mid s[w_2] : \eta_{w_2}'' \mid \\ \qquad s : \langle w_2, drv, l_{grad}(v_g) \rangle \cdot \langle\!\langle drv, \{w_1\} \rangle\!\rangle \cdot \langle\!\langle w_2, \{w_1\} \rangle\!\rangle) \end{array}
\end{array}
\tag{3.7}
$$

In Equation (3.7) *the coordinator collects the failure notification and informs all live participants (*drv *and* w_2*) about the failure of* w_1*.*

$$
\begin{array}{c}
N_5 \xrightarrow{\textbf{(TryHdl)}} \xrightarrow{\textbf{(TryHdl)}} N_6 \textit{ where} \\
N_5 = (\nu s)(G :_s (\{w_1\}, \emptyset) \blacklozenge s[drv] : (\eta_1 \blacktriangleright H_{drv})^{(1,\{w_1\})} \mid s[w_2] : (\eta_2 \blacktriangleright H_{w_2})^{(1,\{w_1\})} \mid \\
\qquad s : \langle w_2, drv, l_{grad}(v_g) \rangle \cdot \langle\!\langle drv, \{w_1\} \rangle\!\rangle \cdot \langle\!\langle w_2, \{w_1\} \rangle\!\rangle)
\end{array}
\tag{3.8}
$$

where

$$
\eta_1 = \begin{array}{l} \textit{def } X_{w_1}(x_w) = \\ \quad w_2!\ l_{pointsw_1}(x_w, getPoints(w_1, w_2)). \\ \quad w_2?\ l_{gradw_1}(x_{res1}). \\ \quad X_{w_1}\langle calcW(x_w, x_{res1}) \rangle \textit{ in} \\ X_{w_1}\langle x_{wi} \rangle \end{array}
\qquad
\eta_2 = \begin{array}{l} \textit{def } X_{w_1}(x) = \\ \quad drv?\ l_{pointsw_1}(x_w, x_{points}). \\ \quad drv!\ l_{gradw_1}(calcG(x_w, x_{points})). \\ \quad X_{w_1}\langle 0 \rangle \\ \textit{in } X_{w_1}\langle 0 \rangle \end{array}
\tag{3.9}
$$

In Equation (3.8) *both* w_2 *and* drv *activate failure handling. The try-handle level changes to*

$$
\begin{array}{rcl}
L & ::= & p!\{l_i(S_i).L_i\}_{i\in I} \mid p?\{l_i(S_i).L_i\}_{i\in I} \mid \left(L \blacktriangleright H^{\downarrow}\right)^{\phi}.L \mid t \mid \mu t.L \mid \mathsf{end} \mid \underline{\mathsf{end}} \\
H^{\downarrow} & ::= & F : L \mid H^{\downarrow}, H^{\downarrow}
\end{array}
$$

Figure 3.11.: The grammar of local types.

$(1, \{w_1\})$ *and the default block gets replaced by the handler body for the failure* $\{w_1\}$.

$$
N_6 \xrightarrow{\textbf{(Cln)}} N_7 \text{ where}
$$

$$
\begin{aligned}
N_7 = (\nu s)(G :_s (\{w_1\}, \emptyset) \blacklozenge s[drv] : (\eta_1 \blacktriangleright H_{drv})^{(1,\{w_1\})} \mid s[w_2] : (\eta_2 \blacktriangleright H_{w_2})^{(1,\{w_1\})} \mid \\
s : \langle\!\langle drv, \{w_1\}\rangle\!\rangle \cdot \langle\!\langle w_2, \{w_1\}\rangle\!\rangle)
\end{aligned} \tag{3.10}
$$

In Equation (3.10) drv *removes the message* $\langle w_2, drv, l_{grad}(v_g)\rangle$ *from the queue. The message was sent before failure activation and failure handling made this message obsolete. From* N_7 *onward* drv *and* w_2 *iteratively improve the model parameter unless a failure of* w_2 *leads to termination.*

3.5. Local Types

Figure 3.11 defines local types for typing behaviors of endpoint processes with failure handling. In contrast with global types, which describe global behavior, local types describe the endpoint behavior of a process. Local types can be derived from global type via projection which we will describe shortly. Type $p!\{l_i(S_i).L_i\}_{i\in I}$ is the primitive for a selection and $p?\{l_i(S_i).L_i\}_{i\in I}$ is the branching counterpart, describing the process perspective of the global type $p \to q\{l_i(S_i).G_i\}_{i\in I}$. $\left(L \blacktriangleright H^{\downarrow}\right)^{\phi}.L$ describes a try-handle process. t and $\mu t.L$ type recursion. end types the inactive process and $\underline{\mathsf{end}}$ types an inactive process in a try-handle that expects a done notification. Note that type $\underline{\mathsf{end}}$ only appears in *runtime* type checking.

Below we define $G{\restriction}p$ to project a global type G on p, thus generating p's local type.

Definition 10 (Projection)**.** *Consider a well-formed top-level global type* $[\tilde{q}]G$*. Then* $G{\restriction}p$ *is defined as follows:*

$$
\begin{aligned}
&(1)\ G{\restriction}p \text{ where } G = \left(G_0 \blacktriangleright F_1 : G_1, ..., F_n : G_n\right)^{\kappa}.G' \\
&= \begin{cases}
\left(G_0{\restriction}p \blacktriangleright F_1 : G_1{\restriction}p, ..., F_n : G_n{\restriction}p\right)^{(\kappa,\emptyset)}.G'{\restriction}p & \text{if } p \in \mathit{roles}(\kappa) \\
\qquad \forall i \in \{1, .., n\}.\ \nexists\left(G'' \blacktriangleright H^{\uparrow}\right) \in G_i & \\
G'{\restriction}p & \text{elseif } p \notin \mathit{roles}(\kappa)
\end{cases}
\end{aligned}
$$

$$(2)\ p_1 \to p_2\{l_i(S_i).G_i\}_{i\in I}\upharpoonright p = \begin{cases} p_2!\{l_i(S_i).G_i\upharpoonright p\}_{i\in I} & \textit{if } p = p_1 \\ p_1?\{l_i(S_i).G_i\upharpoonright p\}_{i\in I} & \textit{elseif } p = p_2 \\ G_1\upharpoonright p & \textit{elseif } \forall i,j \in I. G_i\upharpoonright p = G_j\upharpoonright p \end{cases}$$

$$(3)\ (\mu t.G)\upharpoonright p = \{\mu t.(G\upharpoonright p) \quad \textit{if } \nexists\big(G' \blacktriangleright H^{\uparrow}\big) \in G \ \wedge\ G\upharpoonright p \neq t \qquad \textsf{end} \quad \textit{elseif } G\upharpoonright p = t\}$$

$$(4)\ t\upharpoonright p = t \quad (5)\ \textsf{end}\upharpoonright p = \textsf{end}$$

Otherwise it is undefined.

The main rule is (1): if p appears somewhere in the target try-handle global type ($p \in \mathit{roles}(\kappa)$) then the endpoint type has a try-handle annotated with κ and the default logic (i.e., $F_s = \emptyset$). Note that even if $G_0 \upharpoonright p = \textsf{end}$ the endpoint still gets such a try-handle because it needs to be ready for (possible) failure handling; if p does not appear anywhere in the target try-handle global type, then the projection skips to the continuation. Furthermore, the rule ensures that nested try-handles occur only in the default block.

Rule (2) produces local types for interaction endpoints. If the endpoint is a sender (i.e., $p = p_1$), then its local type abstracts that it will send something from one of the possible internal choices defined in $\{l_i(S_i)\}_{i\in I}$ to p_2, then continue as $G_k \upharpoonright p$, gained from the projection, if $k \in I$ is chosen. If the endpoint is a receiver (i.e., $p = p_2$), then its local type abstracts that it will receive something from one of the possible external choices defined in $\{l_i(S_i)\}_{i\in I}$ sent by p_1; the rest is the same as for the sender. However, if p is not in this interaction, then its local type starts from the next interaction which includes p; moreover, because p does not know what choice the sender, p_1, has made, the p's behavior $G_i\upharpoonright p$ regarding any choice l_i shall be the same. This ensures that interactions are consistent. For example, in $G = p_1 \to p_2\{l_1(S_1).p_3 \to p_1\ l_3(S),\ l_2(S_2).p_3 \to p_1\ l_4(S)\}$, interaction $p_3 \to p_1$ continues after $p_1 \to p_2$ takes place. If $l_3 \neq l_4$, then G is not projectable for p_3 because p_3 does not know which branch p_1 has chosen; if p_1 chooses branch l_1, but p_3 (blindly) sends out label l_4 to p_1, for p_1 it is a mistake (but it is not a mistake for p_3) because p_1 is expecting to receive label l_3. To prevent such inconsistencies, we adopt the projection algorithm proposed in Honda et al. [HYC16]. Other session type works [DY11, SDHY17b] provide ways to weaken the classical restriction on projection of branching which we use.

Rule (3) forbids a try-handle to appear in a recursive body, e.g., $\mu t.\big(G \blacktriangleright F : t\big)^{\kappa}.G$ is not allowed, but $\big(\mu t.G \blacktriangleright H^{\uparrow}\big)^{\kappa}$ and $\big(G \blacktriangleright F : \mu t.G', H^{\uparrow}\big)^{\kappa}$ are allowed. This is because κ is used to avoid confusion of messages from different try-handles. If a recursive body contains a try-handle, we have to dynamically generate different levels to maintain interaction consistency. Other rules are straightforward.

Example 12. *Recall the global type G from Figure 3.2 in Section 3.1. Applying projection*

rules defined in Definition 10 to every role in G we obtain the following:

$$
\begin{array}{rcl}
L_{drv} = G\upharpoonright drv & = & (L'_{drv} \blacktriangleright H^{\downarrow}_{drv})^{(1,\emptyset)} \\
L'_{drv} & = & \mu t.w_1!l_{points}(S).\, w_2!l_{points}(S).\, w_1?l_{grad}(S').\, w_2?l_{grad}(S').\, t \\
H^{\downarrow}_{drv} & = & \{w_1\} : \mu t_1.w_2!l_{pointsw_1}(S).\, w_2?l_{gradw_1}(S').\, t_1, \\
 & & \{w_2\} : \mu t_2.\, w_1!l_{pointsw_2}(S).\, w_1?l_{gradw_2}(S').\, t_2, \{w_1, w_2\} : \mathsf{end} \\
L_{w_1} = G\upharpoonright w_1 & = & (L'_{w_1} \blacktriangleright H^{\downarrow}_{w_1})^{(1,\emptyset)} \\
L'_{w_1} & = & \mu t.\, drv?l_{points}(S).\, drv!l_{grad}(S').\, t \\
H^{\downarrow}_{w_1} & = & \{w_1\} : \mathsf{end}, \{w_2\} : \mu t_2.\, drv?l_{pointsw_2}(S).\, drv!l_{gradw_2}(S').\, t_2, \{w_1, w_2\} : \mathsf{end} \\
L_{w_2} = G\upharpoonright w_2 & = & (L'_{w_2} \blacktriangleright H^{\downarrow}_{w_2})^{(1,\emptyset)} \\
L'_{w_2} & = & \mu t.\, drv?l_{points}(S).\, drv!l_{grad}(S').\, t \\
H^{\downarrow}_{w_2} & = & \{w_1\} : \mu t_1.\, drv?l_{pointsw_1}(S).\, drv!l_{gradw_1}(S').\, t_1, \{w_2\} : \mathsf{end}, \{w_1, w_2\} : \mathsf{end}
\end{array}
$$

3.6. Type System

Next we introduce our type system for typing processes. Figure 3.12 and Figure 3.13 present typing rules for endpoint processes and networks.

3.6.1. Typing environments and rules

We define shared environments Γ to keep information on variables and the coordinator, and session environments Δ to keep information on endpoint types:

$$
\begin{array}{rcl}
\Gamma & ::= & \emptyset \mid \Gamma, X : S\ L \mid \Gamma, x : S \mid \Gamma, a : G \mid \Gamma, \Psi \\
\mathtt{m} & ::= & \langle p, q, l(S)\rangle \mid \llparenthesis p, F \rrparenthesis \mid \langle p, q\rangle^{\phi}
\end{array}
\qquad
\begin{array}{rcl}
\Delta & ::= & \emptyset \mid \Delta, c : L \mid \Delta, s : \mathtt{h} \\
\mathtt{h} & ::= & \emptyset \mid \mathtt{h} \cdot \mathtt{m}
\end{array}
$$

Γ maps process variables X and content variables x to their types, shared names a to global types G, and a coordinator $\Psi = G :_c (F_q, h_d)$ to failures and done notifications that it has observed. Δ maps session channels c to local types and session queues to queue types. We write $\Gamma, \Gamma' = \Gamma \cup \Gamma'$ when $dom(\Gamma) \cap dom(\Gamma') = \emptyset$; same for Δ, Δ'. Queue types $\mathtt{h}$ are composed of message types $\mathtt{m}$. Their permutation is defined analogously to the permutation for messages (see Definition 22). Figure 3.12 and Figure 3.13 use the following typing judgment:

$$\Gamma \vdash N \rhd \Delta$$

which states N is well-typed by Δ under Γ. Figure 3.12 defines the typing rules for (local) processes and Figure 3.13 defines the typing rules for networks.

Figure 3.12 lists our typing rules for endpoint processes. Rule $\lfloor$T-ini$\rfloor$ says that if a process's actions on the session variable y are well-typed by $G\upharpoonright p$, then this process

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \rhd \{y : G\upharpoonright p\}}{\Gamma \vdash a[p](y).P \rhd \emptyset} \qquad \frac{k \in I \quad \Gamma \vdash e : S_k \quad \Gamma \vdash c : \eta_k \rhd \{c : L_k\}}{\Gamma \vdash c : p!\, l_k(e).\eta_k \rhd \{c : p!\, \{l_i(S_i).L_i\}_{i \in I}\}} \quad \lfloor \texttt{T-ini/T-snd} \rfloor$$

$$\frac{\forall i \in I.\, \Gamma, x_i : S_i \vdash c : \eta_i \rhd \{c : L_i\}}{\Gamma \vdash c : p?\, \{l_i(x_i).\eta_i\}_{i \in I} \rhd \{c : p?\, \{l_i(S_i).L_i\}_{i \in I}\}} \quad \lfloor \texttt{T-rcv} \rfloor$$

$$\Gamma \vdash c : 0 \rhd \{c : \mathsf{end}\} \qquad \Gamma \vdash c : \underline{0} \rhd \{c : \underline{\mathsf{end}}\} \quad \lfloor \texttt{T-0/T-yd} \rfloor$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \forall i \in \{1,2\}.\, \Gamma \vdash c : \eta_i \rhd \Delta}{\Gamma \vdash c : \mathsf{if}\ e\ \eta_1\ \mathsf{else}\ \eta_2 \rhd \Delta} \qquad \frac{\Gamma \vdash e : S}{\Gamma, X : S\ L \vdash c : X\langle e \rangle \rhd \{c : L\}} \quad \lfloor \texttt{T-if/T-var} \rfloor$$

$$\frac{\Gamma, X : S\ t, x : S \vdash c : \eta_1 \rhd \{c : L'\} \quad \Gamma, X : S\ \mu t.L' \vdash c : \eta_2 \rhd \{c : L\}}{\Gamma \vdash c : \mathsf{def}\ X(x) = \eta_1\ \mathsf{in}\ \eta_2 \rhd \{c : L\}} \quad \lfloor \texttt{T-def} \rfloor$$

$$\frac{\Gamma \vdash c : \eta \rhd \{c : L\} \quad \Gamma \vdash c : \eta' \rhd \{c : L'\} \quad dom(H) = dom(H^{\downarrow}) \quad \forall F \in dom(H).\, \Gamma \vdash c : H(F) \rhd \{c : H^{\downarrow}(F)\}}{\Gamma \vdash c : (\eta \blacktriangleright H)^{\phi}.\eta' \rhd \{c : (L \blacktriangleright H^{\downarrow})^{\phi}.L'\}} \quad \lfloor \texttt{T-th} \rfloor$$

Figure 3.12.: Typing rules for processes.

can play role p in a. $\langle G \rangle$ means that G is closed, i.e., devoid of type variables. This rule forbids $a[p].b[q].P$ because a process can only use one session channel. Since we do not define sequential composition for processes $\lfloor$`T-ini`$\rfloor$ implicitly forbids session interleaving. This is a difference compared to other MPSTs works [CDYP16, HYC16], where session interleaving is prohibited for the progress property; here the restriction is inherent in the type system. Rule $\lfloor$`T-snd`$\rfloor$ states that an action for sending is well-typed to a sending type if the label and the type of the content are expected; $\lfloor$`T-rcv`$\rfloor$ states that an action for branching (i.e., for receiving) is well-typed to a branching type if all labels and the types of contents are as expected. Their follow-up actions shall also be well-typed. Rule $\lfloor$`T-0`$\rfloor$ types an idle process with `end`. Rule $\lfloor$`T-yd`$\rfloor$ types yielding actions, which only appear at runtime. Rule $\lfloor$`T-if`$\rfloor$ is standard in the sense that the process is well-typed by Δ if e has boolean type and its sub-processes (i.e., η_1 and η_2) are well-typed by Δ. Rules $\lfloor$`T-var,T-def`$\rfloor$ type recursion. Rule $\lfloor$`T-th`$\rfloor$ states that a try-handle is well-typed if it is annotated with the expected level ϕ, its default statement is well-typed, $H^{\downarrow}$ and H have the same handler signatures, and all handling actions are well-typed.

Figure 3.13 shows typing rules for networks. Rule $\lfloor$`T-`$\emptyset\rfloor$ types an empty queue. Rules $\lfloor$`T-m,T-D,T-`$F\rfloor$ simply type messages based on their shapes. Rule $\lfloor$`T-pa`$\rfloor$ says two networks composed in parallel are well-typed if they do not share any session channel.

$$\Gamma \vdash s : \emptyset \rhd \{s : \mathtt{h}\} \qquad \frac{\Gamma \vdash s : h \rhd \{s : \mathtt{h}\} \qquad \Gamma \vdash v : S}{\Gamma \vdash s : h \cdot \langle p, q, l(v)\rangle \rhd \{s : \mathtt{h} \cdot \langle p, q, l(S)\rangle\}} \quad \lfloor \mathtt{T\text{-}\emptyset/T\text{-}m} \rfloor$$

$$\frac{(p_1, p_2) \in \{(p, \psi), (\psi, p)\} \qquad \Gamma \vdash s : h \rhd \{s : \mathtt{h}\}}{\Gamma \vdash s : h \cdot \langle p_1, p_2\rangle^{\phi} \rhd \{s : \mathtt{h} \cdot \langle p_1, p_2\rangle^{\phi}\}} \qquad \Gamma \vdash 0 \rhd \emptyset \quad \lfloor \mathtt{T\text{-}D/T\text{-}N0} \rfloor$$

$$\frac{p \in \{q, \psi\} \qquad \mathtt{m} = \llbracket p, F \rrbracket \qquad \Gamma \vdash s : h \rhd \{s : \mathtt{h}\}}{\Gamma \vdash s : h \cdot \llbracket p, F \rrbracket \rhd \{s : \mathtt{h} \cdot \mathtt{m}\}} \qquad \frac{\Gamma \vdash N_1 \rhd \Delta_1 \qquad \Gamma \vdash N_2 \rhd \Delta_2 \qquad dom(\Delta_1) \cap dom(\Delta_2) = \emptyset}{\Gamma \vdash N_1 \mid N_2 \rhd \Delta_1, \Delta_2} \quad \lfloor \mathtt{T\text{-}}F\mathtt{/T\text{-}pa} \rfloor$$

$$\frac{\Gamma \vdash N \rhd \Delta \qquad \Gamma \vdash \Delta_s \text{ coherent}}{\Gamma \vdash (\nu s)N \rhd \Delta \setminus \Delta_s} \qquad \frac{\Gamma' = \Gamma, \Psi \qquad \Gamma \vdash N \rhd \Delta}{\Gamma' \vdash \Psi \blacklozenge N \rhd \Delta} \quad \lfloor \mathtt{T\text{-}s/T\text{-}sys} \rfloor$$

Figure 3.13.: Typing rules for networks.

Rule $\lfloor$T-N0$\rfloor$ types an idle network. Rule $\lfloor$T-s$\rfloor$ states a session hiding, $(\nu s)N$, is well-typed, if the network, N, in which s is hidden is well-typed under environments Γ and Δ that are coherent for s ($\Gamma \vdash \Delta_s$ *coherent*); then $(\nu s)N$ gets typed under the session environment $\Delta \setminus \Delta_s$, i.e., the session environment Δ without local and queue types of s. We say $\Gamma \vdash \Delta$ is coherent under Γ if the local types of all endpoints are dual to each other after their local types are updated according to messages or notifications in $s : \mathtt{h}$. We define coherence shortly.

Definition 11 (A session environment having s only: Δ_s)**.**

$$\Delta_s = \{s[p] : L \mid s[p] \in dom(\Delta)\} \cup \{s : \mathtt{h} \mid s \in dom(\Delta)\}$$

Rule $\lfloor$T-sys$\rfloor$ says that a system $\Psi \blacklozenge N$ is well-typed if network N is well-typed and there is a coordinator Ψ for handling this network.

Example 13. *In this example we show that the driver process from Figure 3.6 is well-typed.*

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \underbrace{\Gamma \vdash y : \left(\eta_{drv} \blacktriangleright H_{drv}\right)^{(1,\emptyset)}.0 \rhd \{y : G\upharpoonright drv\}}_{T}}{\underbrace{a : G, x_{wi} : S''}_{\Gamma} \vdash a[drv](y).y : \left(\eta_{drv} \blacktriangleright H_{drv}\right)^{(1,\emptyset)}.0 \rhd \emptyset} \tag{3.11}$$

In Equation (3.11) *we type the driver process, via* $\lfloor$*T-ini*$\rfloor$*, using an empty session environment and a shared environment containing the type of the shared name* a *and the variable* x_{wi}*. We show in* T *(see Equation* (3.12)*) that the process uses* y *as described by the global*

type projected to drv. See Example 12 for $G{\upharpoonright}drv$.

$$\frac{\underbrace{\Gamma \vdash\ y : \eta_{drv} \rhd\ L'_{drv}}_{T_1} \quad \forall i \in \{1,2\}.\ \underbrace{\Gamma \vdash\ y : H_{drv}(\{w_i\}) \rhd\ \{H^{\downarrow}_{drv}(\{w_i\})\}}_{T_{w_i}} \quad \Gamma \vdash\ y : 0 \rhd\ \{y : \underbrace{H^{\downarrow}_{drv}(\{w_1, w_2\})}_{\mathsf{end}}\}}{\Gamma \vdash\ y : \left(\eta_{drv} \blacktriangleright H_{drv}\right)^{\phi}.0 \rhd\ \{y : (L'_{drv} \blacktriangleright H^{\downarrow}_{drv})^{(1,\emptyset)}.\mathsf{end}\}} \tag{3.12}$$

Equation (3.12) *shows the type derivation T, i.e. the typing of a try-handle process, via $\lfloor$T-th$\rfloor$; for that we type the continuation and the handler body for $\{w_1, w_2\}$ with* end*, both are idle (0); furthermore, we type the default activity η in T_1 and the two remaining handler bodies for $\{w_1\}$ and $\{w_2\}$ in T_{w_1} and T_{w_2}. We do not show the type derivation for T_{w_1} and T_{w_2} because they are similar and simpler than T_1. Equation* (3.12) *can only be typed if $(L'_{drv} \blacktriangleright H^{\downarrow}_{drv})^{(1,\emptyset)}.\mathsf{end} = G{\upharpoonright}drv$ (See Example 12).*

$$\frac{\underbrace{\Gamma, X : S''\ t, x_w : S'' \vdash\ \eta_X \rhd\ \{y : L\}}_{T_2} \quad \frac{\Gamma \vdash\ x_{wi} : S''}{\Gamma, X : S''\ \mu t.L \vdash\ y : X\langle x_{wi}\rangle \rhd\ \{y : \mu t.L\}}}{\Gamma \vdash\ y : \mathsf{def}\ X(x_w) = \eta_X\ \mathsf{in}\ X\langle x_{wi}\rangle \rhd\ \{c : \mu t.L\}} \tag{3.13}$$

Equation (3.13) *shows the type derivation T_1, i.e. the typing of the default activity, which is a recursive definition (X), which we type via $\lfloor$T-def$\rfloor$; the definition is immediately called. We type the call X simply via $\lfloor$T-var$\rfloor$ and we must ensure that we can type the definition with $y : L$ such that $\mu t.L = L'_{drv}$. We show the typing of the recursive definition in T_2.*

$$\frac{\Gamma \vdash\ e : S \quad \frac{\Gamma \vdash\ e' : S \quad \underbrace{\Gamma, X : S''\ t, x_w : S'' \vdash\ y : \eta''_X \rhd\ \{y : L''\}}_{T_3}}{\Gamma, X : S''\ t, x_w : S'' \vdash\ y : w_2!\ l_{points}(e').\eta''_X \rhd\ \{y : w_2!l_{points}(S).L''\}}}{\Gamma, X : S''\ t, x_w : S'' \vdash\ y : \underbrace{w_1!\ l_{points}(e).\eta'_X}_{\eta_X} \rhd \{y : w_1!l_{points}(S).w_2!l_{points}(S).L''\}} \tag{3.14}$$

Equation (3.14) *shows the type derivation T_2, i.e. the typing of the body of the recursive definition. We type the first two sends in the body, via $\lfloor$T-snd$\rfloor$ twice, with the session environment $\{y : w_1!l_{points}(S).w_2!l_{points}(S).L''\}$. What remains to be shown is that $y : \eta''_X$*

has the type $w_1?l_{grad}(S').w_2?l_{grad}(S').\ t$*, which we do in* T_3*.*

$$\frac{\dfrac{\dfrac{\Gamma, X : S''\ t, x_w : S'', x_{res1}, x_{res2} : S' \vdash\ calcW(x_w, x_{res1}, x_{res2}) : S''}{\Gamma, X : S''\ t, x_w : S'', x_{res1}, x_{res2} : S' \vdash\ y : X\langle calcW(x_w, x_{res1}, x_{res2})\rangle \rhd \{y : t\}}}{\Gamma, X : S''\ t, x_w : S'', x_{res1} : S' \vdash\ y : w_2?\ l_{grad}(x_{res2}).\ \eta_X'''' \rhd \{y : w_2?l_{grad}(S').\ t\}}}{\Gamma, X : S''\ t, x_w : S'' \vdash\ y : w_1?\ l_{grad}(x_{res1}).\ \eta_X''' \rhd \{y : \underbrace{w_1?l_{grad}(S').w_2?l_{grad}(S').\ t}_{L''}\}} \tag{3.15}$$

Equation (3.14) *shows the type derivation* T_3*, i.e. the typing of the remaining two receives followed by the recursive call of* X *with the session environment* $\{w_1?l_{grad}(S').w_2?l_{grad}(S').\ t\}$ *as desired. The type derivation first uses* $\lfloor$`T-rcv`$\rfloor$ *twice and then* $\lfloor$`T-var`$\rfloor$ *once.*

3.6.2. Coherence

We say that a session environment is coherent if, at any time, given a session with its latest messages and notifications, every endpoint participating in it can find someone to interact with (i.e., its dual party exists) right now or later. To check whether two endpoints, say drv and w_2 (from Example 12), can interact safely (are dual), we compare their types; actions described in one must have a matching action in the other. E.g., if one type contains a selection, the other type must contain a matching branching (branching is the dual-action to selection). However, e.g., the type of drv describes actions with both w_2 and w_1. Recall the local type of drv, from Example 12, $(w_1!l_{points}(S).w_2!l_{points}(S). \ldots \blacktriangleright H^{\downarrow}_{drv})^{(1,\emptyset)}$ and the local type of w_2 $(drv?l_{points}(S). \ldots \blacktriangleright H^{\downarrow}_{w_2})^{(1,\emptyset)}$. These types describe safe interactions even if w_2 does not receive any message from drv at this point. Only after drv sends out a message to w_1 will drv send a message to w_2. Following Coppo et al. [CDYP16], we define a second projection operation to extract the parts of a local type that define the interactions towards a specific participant (see Definition 12). Furthermore, we need to consider message types to ensure that two endpoints can interact safely. Let us consider, e.g., N_2' from the LR reduction example (Example 11, Equation (3.4)). We can type N_2', $\Gamma' \vdash\ N_2' \rhd\ \Delta'$ where

$$\begin{array}{rcl}
\Delta' & = & s[drv] : (w_1?l_{grad}(S').\ w_2?l_{grad}(S').\ L'_{drv} \blacktriangleright H^{\downarrow}_{drv})^{(1,\emptyset)}, \\
 & & s[w_1] : (drv?l_{points}(S).\ drv!l_{grad}(S').\ L'_{w_1} \blacktriangleright H^{\downarrow}_{w_1})^{(1,\emptyset)}, \\
 & & s[w_2] : (drv?l_{points}(S).\ drv!l_{grad}(S').\ L'_{w_2} \blacktriangleright H^{\downarrow}_{w_2})^{(1,\emptyset)}, \\
 & & s : \langle drv, w_1, l_{points}(S)\rangle \cdot \langle drv, w_2, l_{points}(S)\rangle
\end{array}$$

In Δ' the types of drv, w_1 and w_2 all have branching type prefixes in the default block, but a branching type requires a dual selection type. The effect of, e.g., the message

type $\langle drv, w_1, l_{points}(S)\rangle$ on w_1's type is to remove the branching prefix, which gives us balanced inputs and outputs between w_1 and drv. We define the effect of message types shortly (see Definition 14). Next we define when two types are dual to each other, i.e., all actions described by one type have matching counterparts in the other (see Definition 15). Lastly, we define the coherence invariant, which ensures that the typed processes can communicate safely (see Definition 16).

Behavior and messages for p

We now define the process of extracting, from a local type, the behavior towards a specific role.

Definition 12 ($L \lfloor p$ (Behaviors for p in L)).

$$
\begin{aligned}
&\big(L \blacktriangleright F_1 : L_1, ..., F_n : L_n\big)^{\phi}.L' \lfloor p = \\
&\quad \begin{cases} \big(L \lfloor p \blacktriangleright F_1 : L_1 \lfloor p, ..., F_n : L_n \lfloor p\big)^{\phi}.L' \lfloor p & \textit{if } p \in \textit{roles}(\kappa) \textit{ where } \phi = (\kappa, F_s) \\ L' \lfloor p & \textit{otherwise} \end{cases} \\
&q!\{l_i(S_i).L_i\}_{i\in I} \lfloor p = \begin{cases} p!\{l_i(S_i).L_i \lfloor p\}_{i\in I} & \textit{if } p = q \\ L_1 \lfloor p & \textit{otherwise} \end{cases} \\
&q?\{l_i(S_i).L_i\}_{i\in I} \lfloor p = \begin{cases} p?\{l_i(S_i).L_i \lfloor p\}_{i\in I} & \textit{if } p = q \\ L_1 \lfloor p & \textit{otherwise} \end{cases} \\
&\underline{\textit{end}} \lfloor p = \underline{\textit{end}} \quad \textit{end} \lfloor p = \textit{end} \quad t \lfloor p = t \quad \mu t.L \lfloor p = \begin{cases} \mu t.L \lfloor p & \textit{if } L \lfloor p \neq t \\ \textit{end} & \textit{otherwise} \end{cases}
\end{aligned}
$$

In general, the rules keep a local type if it contains the role p. The first rule uses $\textit{roles}(\kappa)$ to extract all roles in the try-handle κ in G, where G is the global type that this session follows. It keeps a try-handle if p occurs in the try-handle κ and filters the continuation. The rule uses the global type instead of the local type because, e.g., during done synchronization, a role may be involved in a try-handle in which it no longer syntactically occurs. The following two rules keep a selection or branching type where p is the receiver or sender and filter all continuations. If p is neither the receiver nor the sender, these two rules filter one continuation. To filter only one continuation is safe because of the final case of Definition 10 (Projection) clause (2); if p is neither the sender nor the receiver, then the action of p shall be the same for any branch, e.g., $p_1!\{l_1(S_1).p_2?l(S), l_2(S_2).p_2?l(S)\} \lfloor p_2 = p_2?l(S)$. The last rules handle end and recursion in the expected way.

Next we define $(\mathtt{h})_{p\to q}$ to filter $\mathtt{h}$ to generate (1) the normal message types sent from p heading to q, and (2) the notifications heading to q.

$$\frac{\forall m \in \mathtt{ht}.\, \exists F.\, m = \llbracket F \rrbracket \quad \text{no other rule is applicable}}{L - \mathtt{ht} = L} \qquad \frac{l \notin \mathit{labels}(L)}{L - q?l(S_k) \cdot \mathtt{ht} = L - \mathtt{ht}}$$

$$\frac{\phi \notin L \quad \langle \psi \rangle^{\phi} \in \mathtt{ht}}{L - \mathtt{ht} = L - \mathtt{ht} \setminus \langle \psi \rangle^{\phi}} \qquad \frac{F' = \cup\{A \mid A \in \mathit{dom}(H^{\downarrow}) \wedge F \subset A \subseteq \mathit{Fset}(\mathtt{ht}, p)\} \quad F' : L' \in H^{\downarrow}}{\mathcal{E}[(L \blacktriangleright H^{\downarrow})^{(\kappa, F)}.L''] - \mathtt{ht} = \mathcal{E}[(L' \blacktriangleright H^{\downarrow})^{(\kappa, F')}.L''] - \mathtt{ht}}$$

$$\frac{\langle \psi \rangle^{\phi} \in \mathtt{ht}}{\mathcal{E}[(\underline{\mathsf{end}} \blacktriangleright H^{\downarrow})^{\phi}.L'] - \mathtt{ht} = \mathcal{E}[L'] - (\mathtt{ht} \setminus \langle \psi \rangle^{\phi})} \qquad \frac{k \in I}{\mathcal{E}[q?\{l_i(S_i).T_i\}_{i \in I}] - q?l_k(S_k) \cdot \mathtt{ht} = \mathcal{E}[T_k] - \mathtt{ht}}$$

Figure 3.14.: The Effect of $\mathtt{ht}$ on L.

Definition 13 (Message Types to q)**.** *We define* $(\mathtt{h})_{p \to q}$ *by selecting message types and notifications in* $\mathtt{h}$ *which are (sent from* p*) heading to* q*. Let*

$$\mathtt{mt} ::= p?l(S) \;\mid\; \llbracket F \rrbracket \;\mid\; \langle \psi \rangle^{\phi}$$

and

$$\mathtt{ht} ::= \emptyset \;\mid\; \mathtt{ht} \cdot \mathtt{mt}$$

; then we define $(\mathtt{h})_{p \to q}$ *as follows:*

$$(\emptyset)_{p \to q} = \emptyset \quad (\mathtt{m} \cdot \mathtt{h})_{p \to q} = \begin{cases} p?l(S) \cdot (\mathtt{h})_{p \to q} & \textit{if } \mathtt{m} = \langle p, q, l(S) \rangle \cdot \mathtt{h} \\ \llbracket F \rrbracket \cdot (\mathtt{h})_{p \to q} & \textit{if } \mathtt{m} = \llbracket q, F \rrbracket \cdot \mathtt{h} \\ \langle \psi \rangle^{\phi} \cdot (\mathtt{h})_{p \to q} & \textit{if } \mathtt{m} = \langle \psi, q \rangle^{\phi} \cdot \mathtt{h} \\ (\mathtt{h})_{p \to q} & \textit{otherwise} \end{cases}$$

For example $(\langle p, q, l(S) \rangle \cdot \llbracket q, F \rrbracket \cdot \langle \psi, q \rangle^{\phi} \cdot \llbracket p, F' \rrbracket)_{p \to q} = p?l(S) \cdot \llbracket F \rrbracket \cdot \langle \psi \rangle^{\phi}$. The message types are abbreviated so that they contain essential information only.

Effect of $\mathtt{ht}$ on L

We define $L - \mathtt{ht}$, i.e., the effect of $\mathtt{ht}$ on L, in Definition 14. The concept is similar to the *session remainder* definition [MY15, CVB+16], which returns an updated local type after consuming messages in the global queue. We define typing contexts as $\mathcal{E} ::= [\,] \;\mid\; (\mathcal{E} \blacktriangleright H^{\downarrow})^{\phi}.L,\ \mathit{Fset}(\mathtt{ht}, p)$ analogously to Definition 7, and the permutation of $\mathtt{ht}$ analogously to Definition 6.

Definition 14 (The Effect of $\mathtt{ht}$ on L)**.** *We define* $L - \mathtt{ht}$*, taking into account message type permutations, in Figure 3.14*

The behavior defined in $L - \texttt{ht}$ aligns closely with the network reduction relation (see Figures 3.8 and 3.9). The first rule returns the final L once the effect of $\texttt{ht}$ is calculated; the queue is either empty or contains only failure notifications; remember that a reduction never removes failure notifications sent by the coordinator. The next two rules clean normal message types or done notification types, similar to their reduction counterparts **(RcvDone)** and **(Cln)** (see Figure 3.9). The two rules after that calculate the effect of failure handling activation and leaving a try-handle, similar to their reduction counterparts **(TryHdl)** and **(RcvDone)** (see Figure 3.9). The last rule calculates the effect of receiving a message.

E.g., $\left(q?\{l_i(S_i).T_i\}_{i\in I} \blacktriangleright H^{\downarrow}\right)^{\phi}.L' - q?l_k(S_k) \cdot \texttt{ht} = \left(T_k \blacktriangleright H^{\downarrow}\right)^{\phi}.L' - \texttt{ht}$ where $k \in I$.

Duality

We write $s[p] : L \bowtie s[q] : L'$ to state that actions of the two types are *dual*:

Definition 15 (Duality)**.** *We define $s[p] : L \bowtie s[q] : L'$ as the minimal symmetric relation defined by:*

$$s[p] : \mathsf{end} \bowtie s[q] : \mathsf{end} \qquad s[p] : \underline{\mathsf{end}} \bowtie s[q] : \underline{\mathsf{end}} \qquad s[p] : \mathsf{end} \bowtie s[q] : \underline{\mathsf{end}}$$

$$s[p] : t \bowtie s[q] : t \qquad \frac{s[p] : L \bowtie s[q] : L'}{s[p] : \mu t.L \bowtie s[q] : \mu t.L'} \qquad \frac{\forall i \in I.\ s[p] : L_i \bowtie s[q] : L'_i}{s[p] : q!\ \{l_i(S_i).L_i\}_{i\in I} \bowtie s[q] : p?\ \{l_i(S_i).L'_i\}_{i\in I}}$$

$$\frac{\begin{array}{c} s[p] : L_1 \bowtie s[q] : L_2 \\ s[p] : L'_1 \bowtie s[q] : L'_2 \qquad \mathit{dom}(H_1^{\downarrow}) = \mathit{dom}(H_2^{\downarrow}) \qquad \forall F \in \mathit{dom}(H_1^{\downarrow}).\ s[p] : H_1^{\downarrow}(F) \bowtie s[q] : H_2^{\downarrow}(F) \end{array}}{s[p] : \left(L_1 \blacktriangleright H_1^{\downarrow}\right)^{\phi}.L'_1 \bowtie s[q] : \left(L_2 \blacktriangleright H_2^{\downarrow}\right)^{\phi}.L'_2}$$

The rules on the first line state that end and $\underline{\mathsf{end}}$ are dual to themselves and each other. The next two rules define duality for recursion in the standard way. The rule after that states a selection is dual to a branching if sender and receiver match, both use the same data types and labels, and all continuations are dual. The last rule states two try-handles are dual if they currently handle the same task (ϕ), they handle the same set of failures, and the active handler, the continuation, and all handler bodies are dual.

Coherence

Now we define what it means for Δ to be coherent under Γ:

Definition 16 ($\Gamma \vdash \Delta$ coherent)**.** *$\Gamma \vdash \Delta$ is coherent if the following conditions hold:*

1. *If $s : \mathtt{h} \in \Delta$, then $\exists G :_s (F_q, h_d) \in \Gamma$ and $\{p \mid s[p] \in \mathit{dom}(\Delta)\} \subseteq \mathit{roles}(G)$ and G is well-formed and $\forall p \in \mathit{roles}(G), G{\upharpoonright}p$ is defined.*

2. $\forall s[p] : L, s[q] : L' \in \Delta$ *we have* $s[p] : L \downharpoonleft q - (\mathtt{h})_{q \to p} \bowtie s[q] : L' \downharpoonleft p - (\mathtt{h})_{p \to q}$.

In condition 1, we require a coordinator for the session s so that the coordinator can ensure consistent failure handling. Condition 2 requires that, for any two endpoints, say $s[p]$ and $s[q]$, in Δ, equation $s[p] : L \downharpoonleft q - (\mathtt{h})_{q \to p} \bowtie s[q] : L' \downharpoonleft p - (\mathtt{h})_{p \to q}$, must hold. This condition asserts that interactions of non-failed endpoints are dual to each other after the effect of $\mathtt{h}$. Failed endpoints are removed from Δ, therefore this condition is satisfied immediately for those.

3.7. Properties

We show that our type system ensures the property of subject reduction. The article "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems" [VCE+18] provides a progress property for the calculus.[2] The progress statement is no contribution of this thesis. Therefore, we do not present it here.

Before we present Theorem 2 (Subject Reduction), we introduce the reduction relation over session environments that describes how the environments are updated in relation to asynchronous network reduction. In other words, the reduction relation over session environments mimics the interaction dynamics of network reductions at the type level.

Then we present Theorem 1 (Preservation of Coherence), which establishes that the reduction relation over session environments preserves coherence. The reduction relation over session environments and preservation of coherence is the basis for establishing subject reduction.

Appendix A.2 contains the proofs details, auxiliary formal definitions, and lemmas.

3.7.1. Preservation of coherence

We now define the reduction over session environments,

$$\Psi \vdash \Delta \to_L \Psi' \vdash \Delta'$$

and establish Theorem 1 (Preservation of Coherence).

Definition 17 (Reduction relation over session environments)**.** *We define* $\Psi \vdash \Delta \to_L \Psi' \vdash \Delta'$*, taking into account message type permutations, in Figure 3.15.*

[2]The calculus we present here contains cosmetic changes and minor corrections compared with the calculus presented in our publication [VCE+18].

$$s[p] : \mathcal{E}[q!\{l_i(S_i).L_i\}_{i\in I}], s : \mathtt{h} \to_L s[p] : \mathcal{E}[L_k], s : \mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle \quad k \in I \qquad [\![\mathtt{Snd}]\!]$$

$$s[p] : \mathcal{E}[q?\{l_i(S_i).L_i\}_{i\in I}], s : \langle q, p, l_k(S_k)\rangle \cdot \mathtt{h} \to_L s[p] : \mathcal{E}[L_k], s : \mathtt{h} \quad k \in I \qquad [\![\mathtt{Rcv}]\!]$$

$$\Delta, \{s : \mathtt{h}\} \to_L \Delta \setminus \{s[p] : L\}, \{s : \mathit{remove}(\mathtt{h}, p) \cdot \langle\!\langle \psi, p\rangle\!\rangle\} \quad p \text{ non-robust} \qquad [\![\mathtt{Crash}]\!]$$

$$\frac{F' = \cup\{A \mid A \in \mathit{dom}(H) \wedge F \subset A \subseteq \mathit{Fset}(\mathtt{h}, p)\} \quad F' : L' \in H^{\downarrow}}{s[p] : \mathcal{E}[\big(L \blacktriangleright H^{\downarrow}\big)^{(\kappa, F)}.L''], s : \mathtt{h} \to_L s[p] : \mathcal{E}[\big(L' \blacktriangleright H^{\downarrow}\big)^{(\kappa, F')}.L''], s : \mathtt{h}} \qquad [\![\mathtt{TryHdl}]\!]$$

$$s[p] : \mathcal{E}[\big(\mathsf{end} \blacktriangleright H^{\downarrow}\big)^{\phi}.L],\ s : \mathtt{h} \to_L s[p] : \mathcal{E}[\big(\underline{\mathsf{end}} \blacktriangleright H^{\downarrow}\big)^{\phi}.L],\ s : \mathtt{h} \cdot \langle p, \psi\rangle^{\phi} \qquad [\![\mathtt{SndDone}]\!]$$

$$\frac{\langle \psi, p\rangle^{\phi} \in \mathtt{h}}{s[p] : \mathcal{E}[\big(\underline{\mathsf{end}} \blacktriangleright H^{\downarrow}\big)^{\phi}.L],\ s : \mathtt{h} \to_L s[p] : \mathcal{E}[L],\ s : \mathtt{h} \setminus \{\langle \psi, p\rangle^{\phi}\}} \qquad [\![\mathtt{RcvDone}]\!]$$

$$s[p] : \mathcal{E}[L],\ s : \langle q, p, l(S)\rangle \cdot \mathtt{h} \to_L s[p] : \mathcal{E}[L],\ s : \mathtt{h} \quad l \notin \mathit{labels}(\mathcal{E}[L]) \qquad [\![\mathtt{Cln}]\!]$$

$$\frac{\langle \psi, p\rangle^{\phi} \in \mathtt{h} \quad \phi \notin \mathcal{E}[L]}{s[p] : \mathcal{E}[L],\ s : \mathtt{h} \to_L s[p] : \mathcal{E}[L] \ \mid\ s : \mathtt{h} \setminus \{\langle \psi, p\rangle^{\phi}\}} \qquad [\![\mathtt{ClnDone}]\!]$$

$$\frac{\Delta \to_L \Delta'}{\Delta, \Delta_0 \to_L \Delta', \Delta_0} \qquad [\![\mathtt{Str}]\!]$$

$$\frac{\tilde{p} = \mathit{roles}(G) \setminus (F \cup \{p\}) \quad \mathtt{h}' = \mathtt{h} \cdot \langle\!\langle \tilde{p}, \{p\}\rangle\!\rangle}{G : (F, h_d) \vdash \Delta,\ s : \langle\!\langle \psi, \{p\}\rangle\!\rangle \cdot \mathtt{h} \to_L G : (F \cup \{p\}, h_d) \vdash \Delta,\ s : \mathtt{h}'} \qquad [\![\mathtt{F}]\!]$$

$$\frac{h'_d = h_d \cdot \langle p, \psi\rangle^{\phi}}{G : (F, h_d) \vdash \Delta,\ s : \langle p, \psi\rangle^{\phi} \cdot \mathtt{h} \to_L G : (F, h'_d) \vdash \Delta,\ s : \mathtt{h}} \qquad [\![\mathtt{CollectDone}]\!]$$

$$\frac{\mathit{roles}(h_d, \phi) \supseteq \mathit{roles}(G, \phi) \setminus F \quad \forall F' \in \mathit{hdl}(G, \phi).(F' \not\subseteq F) \quad h'_d = \mathit{remove}(h_d, \phi)}{G : (F, h_d) \vdash \Delta,\ s : \mathtt{h} \to_L G : (F, h'_d) \vdash \Delta,\ s : \mathtt{h} \cdot \langle \psi, \mathit{roles}(G, \phi) \setminus F\rangle^{\phi}} \qquad [\![\mathtt{IssueDone}]\!]$$

Figure 3.15.: Reduction relation over session environments.

When Ψ is not changed during reduction, i.e., $\Psi \vdash \Delta \rightarrow_L \Psi \vdash \Delta'$, we simply write the reduction relation as $\Delta \rightarrow_L \Delta'$. If $\Psi \vdash \Delta \rightarrow_L \Psi' \vdash \Delta'$ and $\Gamma = \Gamma_0, \Psi$ and $\Gamma' = \Gamma_0, \Psi'$, then we write $\Gamma \vdash \Delta \rightarrow_L \Gamma' \vdash \Delta'$. The reduction $\Psi \vdash \Delta \rightarrow_L \Psi' \vdash \Delta'$ closely follows the reduction of networks (see Figures 3.8 to 3.10).

The following property states that a coherent session environment will reduce to another coherent session environment.

Theorem 1 (Preservation of Coherence)**.** $\Gamma \vdash \Delta$ *coherent and* $\Gamma = G :_s (F, h_d), \Gamma'$ *and* $G :_s (F, h_d) \vdash \Delta \rightarrow_L G :_s (F', h'_d) \vdash \Delta'$ *imply that* $\Gamma', G :_s (F', h'_d) \vdash \Delta'$ *is coherent.*

We prove the statement by mechanically proving each case. We present proof details in Appendix A.2.2. The main idea is as follows. In every case, we start with a session environment that is coherent for s, i.e., all participants in s are dual to each other and s is guided by a coordinator, say $G :_s (F_q, \mathtt{h})$ – G is well-formed; after the reduction, we show that the new session environment is still coherent. We consider that the reduction may only use parts of the coherent session environment – the session environment was split by $[\![\mathtt{Str}]\!]$. We now discuss some proof details around failure handling.

Obsolete messages. The reductions $[\![\mathtt{Snd}]\!]$ and $[\![\mathtt{F}]\!]$ can result in message types that are not received but instead must be cleaned – well-formedness Condition 3 is required here. In $[\![\mathtt{Snd}]\!]$, say p sends to q, assume a try-handle, say (κ, F) encloses the selection type of p or the matching branching type of q; and the queue type contains failure notifications that trigger failure handling, e.g., for F', in the try-handle (κ, F). Then the effect of (see Definition 14 (Effect of)) changes the try-handle to (κ, F'), i.e., $[\![\mathtt{Snd}]\!]$ reduces a selection type enclosed by the try-handle (κ, F), for which effect on activates failure handling for F'. By coherence (duality), the types of p and q, *taking into account the queue type* (effect of), are dual to each other before the $[\![\mathtt{Snd}]\!]$ reduction, i.e., for duality both types have the try-handle (κ, F'), note the F'. After the reduction, we show that the sent message that under the try-handle (κ, F') has no matching branching type in q, cannot cause harm; well-formedness Condition 3 ensures that the sent label does not occur in q's type (see $labels(L)$), therefore effect of removes the sent message and coherence is preserved after reduction.

In $[\![\mathtt{F}]\!]$ the send failure notification types can activate failure handling (see Definition 14 (Effect of)), which potentially makes message types inside the queue type obsolete. Following the argumentation in $[\![\mathtt{Snd}]\!]$, those message types can be safely cleaned.

Consistent failure handling. The reduction $[\![\mathtt{F}]\!]$ adds failure notifications to the queue type that can activate (see Definition 14 (Effect of)) different failure handlings at different participants. The proof relies on two key facts to show that the session environment remains coherent. First, well-formedness Conditions 1 and 2 ensure that for any failure set, say F, and any pair of participants, say p and q, a unique outermost try-handle exists;

or alternatively there is no try-handle that is triggered by F. Secondly, $[\![\texttt{F}]\!]$ and $[\![\texttt{IssueDone}]\!]$ ensure that either all relevant participants or no participant have notification types in the queue. Together with Proposition 1, this ensures agreement on when to leave a try-handle and what failures to handle.

3.7.2. Subject reduction

Subject reduction states that: a network N that is well-typed under a coherent session environment remains well-typed under a coherent session environment after reduction. Similarly, a network N that is well-typed under an empty session environment remains well-typed under an empty session environment after reduction.

Theorem 2 (Subject Reduction)**.**

(a). $\Gamma \vdash N \rhd \Delta$ *with* $\Gamma \vdash \Delta$ *coherent and* $N \to N'$ *imply that* $\exists \Delta'$ *such that* $\Gamma' \vdash N' \rhd \Delta'$ *and* $\Gamma \vdash \Delta \to_L \Gamma' \vdash \Delta'$ *or* $\Delta \equiv \Delta'$ *and* $\Gamma' \vdash \Delta'$ *coherent.*

(b). $\Gamma \vdash N \rhd \emptyset$ *and* $N \to N'$ *imply that* $\Gamma \vdash N' \rhd \emptyset$*.*

The proof is by structural induction over the evaluation relation, relying on Theorem 1 for preservation of coherence. We present proof details in Appendix A.2.4. The proof is relatively straightforward; Theorem 1 does the heavy lifting. We show (a) first, then (b) follows from (a) immediately.

Subject reduction ensures communication safety, because subject reduction ensures that the coherence invariant is preserved. The coherence invariant ensures that types of a received messages match the expected message types, failures are handled, and participants perform consistent failure handling.

3.8. Implementation

Based on the calculus presented previously we developed a domain-specific language (DSL) and corresponding runtime system in Scala, using Apache ZooKeeper as the coordinator. In the following, we show that our introduced abstractions match our implementation. In particular, we show that Zookeeper together with our runtime system provide the coordinator abstraction and that our failure detector assumption (which in practice can be weakened by program-controlled crash) is reasonable.

Apache ZooKeeper as coordinator

Our coordinator is an abstraction for a coordinator like Apache ZooKeeper. Similar to global queues, which are an abstraction for pairwise channels like TCP. Before describing how ZooKeeper and our runtime implement the coordinator abstraction, we provide a high-level description of the important ZooKeeper functionalities that we use. In essence, ZooKeeper provides a hierarchical namespace similar to a classical file system which guarantees sequential consistency and atomicity. Clients that work with ZooKeeper enter into a ZooKeeper session (the runtime system ensures that during **(Link)** every linking process enters a ZooKeeper session). ZooKeeper sessions are essential for so-called *ephemeral nodes*, which allow clients to save non-persistent data for as long as their ZooKeeper session is active, in contrast to normal nodes which are persisted. ZooKeeper exchanges heartbeats with all clients and if ZooKeeper does not receive heartbeats from a client for a configuration timespan, it disconnects that client (i.e., ends the ZooKeeper session of that client and removes all its ephemeral nodes).

When participants create a new session via the **(Link)** rule, the runtime system creates a unique namespace for the new session and saves the following information in it: (i) a persistent queue (the notification queue) for all done and failure notifications sent to the coordinator, which ensures that all participants see notifications sent to that queue in the same order; (ii) for every participant a node which contains the information whether that participant has terminated (i.e., successfully finished its involvement in the session); (iii) an ephemeral node for every participant that indicates whether the participant has an active ZooKeeper session.

Done and failure notifications

ZooKeeper, by itself, does not directly implement the reduction rules **(CollectDone)**,**(IssueDone)**, and **(F)**. However, ZooKeeper ensures that every participant sees the done and failure notifications in the same order. That makes it straightforward for the runtime system to provide the coordinator abstraction by applying these rules in order of the notifications. The rules **(CollectDone)**,**(IssueDone)** and **(F)** provide a deterministic outcome (**(IssueDone)** processes done notifications based on the order in which they arrive). In cases in which more than one of these rules is applicable, the runtime system selects a rule based on the following order: **(CollectDone)** over **(IssueDone)** over **(F)**.

Failure detection and false suspicions

The operational semantics model a perfect asynchronous failure detector (cf. **(Crash)**). In practice, false suspicions occur (i.e., non-perfect failure detection). The failure detection in our implementation works as follows: if the runtime system of a participant p suspects that participant q has failed, it only issues a failure notification if participant q has no active ZooKeeper session (the ephemeral node of q does not exist) and the saved status of q in ZooKeeper is not terminated. The key point here is that a failure notification for a participant q will only be issued if q lost its ZooKeeper session.

A participant q can lose its connection with ZooKeeper without failing, e.g., the network drops too many heartbeat messages. After q loses its ZooKeeper session, other participants can issue failure notification for q; therefore, q assumes that it is suspected and stops, i.e., it performs a controlled crash. As mentioned, this occurs very infrequently in practice but is a feature used in many distributed systems as a last resort.

When a participant finishes its actions inside a session, it sets its status to terminated before disconnecting from ZooKeeper. This ensures that other participants cannot detect it as failed.

Automatically inferring try-handle levels and semi-unique labels

In order to simplify the formalism, we require that every try-handle in a well-formed global type is annotated with a unique level. For the formal development we believe this is a reasonable assumption, but our runtime system provides functionalities to remove this burden from the programmer. Concretely, we implemented a deterministic traversal for global types which assigns unique and deterministic levels to all try-handles in a given global type and we allow writing a global type which only contains labels required for branching, adding others automatically.

Type checking

The prototype performs dynamic type checking in a way similar to Chen et al. [CBD+11]. The session channel, which provides the session communication primitives, contains a session monitor that monitors and controls session action performed on the channel. When an endpoint program performs a session action, the channel first verifies that this action is permitted. If that is the case, it performs that action. Otherwise, it raises an exception, and consequently, the endpoint fails. Such a failure then triggers failure handling the same way as a process crashes. Note, our formalism performs static typing, and we could,

e.g., adopt the code generation approach of Zhou et al. [ZFH+20] to provide static type checking in our prototype.

3.8.1. Evaluation

We evaluated our prototype on the LR model, explained in Section 3.1.4. We compare the performance of a session typed LR model implemented in our prototype, which we call SessionLR, with failure agnostic baselines, which use plain TCP sockets (i.e., not the session runtime). We call the different baselines Baseline-xW (x indicates the number of workers the baseline uses).

The SessionLR follows the global type from Figure 3.2, which we slightly extended to, e.g., accommodate more workers. It implements the following behavior: The driver process divides the data points evenly between the workers, provides the current model parameter to the workers, collects the partial model updates, and updates the model parameter in every iteration. The workers calculate the partial model update over the assigned data points using the current model parameter and send it to the driver. SessionLR consists of one driver process and three worker processes. The communication pattern and behavior of the baseline Baseline-xW is similar to that of the SessionLR implementation, except that it has no logic for handling failures, uses x workers, and is implemented over plain TCP sockets.

We train the LR model over 30 training iterations on one million data points, with 10 dimensions and a binary label. The data points are drawn from a normal distribution with a standard deviation of 1.0 and a mean of -0.7 and 0.7 for the two binary classes. All servers have local access to the data points. The evaluation is repeated 10 times, and we provide the average and standard derivation for the 10 runs. The different processes run on dedicated servers with the following specification: Ubuntu 18.04.3 LTS, Intel(R) Xeon(R)E -2278G CPU @ 3.40GHz and 64 GB of RAM.

Figure 3.16 shows the runtime characteristic of the SessionLR and the three failure agnostic baselines. The x-axis shows the 30 training iterations, and the y-axis shows the average time (and standard derivation out of 10 runs) taken to finish each iteration. It shows the following evaluation scenario: SessionLR starts with three workers, and over the 30 training iterations, we induce two worker failures. The three baseline implementations (Baseline-3W, Baseline-2W and Baseline-1W) use a fixed number of worker processes ($1-3$ workers) for all iterations. We induce failures in the following way: In iteration 11 and iteration 21 we let one worker fail, i.e., SessionLR uses three workers for iteration 1 - 10; starts iteration 11 with 3 workers but due to a failure finishes with 2 workers; uses 2 workers for iterations $12-20$; starts iteration 21 with two workers but due to a failure finishes with one worker; and continues until iteration 30 with one worker.

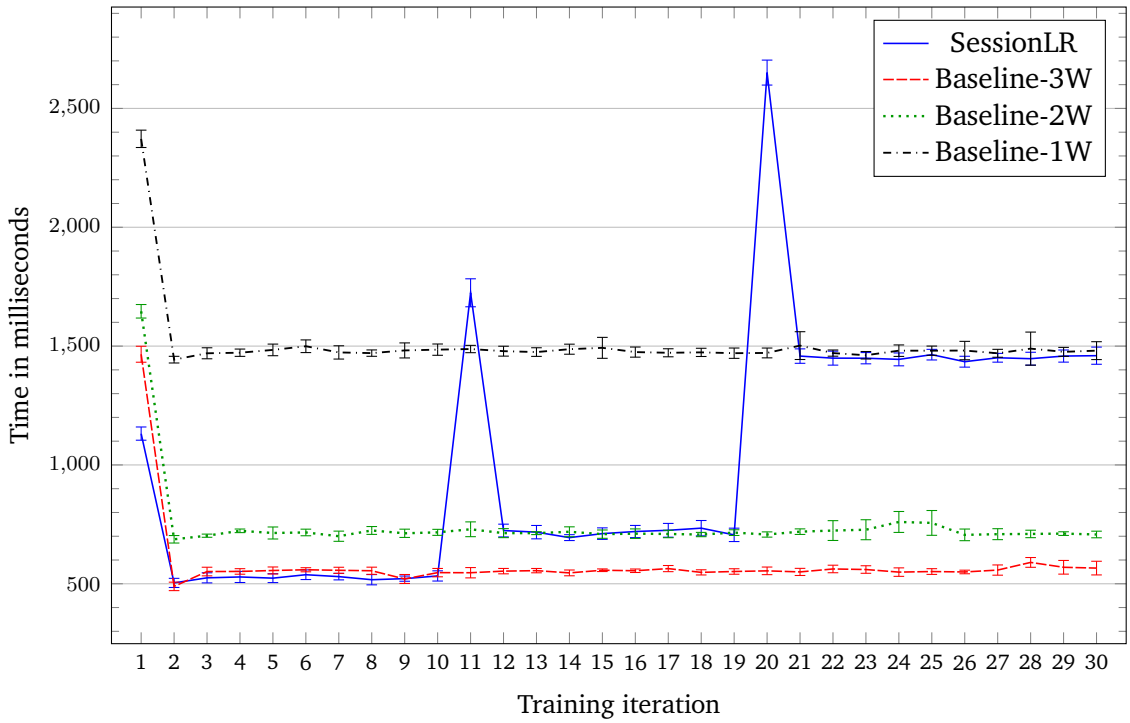

Figure 3.16.: Comparison of the SessionLR (a session typed version of an LR model) with three failure agnostics baselines (Baseline-xW) over 30 training iterations. SessionLR starts with three workers and we induce a worker failure in iteration 11 and in iteration 21. The three baselines execute the LR model with 3 workers (Baseline-3W), 2 workers (Baseline-2W) or 1 worker (Baseline-1W) – without any failure.
The spike in iteration 1 is due to the initial reading of the data points. The spikes at iterations 11 and 21 for SessionLR include, the failure detection, activation of failure handling, and a partial and complete model update calculation.

The first iteration in Figure 3.16 takes longer for all implementations because the processes load the data points. All four implementations perform the training iterations at a relatively constant speed that depends on the number of workers. In the iterations without a failure, SessionLR has a processing speed similar to the baseline that uses the same number of workers. In other words, SessionLR performance is initially comparable with Baseline-3W, which uses three workers. After the first failure, its performance is comparable with Baseline-2W, which uses 2 workers. And after the second failure it is comparable with Baseline-1W, which uses 1 worker.

The two peaks of SessionLR (iteration 11 and 21) contain the time for a partial model update calculation, the failure detection, the coordination to activate failure handling, and a recalculation of the model parameter for the new data point assignment.

3.9. Related Work

Several session type works study exception handling [CHY08, CYH09b, DHH$^+$15, KY14]. However, to the best of our knowledge, this is the first theoretical work to develop a formalism and typing discipline for the coordinator-based model of *crash failure* handling in practical asynchronous distributed systems.

Structured interactional exceptions [CHY08] study exception handling for binary sessions. The work extends session types with a *try-catch* construct and a *throw* instruction, allowing participants to raise runtime exceptions. Global escape [CGY16] extends previous works on exception handling in binary session types to MPSTs. It supports nesting and sequencing of try-catch blocks with restrictions. Reduction rules for exception handling are of the form $\Sigma \vdash P \rightarrow \Sigma' \vdash P'$, where Σ is the *exception environment*. This central environment at the core of the semantics is updated synchronously and atomically. Furthermore, the reduction of a try-catch block to its continuation is done in a synchronous reduction step involving all participants in a block. Lastly, this work can only handle exceptions, i.e., explicitly raised application-level failures. These do not affect communication channels [CGY16], unlike participant crashes.

Similarly, our previous work [CVB$^+$16] only deals with exceptions. An interaction $p \rightarrow q : S \vee F$ defines that p can send a message of type S to q. If F is not empty then instead of sending a message p can throw F. If a failure is thrown only participants that have casual dependencies on that failure are involved in the failure handling. No concurrent failures are allowed, so all interactions which can raise failures are executed in a lock step fashion. Therefore, the model cannot be used to deal with crash failures.

Adameit et al. [APN17] propose session types for link failures, which extend session types with an optional block which surrounds a process and contains default values. The

default values are used if a link failure occurs. In contrast to our work, the communication model is overall synchronous whereas our model is asynchronous; the optional block returns default values in case of a failure (providing a form of failure masking) but it is still the task of the developer to do something useful with it.

Demangeon et al. study interrupts in MPSTs [DHH+15]. This work introduces an interruptible block $\{|G|\}^c\langle l \text{ by } \mathtt{r}\rangle; G'$ identified by c; here the protocol G can be interrupted by a message l from $\mathtt{r}$ and is continued by G' after either a normal or an interrupted completion of G. Interrupts are more a control flow instruction like exceptions rather than an actual failure handling construct, and the semantics cannot model participant crashes. Neykova and Yoshida [NY17] show that MPSTs can be used to calculate safe global states for a safe recovery in Erlang's *let it crash* model [Arm03]. That work is well suited for recovery of lightweight processes in an actor setting. However, while it allows for elaborate failure handling by connecting (endpoint) processes with runtime monitors, the model does not address the fault tolerance of runtime monitors themselves. As monitors can be interacting in complex manners, replication does not seem straightforwardly applicable, at least not without potentially hampering performance (just as with *straightforward* replication of entire applications).

Failure handling is studied in several process calculi and communication-centered programming languages without typing discipline. The conversation calculus [VCS08] models exception behavior in abstract service-based systems with message-passing based communication. The work does not use channel types but studies the behavioral theory of bisimilarity. Error recovery is also studied in a concurrent object setting [XRR+95]; interacting objects are grouped into coordinated atomic actions (CAs) which enable safe error recovery. CAs cannot be nested, however. PSYNC [DHZ16] is a domain specific language based on the *heard-of* model of distributed computing [CBS09]. Programs written in PSYNC are structured into rounds which are executed in a lock step manner. PSYNC comes with a state-based verification engine which enables checking of safety and liveness properties; for that programmers have to define non-trivial inductive invariants and ranking functions. In contrast to the coordinator model, the heard-of model is not widely deployed in practice. Verdi [WWP+15] is a framework for implementing and verifying distributed systems in Coq. It provides the possibility of verifying the system against different network models. Verdi enables the verification of properties in an idealized fault model and then transfers the guarantees to more realistic fault models by applying transformation functions. Verdi supports safety properties but not liveness properties.

3.10. Final Remarks

This chapter introduces a formal model of verified crash failure handling that features a lightweight coordinator, as is common in many real-life systems. The model carefully exposes potential problems that may arise in distributed applications due to partial failures, such as inconsistent endpoint behaviors and orphan messages. Our typing discipline addresses these challenges by building on the mechanisms of MPSTs, e.g., well-formedness of global types for sound failure handling specifications, modeling of asynchronous permutations between regular messages and failure notifications in sessions, and the type-directed mechanisms for determining correct and orphaned messages in the event of failures. We adapt coherence of session typing environments (i.e., endpoint consistency) to consider failed roles and orphan messages, and show that our type system statically ensures subject reduction in the presence of failures. Furthermore, this chapter introduces a prototype based on our formal model. The prototype is implemented in Scala and uses Apache ZooKeeper as a coordination service. Our evaluation shows that a session-type LR model has a runtime performance comparable to failure agnostic baselines in non-failure cases.

Interesting future research would include studying of our model in a wider range of applications. We believe dynamic role participation and role parameterization would be valuable for failure handling, especially in crash-recovery scenarios. Another interesting direction for research would be the development of an option to use the coordinator as part of the protocol so that the coordinator can persist pertinent runtime information.

4. A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming

This chapter presents our article "A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming", which appeared at Proceedings of the ACM on Programming Languages 2021 [VHEZ21]. The chapter presents for most parts that work *directly*, i.e., the chapter is for most parts a verbatim copy of our article.

Correctly designing and implementing distributed systems software is notoriously difficult. Multiparty session types (MPSTs) is a typing discipline for concurrent processes that statically ensures properties such as freedom from message reception errors and deadlocks. The existing approaches in MPSTs cannot, however, be applied to a significant class of real-world distributed systems because they do not support practical specification and verification of protocols that handle and recover from *partial failures*.

This chapter presents the first formulation of MPSTs for practical *fault-tolerant distributed programming*. We tackle the long-standing challenges faced by session types in this context: bringing structure to communication patterns involving asynchronous and concurrent partial failures, and integrating the range of features required to express fault-tolerant protocols in practice, that involve dynamic replacement of failed parties and retrying failed protocol segments in the presence of imperfect failure detection. Key to our approach is that we develop the first model of *event-driven concurrency* for multiparty sessions, to unify the session-typed handling of failures and regular I/O events. Moreover, the characteristics of our model allow us to prove a *global progress* property for well-typed processes engaged in multiple concurrent sessions, which does not hold in traditional MPST systems. To demonstrate its practicality, we implement our approach as a toolchain for Scala, and use it to specify and implement a session-typed version of the cluster manager (CM) system of

the widely employed Apache Spark data analytics engine. Our session-typed CM integrates with the other vanilla Spark components to give a functioning Spark runtime; e.g., it can execute existing third-party Spark applications.

4.1. Introduction

Correctly designing and implementing distributed systems is notoriously difficult. Even an idealized setting requires reasoning about independent program components concurrently executing on a set of physically remote processes, separated by a network that offers communication only via asynchronous message passing. More realistically, the distributed system will also be subject to *partial failures* where some processes may fail while others continue operating.

Multiparty session types (MPSTs) [CHY08, HYC16] is an active area of research on typing disciplines for concurrent processes; see Ancona et al. [A+16]; Gay and Ravara [GR17]; Hüttel et al. [HLV+16] for surveys of session types and the broader area of behavioral types. The idea is to apply type systems to message passing programs to statically ensure properties such as absence of message reception errors and deadlocks. MPSTs is a promising technique for distributed systems: a key notion in MPST theory is *projection*, which derives a decoupled (i.e., distributed) view of a protocol from the perspective of each participant. Existing approaches in MPSTs are, however, not applicable to a significant class of real-world distributed systems, as they do not support the specification and verification of practical protocols dealing with partial failures.

Reasoning about failures is a long-standing challenge for MPSTs and related techniques. First, the chaotic nature of communication patterns involving partial failures among concurrent, asynchronously interacting participants is at odds with the traditional ethos of session types, which seeks to ensure safety by strictly regulating all potential interactions. Second, protocols for *fault-tolerant* distributed systems often require a *combination* of relatively advanced features for handling failures, e.g., dynamically replacing failed participants and retrying failed protocol segments in an application, often in the presence of *imperfect* failure detection. These challenges must be overcome for techniques like session types to be applicable to practical distributed systems.

In the existing literature on MPSTs, there are two main works on the topic of failure handling [APN17, VCE+18]. We elaborate on related work in Section 4.8, but to summarize, both are theoretical works and limited in practical applicability. The system of Adameit et al. [APN17] is essentially based on a *synchronous* communication model. Our previous work [VCE+18], which is also part of this thesis (see Chapter 3), assumes in its formalism *perfect* failure detection (unimplementable in an asynchronous system) and

proposes a new programming model based on an asynchronous variant of `try-catch` that is remote from existing practices. Crucially, neither support the range of features required to express dynamic participant replacement and retrying in an on-going fault-tolerant application.

This chapter develops the first MPST-based theory for *practical* programming of *fault-tolerant* distributed applications. Our aim is to support MPST-based specification and verification of multiparty asynchronous protocols that deal with *crash-stop* process failures, to ensure the key properties of communication safety and progress. We present our approach through the use case of a substantial system from an industry-strength data processing framework – the distributed cluster management (CM) system of Apache Spark[1] [ZCD+12b].

Our theory draws inspiration from one of the major paradigms for distributed programming in practice, *event-driven programming* (EDP), as employed, for instance, in Spark's runtime. Event-driven concurrency allows for highly concurrent and asynchronous programming commensurate with the nature of the network, and the event abstraction captures the notion of failure as a particular kind of event. We exploit these characteristics of EDP to develop the first model of *event-driven concurrency for multiparty sessions*. It enables us to integrate the range of necessary practical features by *unifying* the handling of all the "regular" session I/O and failure events under a uniform session-typeable programming interface. Moreover, it enables us to prove a *global progress* property for well-typed processes engaged in multiple concurrent sessions, which does not hold in traditional MPST systems.

To demonstrate the practicality of our approach, we implement our system as a toolchain for fault-tolerant distributed programming in Scala and use it to specify and implement a session-typed version of Spark's CM. Our session-typed CM is compatible with other vanilla Spark components and provides a functioning Spark runtime to execute existing third-party Spark applications *without* any code modification. This is enabled by the design of our theory, in (1) targeting EDP as a practical, established programming model for asynchronous I/O, (2) supporting the combination of features needed for practical fault-tolerant protocols, and (3) appropriately abstracting failure detection mechanisms, including catering for unreliable failure detection (i.e., *false suspicions* of remote process failures). We use a Spark implementation of the industry-standard TPC-H benchmark suite[2] to test and evaluate the runtime performance of our session-typed CM. Measuring the time it takes from submitting a Spark application to its completion, the results show our prototype implementation incurs an average overhead below 10%.

[1]Apache Spark. `http://spark.apache.org`.
[2]TPC-H benchmark. `http://www.tpc.org/tpch/`

The roadmap and the concrete contributions of this chapter are as follows. We present

- the first MPST-based framework to support the combination of novel features needed to specify practical fault-tolerant protocols. Section 4.2 illustrates these features, including *failure-aware subsessions*, *role parameterization*, and *dynamic role assignment*, using the Spark CM use case.
- the first model of *event-driven concurrency* for multiparty sessions. Section 4.3 demonstrates endpoint projection and implementation of distributed, event-driven processes using our Scala toolchain.
- a formalization of our framework in Section 4.4 and Section 4.5, and proofs of key properties of *communication safety* and *global progress* for an entire well-typed system of *multiple, concurrent subsessions* in Section 4.6. Safely integrating the range of features needed to meet our aims in a tractable formalism is a significant challenge. Global progress does normally not hold in MPST systems.
- a practical evaluation of our framework. We use our Scala toolchain to specify and implement a selection of examples from MPSTs literature in addition to the Spark CM use case, and present the runtime performance of using the latter to execute the TPC-H benchmark in Section 4.7.

Section 4.8 discusses related work. Section 4.9 concludes with future work. The Appendices provide additional examples, auxiliary definitions and proof details (see Appendix B.1 for an outline of the appendix contents).

4.2. MPSTs for Fault-tolerant Sessions

We now describe the Apache Spark use case and using that use case, we illustrate our new features, including failure-aware subsessions, role parameterization, and dynamic role assignment.

4.2.1. Running example: Apache Spark's standalone cluster manager

Apache Spark [ZCD+12b] is a big data analytics framework for distributed clusters. Spark supports several types of *cluster managers* – e.g., Spark's standalone cluster manager (Spark-CM), Mesos [HKZ+11] or YARN [VMD+13b] – for managing the distributed resources in running Spark applications[3]. We use the Spark-CM as a concrete use case (Section 4.7)

[3] Apache Spark cluster mode overview. `https://spark.apache.org/docs/latest/cluster-overview.html`

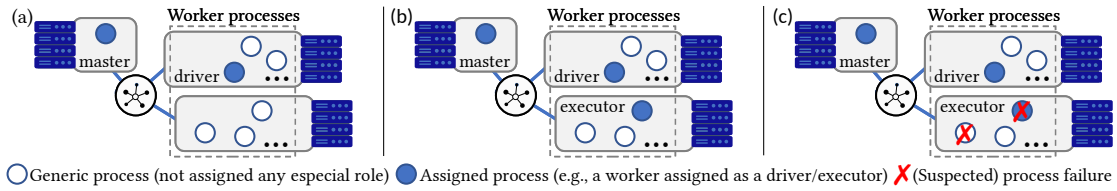

Figure 4.1.: Stages in an Apache Spark cluster management scenario: (a) application driver assigned; (b) the driver and one executor assigned; (c) original executor process failed and a replacement executor assigned.

and a running example.

Figure 4.1 depicts the execution of a Spark application as managed by the Spark-CM, including recovery from a process failure. An application involves the *manager* (called *master* in Spark's source code) and a pool of distributed *worker* processes. The Spark-CM launches a new application by (a) first *assigning* a worker process as the application *driver*, and then (b) assigning multiple workers as *executors* (we depict just one for simplicity). The driver hosts the main user program while the executors perform their allocated computations. If a driver or executor process fails, as in (c), the Spark-CM preserves the running application by assigning another worker to replace the failed one. The cause of a process failure may be the failure of its host machine (the latter can be represented by the former).

The application as a whole thus comprises a set of concurrent, collaborative subtasks that these distributed processes carry out via asynchronous message passing over the network. E.g., in (b) the master and driver may continue the task of spawning further executors, while concurrently engaging with the existing executor in a separate computation subtask. This use case demonstrates a general class of distributed systems and the combination of features they depend on, which existing MPSTs do not support. Systems such as the Spark-CM depend on:

- dynamically initiating new *concurrent tasks* between subsets of the existing participants (e.g., allocating and performing work across different workers in parallel).
- *dynamically* assigning the participants for the above (e.g., when assigning executors);
- *handling of* and *recovery from* participant failures (e.g., replacing a failed executor);
- accounting for the fact that, in general, failure detection is asynchronous and potentially *unreliable*, i.e., a participant may be *falsely suspected* of failure even as it continues operating.

4.2.2. Limitations of standard MPSTs

The standard workflow of MPSTs [HYC16, CDYP16] starts from the user specification of a *global type*, describing all possible interactions between participants from a global perspective. Consider this global type (written in a practical notation for later comparisons):

```
RunEx(roles master, driver, exec) {
    ... master → driver: InitDr. driver → exec: InitEx ...
}
```

This protocol, named `RunEx`, is between three participants, abstracted as *roles* `master` (the manager), `driver`, and `exec`. The above fragment specifies that `master` sends an `InitDr` message to `driver`, which in turn then sends an `InitEx` to `exec`.

We can use this minimal example to clarify some limitations of the *session* abstraction developed in standard MPSTs. First, a session involves a *fixed* set of participants, one participant playing each role, that are all fixed when the session is first *initiated*: there is no notion of dynamically (i.e., *during* an ongoing session) assigning a participant to a role, nor losing or replacing assigned participants. Second, there is no notion of a *failure event* nor how the protocol should proceed in such an event – standard MPSTs support protocol branching (choice) and recursion, but their strictly regulated nature (to conservatively ensure safety) cannot capture the asynchronous nature of failures. Standard MPSTs thus cannot express protocols involving failure handling and recovery, nor verify processes that engage in such behaviors.

4.2.3. MPSTs for fault-tolerant sessions (initial overview)

This chapter develops a generalized notion of session to address the limitations of standard MPSTs and better support real-world distributed systems such as the Spark-CM. In our setting, a session is generalized as a system of *failure-aware, concurrent subsessions*. We first explain individual subsessions, then inter-subsession concurrency.

Role sets. We introduce an abstraction of session participants called *role sets*. In standard MPSTs, a role represents a participant process (participant, for short) with a certain behavior. By contrast, a role set in our model (written with an initial uppercase letter, e.g, `Masters` or `Workers`) represents a non-empty *set* of participant processes that are *capable* of the same behavior. In this setting, a *role* (initial lowercase letter, e.g., a master `m`, a driver worker `wD`, or an executor worker `wEm`, `wD`, `wE`) represents a single participant *dynamically assigned* from its role set (as explained shortly). In addition to the standard role-to-role message passing, e.g., `master → driver`, a role can *multisend* a message to a whole role set, e.g., `master → Workers`.

Role sets provide a practical, lightweight form of participant parametricity. Our framework abstracts from the run-time size of a role set (i.e., the number of participants) and how processes are spawned; it assumes only that a *sufficient* number of participants are available at run-time.

Subsessions. The following *subprotocol* illustrates our general notion of *failure-aware subsession*:

```
// A (sub)protocol
RunEx(roles m: Masters, wD: Workers; assign wE: Workers; rosets Workers) {
// The "normal activity"
  ... m → wE: InitEx ...

// wE is monitored by m for potential failure
with wE@m.
// The "failure handling activity", if wE is _suspected_ by m
  ... m → Workers: FailEx ...
}
```

It specifies a communication (sub)session whose participants are the individual processes playing roles `m`, `wD` and `wE`, combined with all member processes of the role set `Workers`. Each role is annotated by its role set (e.g., `wD: Workers`). The `assign` declaration specifies this participant (i.e., playing `wE`) should be *dynamically* assigned from its role set when this subsession is initiated; we say it is *assigned* to this role. `roles` specifies participants *already* assigned prior to this subsession, and `rosets` specifies role sets. Our framework is agnostic of how the `assign` mechanism is implemented, provided that the assigned participant is not also playing one of the `roles` in *this* subsession.

The main subprotocol definition has two parts: the *normal activity* written before the `with`, and the *failure handling activity* after. The `wE@m` clause specifies that, in this subsession, `m` is responsible for *monitoring* `wE` for failure. (In general, any one of `roles` may be the monitor.) By default, the subsession proceeds following the interactions in the normal activity. However, if `m` *suspects* `wE` of failure, `m` will switch to the failure handling; it may do so at any point during the subsession. Global type well-formedness requires all other participants (i.e., except `wE`) to be notified of `wE`'s failure via *explicit* interactions in the failure handling; these participants also switch to the failure handling upon receiving such a notification (i.e, `FailEx`), and (barring other failures) the remaining participants in this subsession will proceed following the failure handling.

To simplify the presentation of our theory, we restrict a subsession to one `assign`; multiple assignments can be expressed via nested subsessions (see below) or supported as a syntactic sugar.

Our framework is agnostic of any particular failure detection mechanism. Our theory simply models failure *suspicions* asynchronously and non-deterministically – this is a key

design choice for broader practical applicability, including settings with unreliable failure detection.

Concurrent subsessions. In our framework, a *session* is a system of one or more *concurrent subsessions*. Any one participant process may be involved in any number of concurrent subsessions, playing an individual role and/or as a generic role set member in each.

A session starts from a *root* (or *entry*) subsession, i.e., an instance of the root subprotocol, which collects together every participant that may be involved at some point in the overall session.

```
// Root subprotocol
root RunDr(roles m: Masters; assign wD: Workers; rosets Workers) {
    ... with wD@m ...
}
```

We refer to the `roles` in the root subsession as *robust* roles. They are the only roles that are not explicitly monitored for failure *within* the session: we can assume their failure is handled by some external mechanism, or simply causes the session as a whole to fail.

More generally, a subsession is a *child* subsession *spawned* by an initiation interaction between the relevant subset of participants in its *parent* subsession. E.g., from within the root `RunDr`, we can spawn a new subsession instance of `RunEx` (from above) by:

```
// from inside RunDr; RunEx is as above
... spawn RunEx(m, wD; Workers; Workers) ...
```

The arguments correspond to the parameter declarations of `RunEx`. The `roles` arguments may be any *existing* role from the specified role set, and the `assign` argument is the role set from which to assign a participant. The spawned subsession runs *concurrently* with its parent (i.e., with the parent's interactions following `spawn`), meaning that any processes participating in *both* do so concurrently.

A running session thus forms a tree of parent-child subsessions, which we refer to as the *monitoring tree*. Excluding the robust role, every individual role that occurs in some subsession has a monitor, in that subsession or an ancestor (i.e., the subsession in which the role was first assigned). In other words, every case of (non-robust) participant failure is safely covered within the session as a whole. This gives some intuition of how our framework incorporates subsessions, dynamic role assignment, and failure handling while retaining the desired MPST safety and progress properties (Sections 4.3.3 and 4.6.2). We explain this further using the Spark-CM example below.

Note, `spawn` actions may occur within both normal and failure handling activities. The latter are crucial to express fault-tolerant protocols that *replace* failed participants and *retry* failed subsessions.

```
1  root RunDr(roles m: Masters; assign wD: Workers; rosets Workers) {
2    m → wD: InitDr(d: DrInfo).
3    wD → m: Ack(appID: Int).
4    mu t. m → Workers {
5      AddEx(): spawn RunEx(m, wD; Workers; Workers). t,
6      Ok(): end }
7  with wD@m.  // Replace driver and retry
8    m → Workers: FailDr(appId: Int).
9    spawn RunDr(m; Workers; Workers). end }
10
11 RunEx(roles m: Masters, wD: Workers;
12       assign wE: Workers;
13       rosets Workers) {
14   m → wE: InitEx(e: ExInfo).
15   wE → m: ExDone(appId: Int, exId: Int).
16   m → wD: ExFinished(appId: Int, exId: Int).
17   end
18 with wE@m.  // Replace executor and retry
19   m → Workers: FailEx(appId: Int, exId: Int).
20   spawn RunEx(m, wD; Workers; Workers). end
21 }
22
```

Figure 4.2.: Global type for the Session-CM: a session-typed Spark-CM. `RunDr` is the root subprotocol.

4.2.4. Global type for the component Interactions in the Spark-CM

We can now express the message passing behavior of fault-tolerant protocols as MPST global types. Figure 4.2 specifies a global type for a simplified but representative version of the Spark-CM, which we refer to as the **Session**-CM; our full version (Appendix B.7) is compatible with the actual Spark framework and supports existing third-party Spark applications. The Session-CM has two subprotocols, `RunDr` and `RunEx`, for handling the lifecycle of a driver and an executor in a Spark application, respectively.

The root subprotocol, `RunDr`, involves these participants: a (robust) manager role `m`, a driver role `wD` assigned from role set `Workers`, and the role set `Workers` as a whole. To launch a Spark application, `RunDr` dynamically assigns a driver process (as part of session initiation) and initializes it (lines 2–3). `RunDr` then spawns a number of concurrent child subsessions that assign executors to carry out the subtasks of the application in parallel (lines 4–6).

Line 2 is the standard role-to-role message passing of MPSTs [HYC16, CDYP16]: role `m` sends an `InitDr` message (carrying a `DrInfo` payload) to role `wD` to initialize the driver;

wD then replies to m with an acknowledgment on Line 3. As standard in MPSTs, and crucial for real-world applications, messaging in our framework is *asynchronous*, meaning output actions do not block (but input actions do block until a message is received). Lines 4–6 specify a standard *branch type* `m → Workers { AddEx(): ..., Ok(): ... }`: role m makes an internal choice to, in this instance, *multisend* either AddEx or Ok messages to the whole of Workers, who will follow the received message case as an external choice. Note that wD belongs to role set Workers, so the participant playing wD receives the branch message this way.

On line 4, `mu t ...` is a standard *recursive type*: the recursion is enacted by the t in the AddEx case (causing the subsession to "loop back" to line 4), while Ok leads to subsession termination end. As part of the recursive branch case, line 5 may be repeated to spawn multiple RunEx subsessions, all running concurrently with the RunDr subsession. Each child subsession involves: m and wD from RunDr as roles, an assigned Workers participant to play the new role wE, and the role set Workers. Our spawn is novel to MPSTs in supporting (i) unrestricted spawning of child tasks, where participants may be involved in *multiple* such tasks, that are (ii) fully concurrent with the parent task. Notably, our theory ensures *global progress* for the entire system of all subsessions: prior work on nested protocols [DH12] in MPSTs ensures progress only for a single session.

Lastly, lines 7–9 specify the failure handling behavior for the RunDr subsession. If the monitor m *suspects* the participant playing wD has failed (by whatever failure detector it employs), it switches from the normal to the failure handling activity in this subsession *and* stops its activity in all descendent subsessions. Our framework requires the monitor to explicitly notify all (potentially) remaining participants as part of the failure handling: m does so by a multisend to Workers (this implicitly excludes the suspected participant), who switch to failure handling and stop any descendent activities upon receiving the notification. A *new* RunDr subsession is then spawned between m and the remaining Workers, with some *other* participant assigned to wD (i.e., a replacement).

The structure of RunEx is similar. It dynamically assigns and initializes an executor to carry out some of the actual application work. By default, our theory permits the same participant to play wE in multiple RunEx subsessions. Note, a failure of wD during RunEx is handled by the parent RunDr subsession (as mentioned, remaining participants stop activity in all descendent subsessions when failure handling is activated); to reiterate, all roles (except robust roles) are *necessarily* monitored in the subsession where they were first assigned.

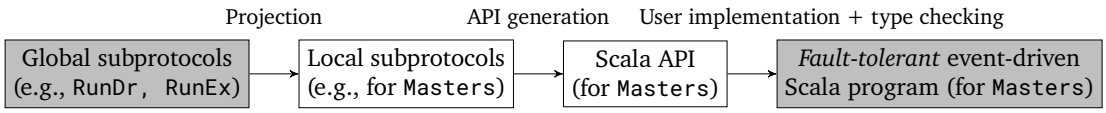

Figure 4.3.: Scala toolchain for MPST-based fault-tolerant distributed programming. Gray is written by the user.

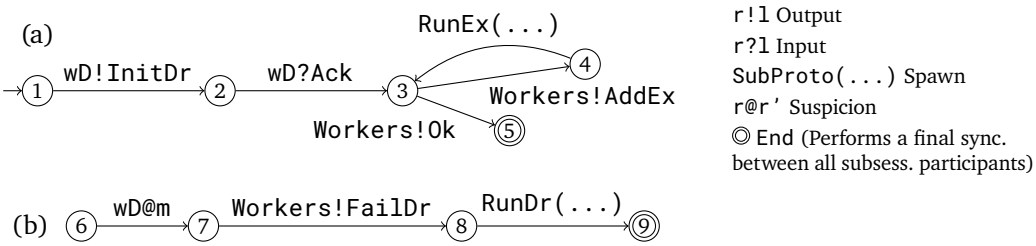

Figure 4.4.: CFSM pair for the (a) normal and (b) failure handling activities, respectively, of role `m` in `RunDr`.

4.3. Event-driven Programming for Fault-tolerant Sessions

We now illustrate endpoint projection and distributed, event-driven processes in our toolchain.

4.3.1. Scala toolchain overview

Figure 4.3 shows the main stages and artifacts in our MPST-based toolchain for fault-tolerant distributed programming in Scala. It is designed to support concurrent subsessions, failure handling, and *event-driven programming* (EDP) following a standard MPST top-down workflow. It starts from a user specification of the fault-tolerant application protocol as a global type, i.e., a set of interdependent subprotocols. We continue the Session-CM example (Figure 4.2) to illustrate the subsequent stages.

Local types and communicating finite state machines (CFSMs). A global type is *projected* to a *local type* describing the view of the protocol from each *kind* of participant. In our setting, each participant kind (e.g., a master) must support its behavior as a *generic* role set member (i.e., `Masters`) *and* as all its (potential) roles (i.e., `m`) – note, in this instance, `Masters` has *no* explicit generic behavior (unlike `Workers`). Section 4.4.2 presents projection in detail; here, we first illustrate the concept by showing the CFSM representation for local types used internally by our toolchain implementation.

In a nutshell, our toolchain represents each *localised* subprotocol as a pair of CFSMs, for the normal and failure handling activities; e.g., Figure 4.4 depicts the CFSM pair for role `m` in `RunDr`. Each state represents a point in the subprotocol where this participant performs one of the localized interaction types: *select* (output choice, subsuming the basic send), *branch* (input choice, subsuming receive), failure *suspicion* (by a monitor), *spawn* (subsession initiation), or *end* (subsession termination). By the design of our framework, spawn and suspicion states have exactly one outgoing transition, and end is a terminal state. Each transition represents an I/O action from that state to the next. Output `!` is a *non*-blocking action. The potentially blocking actions are input `?` and spawn (denoted by the subprotocol name). Following session initiation in standard MPSTs, spawn is a synchronization between the relevant participants; we also treat subsession termination as a similar synchronization.

A participant behavior starts from the initial state of its normal activity CFSM. The suspicion action `@` may switch the *monitor* (e.g., `m`) to its failure handling CFSM from *any* state of the normal activity, including end (i.e., state 5) if the subsession is not yet terminated. The other subsession participants (e.g., `wD`) switch to their failure handling CFSMs on receiving the initial message (i.e., a failure notification) in those CFSMs; the one exception is the *monitored* participant, whose failure handling is empty. See Appendix B.2 for all local types and CFSMs for Session-CM; e.g., `wD/wE/W` in `RunEx`.

Figure 4.4 thus represents the behavior of `m`. In the normal activity (a), `m` first performs a send to, then a receive from, `wD`. It then selects (by some internal decision) between the spawn loop (i.e., sending `Ack` to `Workers` and spawning `RunEx`) or the terminal case. In the failure handling (b), `m` multisends the failure notification, then joins the spawning of the replacement `RunDr` subsession.

Scala API generation. For practical programming, we implement our theory as a toolchain in Scala. Our toolchain uses the CFSM representation of each local type to generate a *type-directed API [HY16] for MPST-based EDP*. The basic idea is: for each state, it generates a state-specific *channel type*; and for each transition, it generates a transition-specific I/O method on the associated channel type. Each I/O method takes the arguments, if any, required by the action (e.g., a message), and returns the channel type for its *successor* state in the CFSM; input methods additionally return the received message. For each transition, it also generates a transition-specific *singleton type* to serve as a type-level identifier for that event, which we refer to as *event types*.

Figure 4.5 summarizes the key API elements generated for `m` in `RunDr`, i.e., from the CFSM pair in Figure 4.4. Select and branch states (including the unary cases) have methods named `!` and `?`, respectively. The single method of a spawn or suspicion state is named `init` and `failure`, respectively. The corresponding event type is given next to each I/O

State	Channel types	Chanel method signatures	Event types
1	`M1`	`!(InitDr): M2`	`SndInitDr`
2	`M2`	`?(): (Ack, M3)`	`RcvAck`
3	`M3`	`!(AddEx): M4`	`SndAddEx`
		`!(Ok): End`	`SndOk`
4	`M4`	`init(...): M3`	`SpwRunEx`
5, 9	`End`		
6	`M6`	`failure(): M7`	`SuswD`
7	`M7`	`!(FailDr): M8`	`SndFailDr`
8	`M8`	`init(...): End`	`SpwRunDr`

(Note: `!` and `?` are method names.)

Figure 4.5.: Generated Scala API for session-typed EDP implementation of a `Masters` participant by the user.

method signature. To simplify this presentation, we assume message labels are unique across the subprotocol, and event type *values* are referred to by same name as their type. We illustrate how a programmer may use this API next.

4.3.2. MPST-based event-driven programming in Scala

MPST theory has primarily been developed in the context of the π-calculus, where the model of concurrency centers on parallel composition of sequential processes. From a practical perspective, it is akin to *multithreaded* programming with message passing, where each user process (or thread) calls the runtime system to perform its I/O (and may be blocked until the action is completed).

This chapter proposes a novel *event-driven* model of concurrency for multiparty sessions. It is characterized by an *inversion of control*: the user relies on the runtime system to monitor a set of concurrent event sources (e.g., channels), and dispatch the processing of each event occurrence (e.g., message arrivals) by *calling back* the appropriate routines in the user code; user code is never blocked because it is only called when the event is ready. Our approach is inspired by the fact that EDP is a major paradigm for distributed programming in practice; it used to implement applications featuring all the features we have discussed, including the Spark CM[4]. As we shall see, event-driven concurrency is key

[4] For example, for event loop of master see lines 228–554 in `https://github.com/apache/spark/blob/1c3bdabc03117494ffbf8fd6863ea82d4961379b/core/src/main/scala/org/apache/spark/deploy/master/Master.scala`

to realizing our framework by unifying the handling of all the "regular" session I/O and failure events under a uniform session-typeable programming interface.

The central abstractions in EDP are the *event loop* and *event handlers*. In our setting, each participant process is a session event loop that monitors a set of (sub)session channels for I/O events and dispatches each by firing its corresponding session-typed event handler.

Session-typed event handlers. Our framework provides a library function for creating handlers:

```
λ(event1, event2) {
    case (s, c: ChanType) =>
        /* handler body: return the channel after event2 */
}
```

The above is a function call that takes three arguments. We have named it λ here to assist comparison with our formal notation (Section 4.4.3). In parentheses are values of the singleton event types. The first specifies the immediate event type to be handled (i.e., the event that triggers the firing of this handler). The second specifies the last I/O action that this handler will perform before handing the resulting channel back to the event loop for monitoring. We refer to these as the *initial* and *return* events of this handler. The return event must be reachable from the initial in the CFSM, and every action after the initial must be *non*-blocking (i.e., output). A handler that performs only an initial I/O action specifies the same event for both.

The third argument is the handler function itself, partial function written in Scala as `{ case (s, c) => ... }`. Its parameters `(s, c)` are supplied by the framework when the event loop calls this function to handle the specified event: `c` is the subsession channel on which the event has occurred and is ready to be handled, and `s` is just a plain data object for storing and retrieving data across separate handlers. Guided by *static* Scala typing on the channel types, the user implements the handler body to handle the initial event and perform any subsequent I/O actions up to and including the return event. The framework and generated APIs are designed such that static type checking *ensures*: (i) the type of the channel parameter `c` corresponds to the initial event, and (ii) the handler returns a pair `(s2, c2)` back to the event loop, where `s2` is the (possibly updated) data object and `c2` is a channel whose type corresponds to the return event, i.e., the channel yielded by the last I/O method performed. We write the type annotations on `c` but they may be omitted.

Figure 4.6 demonstrates a set of *minimal* handler implementations for `Masters` in `RunDr` (Figure 4.2); in general, handlers may also contain other arbitrary Scala code. The first handler performs the single output action of its sole event. Its channel parameter type `M1` corresponds to its initial event `SndInitDr`; static typing ensures we perform `!` with the message `InitDr(...)` (payload arguments omitted) and return the resulting channel to satisfy the return event. The second handles its input event similarly.

```
(Initial, Return) events
    Initial-to-Return event handler functions
λ(SndInitDr, SndInitDr)  // Payload args omitted
    { case (s, c: M1) => (s, c ! InitDr(...)) }
λ(RcvAck, RcvAck)        // ._2 accesses 2nd tuple element
    { case (s, c: M2) => (s, (c?())._2) }
λ(SndAddEx, SndAddEx)
    { case (s, c: M3) if s.workRemaining() => (s, c ! AddEx()) }
λ(SndOk, SndOk)
    { case (s, c: M3) => (s, c ! Ok()) }
λ(SpwRunEx, SpwRunEx)
    { case (s, c: M4) => (s, c.init(...)) }
λ(SuswD, SndFailDr)
    { case (s, c: M6) => (s, c.failure() ! FailDr(s.appId)) }
λ(SpwRunDr, SpwRunDr)
    { case (s, c: M8) => (s, c.init(...)) }
```

Figure 4.6.: User-written event handlers for a `Masters` endpoint program using the generated API.

The `(SuswD, SndFailDr)` handler is an example of performing multiple I/O actions in one handler. In the `SndAddEx` handler, we use this Scala syntax for partial functions: `{ case (s, c) if /*cond*/ => ... }`. Scala checks the additional `if` clause at run-time before allowing the function to be applied; i.e., this `SndAddEx` handler is only fired if `s.workRemaining()` is true then.

Session event loops. A user constructs an endpoint program by registering event handlers like the above with the *event loop* runtime provided by our framework as a library. The handlers must cover *all* possible events that the endpoint may participate in according to the protocol. Our theory checks such *event coverage*, and that each handler is non-blocking apart from the initial event, as part of *static* type checking (Section 4.6.1). Our prototype implementation checks these dynamically when the list of handlers is registered (i.e., when the endpoint is first initialized); however, these are simple to check and would be straightforward to implement statically, e.g., as a compiler plugin.

The event loop library uses a lightweight internal representation of the CFSMs (built by the API generation) to track the current state of the local endpoint in each of its subsessions at run-time, and thereby determine which handlers are eligible for firing. The current state is the only precondition for firing an output handler (i.e., outputs are non-blocking). In addition to the current state, the event loop fires an input handler when a message is available in the input queue. A spawn handler is fired when the subsession initiation synchronization is complete. Following our theory, our prototype separates the mechanism

for failure detection from the event loop itself. The former may be implemented in various ways (e.g., heartbeats); the latter simply reacts to a failure *suspicion* event by switching to the failure handling CFSM and firing the suspicion event handler (e.g., `SuswD`).

4.3.3. Failure model and properties

We make the following practical assumptions on distributed systems (see Appendix B.3 for details): *asynchronous system* – there are no upper bounds on processes' relative speeds or message transmission delays [FLP85]; *crash(-stop) failures* – processes can fail by halting prematurely, except *robust roles* – roles played by failure-resilient processes, one being minimally required for our (progress) properties to hold; *reliable FIFO communication* – messages between non-failed processes are received eventually, in the order sent [BCT96]; *monitors* – failures of non-robust roles are detected by designated (possibly failure-prone) peers, with the possibility of *false suspicions* – inevitable in the above model [CT96]. Note that suspicions are not reverted.

The next sections present a formal model of our framework as a novel *event-driven* formulation of a multiparty session π-calculus, and establish the properties that it guarantees. Formally, a well-typed system – i.e., a system of multiple, dynamically spawned concurrent subsessions – is guaranteed to enjoy *communication safety* and *global progress*. In short: a participant will *never* receive a message for which it does not have a handler, and the system as a whole will *never* be stuck (some subsession can always progress), despite processes concurrently engaged in multiple subsessions. Prior MPST systems offer progress only when individual participants engage in a *single* session; e.g., the "simple" condition of Deniélou et al. [DYBH12, p. 30, item 1] and Demangeon and Honda [DH12, Prop. 3] (e.g., consider two *well-typed* processes p_1, p_2 in *two* sessions s_1, s_2, where p_1 first receives on s_1 but p_2 on s_2 – deadlock). We note Coppo et al. [CDYP16] study global progress using an *additional* interaction type system on top of MPSTs. The intuition for global progress in our framework is that event-driven processes by definition never engage in an I/O action unless the event is *ready* for processing, and this characteristic is *independent* of the number of active (sub)sessions. Second, our framework ensures that a protocol is safely structured (Section 4.4.1) in both "regular" interactions and handling of *all* potential participant failures, and every participant process safely provides compliant handlers for *every* potential event it may engage in (Section 4.6.1).

Using our Scala toolchain, the generated API types ensure communication safety via the native Scala type system modulo *linear usage* of each channel instance, i.e., the user should call *exactly one* I/O method on each channel instance in a handler until the return event is reached. Our theory integrates linearity into the static type system, as standard in session types. Our implementation checks linearity at runtime [Pad17, CHJ+19] and

$$
\begin{array}{lll}
\mathcal{G} & ::= & \{g_i(\bar{r}; r; \overline{R}) = G \text{ with } r@r \,.\, G\}_{i \in I} \qquad z ::= r \mid R \\
G & ::= & r \to z\,\{l_i : G_i\}_{i \in I} \mid g(\bar{r}; R; \overline{R}) \,.\, G \mid \mu \mathtt{t} \,.\, G \mid \mathtt{t} \mid \mathsf{end} \\
\\
\mathcal{L} & ::= & \{g_{i_z}(\bar{r}; r; \overline{R}) = L\}_{i \in I} \\
L & ::= & u!\{l_i : L_i\}_{i \in I} \mid y?\{l_i : L_i\}_{i \in I} \mid g(\bar{y}; R; \overline{R}) \,.\, L \mid L \text{ with } L \\
 & \mid & y{\downarrow} \,.\, L \mid - \text{ with } L \mid \mu \mathtt{t} \,.\, L \mid \mathtt{t} \mid \mathsf{end} \mid \mathsf{end}_L \\
u & ::= & y \mid R \qquad y ::= r \mid p{:}r
\end{array}
$$

Figure 4.7.: Global types and local types.

can safely treat violations as failures. Note, this is *not* a fundamental limitation (it is straightforward to adjust the API design so channels are never directly exposed to the user, e.g., [ZFH+20]), but we opt for the current style to correspond with our formal presentation. Regarding progress: since a user may include arbitrary Scala code within handlers and the `if`-clause of partial functions, the practical progress guarantee is modulo termination of handler code and exhaustiveness of the relevant handler `if`-conditions. *Regardless* of linearity and progress, however, our toolchain guarantees a well-typed Scala endpoint program will *never* perform session I/O that is not compliant with the source global protocol, other than premature termination due to failure.

4.4. Global Types, Local Types and Event-Driven Session Processes

This section formally defines our global types, local types, and event-driven session processes.

4.4.1. Global types for specifying fault-tolerant, multiparty protocols

Figure 4.7 defines the syntax of *global types*. Message *labels* range over $l, l_1, l_2, \ldots$. *Type variables* range over $\mathtt{t}, \mathtt{t}_1, \mathtt{t}_2 \ldots$. *Subprotocol names* range over $g, g_1, g_2, \ldots$. A *role set* represents a set of participant processes that support a common behavior, e.g., a set of worker processes. A *role* represents a participant drawn from a role set to fulfill some additional *individual* behavior, e.g., a worker that runs a Spark driver. Role set names range over $R, R_1, R_2, \ldots$, and *role names* range over $r, r_1, r_2, \ldots$. We may write $r \in R$ to clarify that R is the role set of r. A role set R, always refers to *all* participant processes in the set, whereas a corresponding role r refers to *one* of these participants. An overbar, e.g., $\overline{R}$

denotes a sequence, and may be empty unless explicitly stated otherwise. In examples, role set names start with an uppercase letter and roles with a lowercase.

A *top-level* global type $\mathcal{G}$ describes a complete protocol comprising a set of distinctly named *subprotocols*. $g(\bar{r}; r; \overline{R}) = G_1$ with $r@r' \,.\, G_2$ defines a subprotocol, named g, between the participants given by the three parameter segments: (1) a non-empty sequence of distinct *roles* $\bar{r}$; (2) a role r *assigned* from R ($r \in R$) when g is spawned; and (3) a sequence of distinct *role sets* $\overline{R}$. Note, a role r and its role set R may both be present, meaning the participant assigned to r fulfills *both* behaviors concurrently; e.g., the participant assigned to `wD` in `RunDr` (see Figure 4.2) behaves as `wD` *and* a member of `Workers`. Multiple roles r_1, r_2 may be from the same role set R. G_1 is the *normal activity* of g, whereas the shaded parts relate to failure handling. $r@r'$ specifies that r' is the *monitor* of the assigned r, and G_2 is the corresponding *failure handling activity* (or simply, *failure handling*); r is not permitted to occur in its own failure handling. **Note**: our formalism assumes the message labels (resp. subprotocol names) in the normal activity are disjoint from those in the failure handling.

A *spawn* type $g(\bar{r}; R; \overline{R}) \,.\, G$ specifies spawning a new subsession instance of subprotocol g that runs *concurrently* with the continuation G. The spawn arguments are in three segments corresponding to the parameters of g: non-empty, distinct roles $\bar{r}$; a role set R from which to assign a participant; and distinct role sets $\overline{R}$. Spawns in failure handlings and recursive spawns are permitted. A branch type $r \to z\,\{l_i : G_i\}_{i \in I}$ specifies that r decides which case the protocol will follow by sending a message labeled l_i to another role $z = r'$ or by *multisending* messages labeled l_i to a role set $z = R'$; the protocol then proceeds according to G_i. Following MPSTs tradition, for the former we assume $r \neq r'$ (note, no two roles in one subsession are ever played by the same participant); for the latter, we assume $R' \neq R$, where $r \in R$. We may write $r \to z\ \ l \,.\, G$ for short when I is a singleton. $\mu \mathtt{t} \,.\, G$ and t allow recursive types (our model adopts an iso-recursive approach). We assume type variable occurrences are bound and guarded in the usual way (e.g., $\mu \mathtt{t} \,.\, \mathtt{t}$ is not permitted). Lastly, end denotes subprotocol termination (often elided for brevity).

Example 14. *The top-level global type $\mathcal{G}_{CM}$ corresponding to Figure 4.2 comprises two subprotocols:*

$$
\begin{aligned}
&g_{RunDr}(m; w_D; W) = \\
&\quad m \to w_D\ l_{InitDr} \,.\, w_D \to m\ l_{Ack} \,.\, \mu\,\mathtt{t} \,.\, m \to w_D\ \{\, l_{AddEx} : g_{RunEx}(m, w_D; W; W) \,.\, \mathtt{t}\ , \\
&\qquad l_{Ok} : \mathsf{end}\,\}\ \ \text{with}\ \ w_D@m \,.\, m \to W\ l_{FailDr} \,.\, g_{RunDr}(m; W; W) \,.\, \mathsf{end} \\
&g_{RunEx}(m, w_D; w_E; W) = \\
&\quad m \to w_E\ l_{InitEx} \,.\, w_E \to m\ l_{ExDone} \,.\, m \to w_D\ l_{ExFinished} \,.\, \mathsf{end} \\
&\qquad \text{with}\ \ w_E@m \,.\, m \to W\ l_{FailEx} \,.\, g_{RunEx}(m, w_D; W; W) \,.\, \mathsf{end}
\end{aligned}
$$

Well-formedness. Following the standard approach in MPSTs, we impose basic constraints

to prevent global types that are not safely realizable as distributed protocols. (1) As standard, our system considers only global types for which *projection* (Section 4.4.2) is well-defined for every participant kind: a key point is to ensure consistency of multiparty branching, which leads to the next condition. (2) We require the G in a monitor term $r_1@r_2 \,.\, G$ to start with the monitor (r_2) sending some label (i.e., *failure notification*) to all other roles (except the monitored role r_1) and role sets in the subprotocol. This ensures that the monitor soundly informs all other participants if it suspects a failure. E.g., in both failure handlings in Example 14 and Figure 4.2, the monitor m informs all workers W about the failure (note, $w_D \in W$). Appendix B.4 provides the straightforward formal definition of well-formedness.

4.4.2. Local types and endpoint projection

Figure 4.7 defines the syntax of local types, which represent the view of a protocol localized to one kind of participant, i.e., the combined view of a role set and its role assignments. Local types are used to *statically* ensure that a participant process (i.e., an *event loop*) safely implements the full behavior of its role set and potential role assignments. Parts related to failure handling have darker shading, and runtime terms (i.e., only occurring during execution) are lightly shaded; some terms are both. As explained in Sections 4.4.3 and 4.5, our formalism also reuses local type syntax as an artifact for recording the current state of subprotocols at runtime, to drive event loops and handler firing.

Participant names (i.e., process identifiers) range over $p, p_1, p_2, ..., q, q_1, q_2, ...$ Meta symbol u stands for a role r or role set R; and y for a role r, or the additional runtime annotation $p{:}r$ specifying that participant p *is playing role* r in a given subsession. E.g., assume participants $\{p_1, .., p_n\}$ constitute a worker role set W and the protocol involves a role $w \in W$. Although any of $\{p_1, .., p_n\}$ may play w, the specific p_j picked to play w at runtime is recorded by the annotation $p_j{:}w$.

A *top-level* local type $\mathcal{L}$ describes a complete localized protocol comprising a set of *local* subprotocols, as yielded by projection (explained shortly). $g_z(\bar{r}; r; \overline{R}) = L$ defines the local subprotocol view of g from the annotated z, i.e., a local role or role set. $g(\bar{y}; R; \overline{R}).L$ is a local spawn type for subprotocol g. The three argument/parameter segments are as explained for their global type counterparts: non-empty, distinct roles $\bar{y}$ to play $\bar{r}$; a role set R from which to assign a participant to play r; and distinct role sets $\overline{R}$. The send/select type $u!\{l_i : L_i\}_{i \in I}$ specifies: select and (multi)send l_i to u, then proceed as L_i. Receive/branch $y?\{l_i : L_i\}_{i \in I}$ is the dual: receive an l_i, then proceed as L_i; note, we receive only from a role (not a role set). L_1 with L_2 represents a normal activity L_1 under execution and the as yet *inactive* failure handling L_2. For the monitor, L_2 is (initially) a *suspicion* type $y{\downarrow} \,.\, L$ specifying: monitor y and switch to L (i.e., switch from the normal

to the failure activity) upon the suspected failure of y. There is no dual to $y\downarrow . L$ because failing is an implicit action that can occur at any time. $-$ with L represents an *active* failure handling, and only occurs at runtime. $\mu \mathtt{t}.L$ and $\mathtt{t}$ are for recursive types. end (often elided) represents the termination of a normal or failure handling activity. A *stopped* type end_L represents a behavior that is (prematurely) halted due to failure handling being activated in some *ancestor* subsession, causing all child subsession activities to be halted. The L annotation specifies the point at which the behavior was halted.

We use a context notation $\mathcal{E}[\cdot]$ for local types (defined: $\mathcal{E} ::= [] \mid \mathcal{E}$ with $L \mid -$ with $\mathcal{E}$) to access a normal or *active* failure handling activity in with terms, e.g., in reduction rules (Section 4.5).

Projection. Our typing statically associates every participant with a *role set*. However, a safe implementation of a participant process must also be capable of all potential *role assignments*. Our notion of *endpoint projection* [HYC16, CDYP16] is thus the projection of a top-level global type onto a role set *together* with the projections onto each of its role assignments. E.g., for the subprotocol g_{RunEx} in Example 14 and Figure 4.2, the top-level local type of a `Workers` participant includes multiple local subprotocols: $g_{RunEx_{w_D}}$, $g_{RunEx_{w_E}}$ and g_{RunEx_W} (see Appendix B.2 for all).

Definition 18 (Projection $\mathcal{G} \upharpoonright R = \mathcal{L}$)**.** *The projection of a* top-level global type $\mathcal{G}$ *onto a* role set R *is* $\mathcal{G} \upharpoonright R = \{g \upharpoonright R \mid g(\bar{r}; r_1; \overline{R}) \in \mathcal{G} \ \wedge \ R \in \overline{R}\} \cup \{g \upharpoonright r_2 \mid g(\bar{r}; r_1; \overline{R}) \in \mathcal{G} \ \wedge \ r_2 \in \bar{r} \cdot r_1\}$.

The LHS of the union projects *subprotocol* g onto *role set* R ($g \upharpoonright R$) if g uses R. The RHS projects g onto *role* r_2 ($g \upharpoonright r_2$) if g uses r_2. We define $g \upharpoonright r$ below. We leave the definition of $g \upharpoonright R$ to Appendix B.4.1; it is very similar but simpler than $g \upharpoonright r$.

Definition 19 (Projection $g \upharpoonright r = g_r$)**.** *The projection of a* subprotocol g *onto a* role r*, and the projection of a* global type G *onto a role* r *(i.e.,* $G \upharpoonright r = L$*), are respectively defined:*

$\boxed{g \upharpoonright r = g_r}$

$$\big(g(\bar{r}; r_1; \overline{R}) = G_1 \text{ with } r_1@r_2 \,.\, G_2\big) \upharpoonright r = \begin{cases} g_r(\bar{r}; r_1; \overline{R}) = G_1 \upharpoonright r \text{ with } r_1\downarrow.G_2 \upharpoonright r & \textit{if } r = r_2, \\ g_r(\bar{r}; r_1; \overline{R}) = G_1 \upharpoonright r \text{ with } G_2 \upharpoonright r & \textit{if } r \in \bar{r}, \\ g_r(\bar{r}; r_1; \overline{R}) = G_1 \upharpoonright r \text{ with end} & \textit{if } r = r_1 \end{cases}$$

$\boxed{G \upharpoonright r = L}$

$$r_1 \to z\,\{l_i : G_i\}_{i \in I} \upharpoonright r = \begin{cases} z!\{l_i : G_i \upharpoonright r\}_{i \in I} & \textit{if } r = r_1, \\ r_1?\{l_i : G_i \upharpoonright r\}_{i \in I} & \textit{else if } z = r \vee r \in z, \\ G_1 \upharpoonright r & \textit{else if } \forall i, j \in I \,.\; G_i \upharpoonright r = G_j \upharpoonright r \end{cases}$$

$$(g(\bar{r}; R; \overline{R}) \,.\, G) \upharpoonright r = \left\{ \begin{array}{ll} g(\bar{r}; R; \overline{R}) \,.\, (G \upharpoonright r) & \textit{if } r \in \bar{r} \vee \exists R' \in \{R, \overline{R}\} \,.\, r \in R', \\ G \upharpoonright r & \textit{otherwise} \end{array} \right\}$$

$$\mu t.G \upharpoonright r = \{\ \mu t.(G \upharpoonright r) \ \textit{if } G \upharpoonright r \neq t, \quad \text{end} \ \textit{otherwise}\ \} \qquad t \upharpoonright r = t \qquad \text{end} \upharpoonright r = \text{end}$$

$$
\begin{array}{lll}
P & ::= & (\overline{H}, p) \mid c[u]!l.P \mid c[r]?l.P \mid c[\bar{r}; R; \overline{R}](g) \mid c[r]{\downarrow}.P \mid \mathsf{loop} \\
N & ::= & P \mid N \,||\, N \mid (\nu s : \mathcal{G})\, N \mid s[p] : (L, b) \mid 0 \\
H & ::= & [L]\lambda x.P \quad c ::= x \mid s[p] \quad b ::= [p \mapsto \bar{l}] \quad \Theta ::= [s \mapsto (\widetilde{p}, \widetilde{s})] \quad \mathcal{F} ::= \widetilde{p}
\end{array}
$$

Figure 4.8.: Syntax of participant processes and networks.

The top rule projects the normal activity of a subprotocol onto r, and projects the failure handling according to r's involvement in monitoring. If r is the *monitor*, the projection of G_2 is prefixed by a monitor type $r_1{\downarrow}$. If r is neither the monitor nor the monitored role, then we just project G_2. If r is the *monitored* role, then failure handling is set to end – r is not involved in the handling of its own failure! We explicitly project the failure handling of the monitored role r as end, even if the role set R of r occurs in the global failure handling. The rule below the dashed line is the standard projection (e.g., Coppo et al. [CDYP16]) of an interaction according to the role's involvement, with the additional clause for a multisend to the role set R of r (via the condition $r \in z$) – the latter projects to a receive because the participant playing a assigned role r is still a member of R. The next rule projects a spawn term and its continuation onto r: the spawn is retained if r or the role set of r occurs in the arguments. The recursion and termination rules are standard.

Example 15. *The projection $\mathcal{G}_{CM} \upharpoonright M = \{\, g_{RunDr} \upharpoonright m,\ g_{RunEx} \upharpoonright m \,\}$ (i.e., both as role m), yielding:*

$$
\begin{array}{l}
g_{RunDr_m}(m; w_D; W) = \\
\quad w_D!l_{InitDr} \,.\, w_D?l_{Ack} \,.\, \mu\,\mathtt{t} \,.\, W\,!\{\, l_{AddEx} : g_{RunEx}(m, w_D; W; W) \,.\, \mathtt{t}\,,\ l_{Ok} : \mathsf{end} \,\} \\
\quad \mathsf{with}\ \ w_D{\downarrow} \,.\, W!l_{FailDr} \,.\, g_{RunDr}(m; W; W) \,.\, \mathsf{end} \\
g_{RunEx_m}(m, w_D; w_E; W) = \\
\quad w_E!l_{InitEx} \,.\, w_E?l_{ExDone} \,.\, w_D!l_{ExFinished} \,.\, \mathsf{end} \\
\quad \mathsf{with}\ \ w_E{\downarrow} \,.\, W!l_{FailEx} \,.\, g_{RunEx}(m, w_D; W; W) \,.\, \mathsf{end}
\end{array}
$$

Our toolchain represents local types as CFSMs as explained in Section 4.3.1; simply put, each action in the local type is a CFSM transition, and the normal and failure handling activities are represented as separate CFSMs. E.g., g_{RunDr_m} is represented by the CFSM pair in Figure 4.4. See Appendix B.2 for $\mathcal{G}_{CM} \upharpoonright W$, which includes both role ($g_i \upharpoonright w_D$ and $g_i \upharpoonright w_E$) and *generic role set ($g_i \upharpoonright W$) behaviors.*

4.4.3. Event-driven session processes and networks

Figure 4.8 defines the syntax of *participant processes* (or simply *processes*), denoted by $P, P_1, P_2, \ldots$, and *networks*, denoted by $N, N_1, N_2, \ldots$. Failure handling terms have darker

shading, and runtime terms (i.e., that arise only during execution) are lightly shaded; some are both. *(Sub)session names* range over $s, s_1, s_2, \ldots$. Meta symbol c denotes a *session channel* (or simply, *channel*) $s[p]$, or a *channel variable* ranging over $x, x_1, \ldots$. We write $\widetilde{x}$ for a potentially empty set $\{x_i\}_{i \in I}$.

The heart of a participant process is an *event loop* and its *event handlers* (or simply, *handlers*). $(\overline{H}, p)$ is an *event loop* for participant p with handlers $\overline{H}$. As discussed in Section 4.5, an event loop encapsulates and monitors a set of channels for I/O events, and handles each event occurrence by *firing* a handler, which is executed to completion before the event loop resumes. The handler is dynamically selected according to the event, e.g., the message received, *and* the current state of the participant in the subprotocol for this subsession. As mentioned, every participant p is statically associated with a role set R: we write R_{ids} for the set of all participants associated with R.

A *handler* $[L]\lambda x.P$ defines a session computation. Channel variable x is substituted for the event-ready channel when the handler is fired, P is the handler body, and *guard type* L specifies the initial event type and I/O behavior to perform on x in P. For comparison, a handler is written using our toolchain (Section 4.3.2): `λ(/*guard type*/) { case (..., x) => /*P*/ }`. (The "initial" and "return" events there correspond to our single guard type L here.) The idea is that a handler is *only* fired if it will *not* block – P may contain a blocking action like receive, but is only fired if the expected message is present. To this end, our type system (Section 4.6.1) restricts the guard type to use: (a) at most one *blocking* action, which must be the initial action; (b) no recursion; (c) no branching or selection, i.e., be *flat*. These restrictions are natural to EDP and help ensure progress *without* limiting expressiveness: *branching and selection are expressed in our system via* separate *handlers, and recursion manifests as recurring events and repeat handler firings* (see Examples 16 and 20).

The other P terms occur inside handler bodies. $c[u]!l.P$ performs a *(multi)send* of label l on channel c to a role or role set u, followed by P; $c[r]?l.P$ is the *receive* counterpart. For simplicity, we do not model message payload types/values. A *spawn* $c[\bar{r}; R; \overline{R}](g)$ uses channel c to initiate a new subsession for g. The participants joining the new subsession from the current subsession are: roles $\bar{r}$, a participant assigned from R, and role sets $\overline{R}$. $c[r]\downarrow.P$ is the action by a monitor signifying it *suspects* r has failed and will switch to P. loop denotes the completion of a *handler* execution and resumption of the event loop: loop is substituted for the original event loop term when a handler is fired, hence we distinguish it from the typical 0 term. We may omit loop in examples; e.g., we write $\lambda x\,.\,x[w_D]!l_{InitDr}$ with the trailing $.\,$loop omitted.

Example 16. *Let $(\overline{H}, p)$ be an event loop for an M participant from Example 14 and Figure 4.2. p is the participant identifier. $\overline{H}$ contains, e.g., the handlers $[W!l_{AddEx}]\lambda x\,.\,x[W]!l_{AddEx}$*

and $[W!l_{Ok}]\lambda x\,.\,x[W]!l_{Ok}$ *– cf. the* `SndAddEx` *and* `SndOk` *handlers in Figure 4.6 (similarly for the others; see Appendix B.2.4 for all). These two handlers implement the* selection *from* `RunDr` *(i.e.,* $W\,!\{\,l_{AddEx}:\ldots\,,\,l_{Ok}:\ldots\}$*); the first covers the* l_{AddEx} *choice, and the second covers* l_{Ok}*. Compared to Figure 4.6, our formalism omits the* `if`*-clause – we abstract from the internal decision procedure and simply model it non-deterministically.*

Network $N \,||\, N$ is a standard *parallel composition*, for composing participant event loops and configurations into a complete session system. We write $||_{p\,\in\,\widetilde{p}}\,(N)$ for a composition of N terms for each p. For each process P in a network, we assume all terms within P relate to a single participant p. In our system, *restriction* $(\nu s : \mathcal{G})\ N$ is applied only to the root session, to annotate the $\mathcal{G}$; for simplicity, we assume subsessions are always created with fresh names. An *endpoint configuration* (or simply, *configuration*) $s[p] : (L, b)$ records the runtime information used by participant p to execute its part in subsession s: a local type L, called the current *subprotocol state*, and an *asynchronous input buffer* b, which holds a FIFO per subsession peer for incoming messages. An empty buffer is written ε. The subprotocol state serves two purposes: to designate the next set of events expected in subsession s and thereby direct (in conjunction with the event occurrence) which handler to fire, and to record the binding of (peer) roles to participant names. We opt to represent the notion of subprotocol state by reusing local type syntax; however, this mechanism may be implemented in various ways (e.g., we use an FSM), and we emphasize our system is fully *statically* typed.

Example 17. *Take the event loop from Example 16, for an* M *participant* p*, and two configurations* $s[p] : (\mathcal{E}[p_1{:}w_D!l_{InitDr}\,.\;\,p_1{:}w_D?l_{l_{Ack}}\,.\,L'], b_1)$ *and* $s[p] : (\mathcal{E}[p_1{:}w_D?l_{l_{Ack}}\,.\,L'], b_2)$ *for some* L'*. The event loop can fire the handler* $[w_D!l_{InitDr}]\lambda x\,.\;x[w_D]!l_{InitDr}$ *under the first configuration, but not the second. The subprotocol state in the first (resp. second) expects to send* l_{InitDr} *(resp. to receive* $l_{l_{Ack}}$*).*

The remaining items are runtime environments for process reduction: Θ is the *monitoring tree*, i.e., the parent-child subsession relation. $\mathcal{F}$ is the *fail set*, which records all *suspected* participants. We can model these environments in a global fashion since their decomposition into local views (as implemented in our prototype) is straightforward and has no significant impact on the overall design. Regarding Θ, a participant is concerned only with the subtree pertaining to the subsessions itself is involved in: a lookup in a local or a global view is no different. Regarding $\mathcal{F}$, as we shall see, participants are only ever *added* to $\mathcal{F}$ (i.e., crash-*stop* failures): it is straightforward (even in an asynchronous system) to have participants converge on a consistent view, which is all we require.

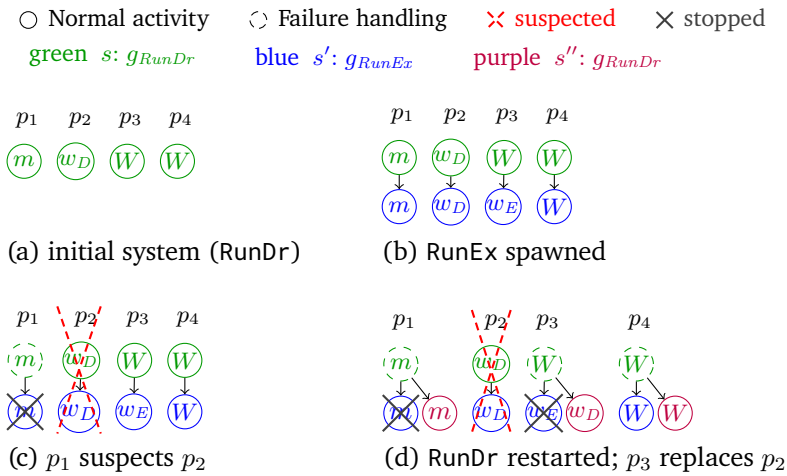

Figure 4.9.: Participant engagement in subsessions in a scenario where the driver process fails (following Figure 4.2).

4.5. Operational Semantics for Event-Driven Concurrent Subsessions

We define a *reduction* relation on *systems* of the form $(\Theta, \mathcal{F}, N)$. Relation $(\Theta_1, \mathcal{F}_1, N_1) \rightarrow (\Theta_2, \mathcal{F}_2, N_2)$ reduces a network N_1 under environments Θ_1 and $\mathcal{F}_1$ to N_2 and updated environments Θ_2 and $\mathcal{F}_2$. We streamline our formalism by assuming the root subsession to be already established in an *initial system*: this allows us to omit the usual "shared channel" prefixes used to bootstrap a session – that aside, our model freely supports concurrent subsession spawning, based on the standard session initiation in MPSTs. We present the reduction rules in three parts: event loops and handler firing, subsession spawning, and failure handling. *On notation*: we (1) omit Θ and/or $\mathcal{F}$, for brevity, from a rule definition if they are neither used nor modified; (2) write '_' to denote an irrelevant element.

We continue with the session-typed CM (Session-CM) as running example. Figure 4.9 illustrates the system over a few reduction steps in a scenario where the driver participant fails after a g_{RunEx} subsession has already been spawned: Figure 4.9 (a) shows the initial system from Example 18; Example 21 demonstrates spawning the g_{RunEx} protocol, leading to (b); and in Example 22, the participant playing w_D fails and g_{RunDr} is restarted, corresponding to (c) and (d).

Example 18. *Figure 4.9 (a) depicts an* initial system *for Session-CM. This root subsession s involves participants p_1 in role set M and $p_{1..3}$ in W: the monitoring tree is $\Theta_0 =$*

$[s \mapsto \{p_1, p_2, p_3, p_4\}]$. *No failures have occurred yet: the failure set is* $\mathcal{F}_0 = \emptyset$. *This system is thus* $(\Theta_0, \mathcal{F}_0, N_0)$, *where* $I = \{1..4\}$, $N_0 = (\nu s : \mathcal{G}_{CM})\ \big(\|_{i \in I}\ (\overline{H}_i, p_i)\ \|_{i \in I}\ s[p_i] : (L_i, \varepsilon)\big)$ *and*
$L_1 = p_2{:}w_D!l_{InitDr}\,.\,p_2{:}w_D?l_{Ack}\,.\,\mu\mathtt{t}\,.\,W!\{l_{AddEx} : g_{RunEx}(p_1{:}m, p_2{:}w_D; W; W)\,.\,\mathtt{t}\,,\ l_{Ok} : \mathsf{end}\}$ with ...
$L_2 = p_1{:}m?l_{InitDr}\,.\,p_1{:}m!l_{Ack}\,.\,\mu\mathtt{t}\,.\,p_1{:}m?\{l_{AddEx} : g_{RunEx}(p_1{:}m, p_2{:}w_D; W; W)\,.\,\mathtt{t}\,,\ l_{Ok} : \mathsf{end}\}$ with ...
$L_k = \mu\mathtt{t}\,.\,p_1{:}m?\{l_{AddEx} : g_{RunEx}(p_1{:}m, p_2{:}w_D; W; W)\,.\,\mathtt{t}\,,\ l_{Ok} : \mathsf{end}\}$ with ... $\quad k \in \{3, 4\}$

The $(\nu s : \mathcal{G}_{CM})$ *specifies that* s *is an instance of the root subprotocol* g_{RunDr}. *Example 16 gave an example of* $\overline{H}_1$, *i.e., a handler list for an* M *participant;* $\overline{H}_{2..4}$ *may be implemented similarly. Every configuration has an empty buffer. The initial subprotocol state* L_1 *of* p_1 *is essentially the projection of* g_{RunDr} *onto* m *annotated with the runtime bindings of roles to participants, i.e.,* p_1 *is playing* m *(i.e.,* $p_1{:}m$*) and* p_2 *is playing* w_D *(i.e.,* $p_2{:}w_D$*); similarly for* $L_{3/4}$. *Subprotocol state* L_2 *differs from* $L_{3/4}$ *because* p_2 *has been assigned to* w_D *whereas* $p_{3/4}$ *are still generic members of role set* W. *That is,* L_2 *describes the combined behavior of* w_D *and* W, *whereas* $L_{3/4}$ *describes* W *only.*

4.5.1. Event loops and handler firing

The main reduction rules are defined in Figure 4.10–4.13. We leave the standard structural rules (for parallel composition, restriction and structural congruence) to Appendix B.5.1. Figure 4.10 gives the core rules for event loop and handler execution. Fire is the key rule for the event loop $(\overline{H}, p)$ of participant p to fire a handler $[L_2]\lambda x.\,P$. Premise $L_1 \asymp L_2$ checks that the current *subprotocol state* L_1 of s *matches* the handler's *guard type* L_2, and $\mathsf{fire}(L_1, L_2, b, \mathcal{F})$ checks that for the required runtime information (e.g., the expected message) to ensure the handler will *not* block. The event loop term is then replaced by the handler body P with the session channel substituted for x (written $\{s[p]/x\}$), and the event loop itself substituted for loop, to return control back to the event loop after the handler is completed. When in this form, we say the event loop has an *active* handler.

The *match* predicate $L \asymp L'$ holds when local type *subtraction* $L - L'$ is defined for these types. (We define $L \asymp L'$ via $L - L'$ because the latter is anyway needed again later in the type system.)

Definition 20 (Subtraction)**.** L subtracted *by guard type* L' *(*$L - L' = L''$*) is defined in Figure 4.11.*

$L - L'$ is defined when L' is a "prefix" of the normal or failure handling activity in L. It yields the "remainder" of L after subtracting L', and converts it to an *active* failure handling if an initial failure notification is consumed. Two actions are matched if they are of the same kind and use the same roles; participant names (i.e., p in $p{:}r$) are ignored. As an example: $p_1{:}r_1?\{l_1 : p_2{:}r_2!l\,.\,L'_1,\ l_2 : L'_2\} - r_1?l_1\,.\,r_2!l = L'_1$, by consuming the l_1 and then the l within that branch case.

$\boxed{(\Theta, \mathcal{F}, N) \to (\Theta', \mathcal{F}', N')}$

$$\textsc{Fire}\ \frac{[L_2]\lambda x.\, P \in \overline{H} \qquad L_1 \asymp L_2 \qquad \mathsf{fire}(L_1, L_2, b, \mathcal{F}) \qquad N = s[p] : (L_1, b)}{(\mathcal{F}, (\overline{H}, p) \mid\mid N) \to (\mathcal{F}, P\{s[p]/x\}\{(\overline{H}, p)/\mathsf{loop}\} \mid\mid N)}$$

$$\textsc{Send}\ \frac{j \in I \qquad b'_q = b_q[p \mapsto b_q(p) \cdot l_j]}{s[p][r]!l_j.P \mid\mid s[p] : (\mathcal{E}[q{:}r!\{l_i : L_i\}_{i\in I}], \_) \mid\mid s[q] : (\_, b_q) \ \to\ P \mid\mid s[p] : (\mathcal{E}[L_j], \_) \mid\mid s[q] : (\_, b'_q)}$$

$$\textsc{Recv}\ \frac{(q \mapsto l_j \cdot \bar{l}) \in b \qquad j \in I \qquad b' = b[q \mapsto \bar{l}]}{s[p][r]?l_j.P \mid\mid s[p] : (\mathcal{E}[q{:}r?\{l_i : L_i\}_{i\in I}], b) \to P \mid\mid s[p] : (\mathcal{E}[L_j], b')}$$

$$\textsc{MSnd}\ \frac{L = \mathcal{E}[R!\{l_i : L_i\}_{i\in I}] \qquad \widetilde{p} = R_{ids} \setminus \mathcal{F} \qquad j \in I \qquad \forall q \in \widetilde{p}.\ b'_q = b_q[p \mapsto b_q(p) \cdot l_j]}{(\mathcal{F}, s[p][R]!l_j.P \mid\mid s[p] : (L, \_) \mid\mid_{q\in\widetilde{p}} s[q] : (\_, b_q)) \ \to\ (\mathcal{F}, P \mid\mid s[p] : (\mathcal{E}[L_j], \_) \mid\mid_{p\in\widetilde{p}} s[q] : (\_, b'_q))}$$

$$\textsc{Unfold}\ \ (\overline{H}, p) \mid\mid s[p] : (\mathcal{E}[\mu t.L], \_) \to (\overline{H}, p) \mid\mid s[p] : (\mathcal{E}[L\{\mu t.L/t\}], \_)$$

Figure 4.10.: Event loops (We omit Θ and/or $\mathcal{F}$ where irrelevant; N is always present).

$$\boxed{L - L' = L''} \qquad \frac{L - L_g = L''}{L \text{ with } L' - L_g = L'' \text{ with } L'} \qquad \frac{L - L_g = L'}{- \text{ with } L - L_g = - \text{ with } L'}$$

$$\frac{L' - L_g = L''}{L \text{ with } L' - L_g = - \text{ with } L''} \qquad \frac{l_j \in \{l_i\}_{i\in I} \quad y \equiv r \quad L_j - L_g = L \quad \dagger \in \{!, ?\}}{y\dagger\{l_i : L_i\}_{i\in I} - r\dagger l_j.L_g = L}$$

$$\frac{l_j \in \{l_i\}_{i\in I} \qquad L_j - L'_g = L}{R!\{l_i : L_i\}_I - R!l_j.L'_g = L} \qquad \frac{y \equiv r \quad L - L'_g = L'}{y{\downarrow}.L - r{\downarrow}.L'_g = L'} \qquad \frac{\bar{r} \equiv \bar{y}}{g(\bar{y}, R, \overline{R}).L - g(\bar{r}, R, \overline{R}).\mathsf{end} = L}$$

$$L - \mathsf{end} = L \qquad\qquad (y \equiv r \text{ means } y = p{:}r \vee y = r)$$

Figure 4.11.: L *subtracted* by guard type L'.

Predicate $\mathsf{fire}(L_1, L_2, b, \mathcal{F})$ intuitively checks whether any potentially blocking action in guard type L_2 (the initial action, at most) is executable *without* blocking under the given b and $\mathcal{F}$. The subprotocol state L_1 provides the binding of roles to participants. E.g., consider $\mathsf{fire}(p_1{:}r_1?l_1, L, b, \mathcal{F})$ with $L = r_1?l_1.r_2!l_2$. The predicate holds iff $b(p_1) = l_1 \cdot \bar{l}$; i.e., it is false if $b(p_1) = \varepsilon$ or $b(p_1) = l' \cdot \bar{l} \wedge l' \neq l_1$. The definition of fire is straightforward and left to the supplement (Appendix B.5.1).

Example 19 (Handler firing)**.** *Let* $N = (\overline{H}, p) \mid\mid s[p] : (L, b)$ *where* $L = L_1$ with ... *and* $[L']\,\lambda x.\, x[r_1]?l_1.\, x[r_2]!l'_1.\,\mathsf{loop} \in \overline{H} \qquad L' = r_1?l_1.\, r_2!l'_1 \qquad L_1 = p_1{:}r_1?\{l_1 : p_2{:}r_2!l'_1.\, L, l_2 : L'_2\}$. *Then* $L \asymp L'$*, as* L' *is a "prefix" of the normal activity* L_1*. Given* $b(p_1) = l_1 \cdot \bar{l}$ *and some* $\mathcal{F}$*, then* $\mathsf{fire}(L, L', b, \mathcal{F})$ *holds. Hence,* N *may be reduced by* Fire *to activate this handler, i.e.,*

$$(\mathcal{F}, N_1) \to (\mathcal{F}, s[p][r_1]?l_1.\, s[p][r_2]!l_2.\, (\overline{H}, p) \mid\mid s[p] : (L, b)).$$

$\boxed{(\Theta, \mathcal{F}, N) \to (\Theta', \mathcal{F}', N')}$

$$\textsc{Spawn}\ \frac{\begin{array}{c}\widetilde{p_1} = \bar{q} \cup ((R'_{ids} \cup \overline{R}_{ids}) \setminus \mathcal{F}) \qquad p_0 \in R'_{ids} \setminus \bar{q} \qquad \widetilde{p}_2 = \bar{q} \cup p_0 \cup (\overline{R}_{ids} \setminus \mathcal{F}) \qquad \widetilde{p}_2 \cap \mathcal{F} = \emptyset \\ s_2 \text{ fresh} \qquad \Theta' = \Theta[s_2 \mapsto (\widetilde{p}_2, \emptyset), s_1 \mapsto \Theta(s_1) \cup s_2] \qquad L = g(\overline{q{:}r}; R'; \overline{R}) \,.\, L_p \qquad L' = g^{\bar{q};p_0}\!\upharpoonright\! p\end{array}}{\begin{array}{c}(\Theta, \mathcal{F}, ||_{p \in \widetilde{p_1}} ((\overline{H}_p, p) \,||\, s_1[p] : (\mathcal{E}_p[L], \_))) \to \\ (\Theta', \mathcal{F}, ||_{p \in \widetilde{p_1}} ((\overline{H}_p, p) \,||\, s_1[p] : (\mathcal{E}_p[L_p], \_)) \,||_{p \in \widetilde{p}_2} s_2[p] : (L', \varepsilon))\end{array}}$$

$$\textsc{Sess-GC}\ \frac{\Theta(s_1) = (\widetilde{p}_1, \widetilde{s}) \qquad \Theta(s_2) = (\widetilde{p}_2, \emptyset) \qquad \widetilde{p} = \widetilde{p}_2 \setminus \mathcal{F} \qquad s_2 \in \widetilde{s} \qquad \forall p \in \widetilde{p}.\ \mathsf{done}(L_p)}{(\Theta, \mathcal{F}, ||_{p \in \widetilde{p}} ((\overline{H}_p, p) \,||\, s_2[p] : (L_p, \_))) \to (\Theta[s_1 \mapsto (\widetilde{p}_1, \widetilde{s} \setminus s_2)] \setminus s_2, \mathcal{F}, ||_{p \in \widetilde{p}} (\overline{H}_p, p))}$$

$$\textsc{Root-GC}\ \frac{s \text{ is the root} \qquad \Theta(s) = (\widetilde{p}_1, \emptyset) \qquad \widetilde{p} = \widetilde{p}_1 \setminus \mathcal{F} \qquad \forall p \in \widetilde{p}.\ \mathsf{done}(L_p)}{(\Theta, \mathcal{F}, ||_{p \in \widetilde{p}} (\overline{H_p}, p) \,||_{p \in \widetilde{p}} s[p] : (L_p, \_)) \to (\Theta \setminus s, \mathcal{F}, 0)}$$

Figure 4.12.: Subsession spawning and garbage collection.

I/O actions are performed by event loops with active handlers. Send dispatches label l_i from p to the participant playing r. The subprotocol state of p must have a send type with an l_i case as its normal or active failure handling activity, and provide the q bound to r; then l_i is appended to the buffer for p at q. Recv consumes a label l_i from the participant playing r, if the subprotocol state of p has a corresponding receive type, and there is an available l_i from the q bound to r. MSnd dispatches a label l_i to each non-suspected participant associated with role set R (i.e., $R_{ids} \setminus \mathcal{F}$; see Section 4.4.3 for R_{ids}). In these three rules, the local participant p updates its subprotocol state and continues as P. Unfold unfolds a recursive subprotocol state. Note, recursive behaviors are driven by recurring subprotocol states and event occurrences, leading to repeat handler firings.

Example 20 (Recursion)**.** *Let* $N' = (\overline{H}', p) || s[p] : (\mathcal{E}[\mu t.\, q{:}r!l.\, t], \_) || \ldots$ *with appropriate handlers* $\overline{H}'$*. By unfolding the recursive subprotocol state,* N' *allows repeated handler firing like:* $N' \xrightarrow{\textsc{Unfold}} N'_1 \xrightarrow{\textsc{Fire}} N'_2 \xrightarrow{\textsc{Send}} N'' \xrightarrow{\textsc{Unfold}} N''_1 \xrightarrow{\textsc{Fire}} N''_2 \xrightarrow{\textsc{Send}} \ldots$ *(showing only the main rules).*

4.5.2. Subsessions and spawn

In Figure 4.12, Spawn performs a *subsession initiation* to spawn a new concurrent instance of a subprotocol g. We base it on session initiation in standard asynchronous MPSTs (e.g, [CDYP16]), i.e., a synchronization (e.g., a multiparty handshake) that collects all the participants needed to conduct the (sub)session. Spawn is straightforward but involves several aspects: (1) ensure that all participants involved in the spawn have reached their corresponding subprotocol states and perform the spawn together; (2) determine which

participants join the new subsession; and (3) create the new subsession and update/create the configurations. We first illustrate (1)–(3).

Example 21. *Assume a step* $(\mathcal{F}_0, \Theta_0, (\nu s : \mathcal{G}_{CM})\ N_1) \xrightarrow{\textsc{Spawn}} (\mathcal{F}_0, \Theta', (\nu s : \mathcal{G}_{CM})\ N_2)$ *that spawns a new* g_{RunEx} *subsession. The LHS system (reached from Example 18 by some steps) may have*

$$N_1 = ||_{i \in I}\ (\overline{H}_i, p_i)\ ||_{i \in I}\ s[p_i] : (L_i,\ \varepsilon) \qquad L_i = g_{RunEx}(p_m{:}m, p_2{:}w_D; W; W)\,.\,L_i'' \text{ with } \ldots$$

with $I = \{1..4\}$, $\mathcal{F}_0 = \emptyset$ *and* $\Theta_0 = [s \mapsto \{p_1, p_2, p_3, p_4\}]$. *For (1),* Spawn *requires in* N_1 *that no event loop of any spawn participant has an active handler, and all subprotocol states (*L_i*) are at the corresponding spawn type. For (2),* Spawn *determines the participants of the new subsession, say* s'*, based on the above spawn arguments (inherited from* $\mathcal{G}_{CM}$*) and target subprotocol parameters, i.e.,* $g_{RunEx}(m, w_D; w_E; W)$*. Thus: (i)* p_1 *playing* m *in* s *will play* m *in* s'*, and* p_2 *playing* w_D *in* s *will play* w_D *in* s'*; (ii)* Spawn *assigns a participant from* W*, say* p_3*, to play* w_E *in* s'*; (iii) all* W *participants that do not play named roles in* s' *(i.e.,* p_4*) will behave as plain* W *in* s'*.*

Figure 4.9 (b) depicts the RHS system (i.e, that containing N_2*) after the spawn, where:*

$$N_2 = ||_{i \in I}(\overline{H}_i, p_i)\ ||_{i \in I}\ s[p_i] : (L_i'' \text{ with } \ldots,\ \varepsilon)\ ||_{i \in I}\ s'[p_i] : (L_i', \varepsilon)$$

$$L_1' = p_3{:}w_E!l_{InitEx}\,.\,p_3{:}w_E?l_{ExDone}\,.\,p_2{:}w_D!l_{ExFinished} \text{ with } \ldots$$

$$L_2' = p_1{:}m?l_{ExFinished} \text{ with } \ldots \quad L_3' = p_1{:}m?l_{InitEx}\,.\,p_1{:}m!l_{ExDone} \text{ with } \ldots \quad L_4' = \text{end with } \ldots$$

For (3), Spawn *updates the subprotocol states* L_i'' *in the* s *configurations. It creates new configurations for* s' *with these subprotocol states:* p_1 *(playing* m*) has* g_{RunEx} *projected onto* m *(*L_1'*);* p_2 *has* g_{RunEx} *projected onto* w_D *(*L_2'*);* p_3 *has* g_{RunEx} *projected onto* w_E *(*L_3'*); and* p_4 *has* g_{RunEx} *projected onto* W *(*L_4'*) – with role-to-participant bindings* $p_1{:}m, p_2{:}w_D$*,* $p_3{:}w_E$ *embedded in all these projections. It adds the new subsession* s' *to the monitoring tree, i.e.,* $\Theta' = \Theta[s' \mapsto \{p_1, p_2, p_3, p_4\}]$.

We now explain the Spawn rule and (1)–(3) in more detail. For (1), premise $\widetilde{p_1} = \bar{q} \cup ((R'_{ids} \cup \overline{R}_{ids}) \setminus \mathcal{F})$ determines all of the parent subsession (s_1) participants involved in the spawn event; i.e.: the participants bound to role arguments ($\overline{q{:}r}$), all *unsuspected* participants in the assigned role's role set (R'), and all *unsuspected* participants in the role set arguments ($\overline{R}$). We model participants $\widetilde{p_1}$ as synchronized when each has *no* active handler and its subprotocol state designates the spawn event (i.e., its configuration for s_1 has the form $(s_1[p] : (\mathcal{E}_p[L], \_),\ L = g(\overline{q{:}r}; R'; \overline{R})\,.\ L_p)$. For (2), we assign a participant ($p_0 \in R'_{ids} \setminus \bar{q}$) that is *not* already bound to a role in the new subsession. Then $\widetilde{p_2} = \bar{q} \cup p_0 \cup (\overline{R}_{ids} \setminus \mathcal{F})$ is the set of participants joining the subsession; the non-assigned members of R' do not join (unless $R' \in \overline{R}$). We also check that all participants bound to roles are non-suspected ($\widetilde{p_2} \cap \mathcal{F} = \emptyset$). For (3), we update the subprotocol states of each $p \in \widetilde{p_1}$ involved in the spawn (i.e., $s_1[p] : (\mathcal{E}_p[L_p], \_)$). A fresh name s_2 for the new subsession is added

to the monitoring tree and recorded as a child of s_1 ($\Theta' = \Theta[s_2 \mapsto (\widetilde{p}_2, \emptyset), s_1 \mapsto \Theta(s_1) \cup s_2]$), and an initial configuration for s_2 is created for each $p \in \widetilde{p}_2$. The initial subprotocol state L' is given by $g^{\bar{q};p_0}\upharpoonright p$. It simply computes $g \upharpoonright r = g_r$ if p is bound to a role r, else $g \upharpoonright R = g_R$ where R is p's role set; then yields the subprotocol body L' of $g_{r/R}$, with all the role bindings of participants $\bar{q}; p_0$ embedded as $p{:}r$ annotations.

To simplify Spawn without impacting the overall theory, our formalism does not actually *fire* spawn handlers, as the actions described above are driven solely from the subprotocol states (this is why spawn terms are formalised without a continuation). Our prototype (Section 4.7) *does* fire spawn handlers, and implements Spawn as a simple handshake – coordinated by a designated role using its *local* fail set (i.e., our implementation decomposes $\mathcal{F}$ into local views). The rules Sess-GC and Root-GC in Figure 4.12 perform garbage collection of configurations, e.g., $s[p] : (L_p, \_)$, and monitoring tree Θ entries for finished subsessions. Predicate done used in these rules holds if the subprotocol state is terminated (completed, i.e., $L_p = \mathsf{end}$ with $\_$, or $L_p = -$ with end) or stopped (halted due to active failure handling by an ancestor subsession, i.e., L_p is end_L). Sess-GC removes the configurations (of unsuspected participants) and the monitoring tree entry for subsession s_2 when all unsuspected participants $\widetilde{p}$ are finished in s_2. Root-GC is a special case for the root subsession that also removes the corresponding event loops. The garbage collection rules allows a tidier progress statement. The direct practical interpretation of these rules is a subsession teardown synchronization (as we employ in our implementation). As for subsession initiation, more elaborate schemes are possible but outside the main topic, hence we opt for a simpler core formalism.

4.5.3. Failure suspicion and handling

Figure 4.13 gives the rules for failure suspicion and handling. Susp non-deterministically declares that an arbitrary participant is suspected of failure. For broader applicability, we expressly abstract from specific kinds of failure and failure detection mechanisms: this rule reflects the absence of control over failures and *unreliability of failure detection* in real-world distributed systems – being *suspected* has no bearing on whether the participant *continues* execution or actually *stops*.

Rule Mon activates the failure handling at the monitor participant. If p is suspected, Mon can switch the monitor's subprotocol state to the failure handling (i.e., $-$ with L). We write $s \rightsquigarrow^{+}_{\Theta}$ for the set of subsession names that are transitively reachable from the subsession name s in Θ (i.e., all descendant subsessions of s). All configurations of p for these s_i are *stopped* ($s_i[p] : (\mathsf{end}_{L_p}, \_)$), and p will no longer participate in them: a stopped subprotocol state end_L does not match any local type, i.e., $\mathsf{end}_L \asymp L'$ is always false. RcvFN activates failure handling at a *non*-monitor participant, due to receiving

$\boxed{(\Theta, \mathcal{F}, N) \to (\Theta', \mathcal{F}', N')}$

$$\textsc{Susp}\ \frac{p \notin \mathcal{F}}{(\Theta, \mathcal{F}, N) \to (\Theta, \mathcal{F} \cup p, N)}$$

$$\textsc{Mon}\ \frac{q \in \mathcal{F} \qquad s \rightsquigarrow^{+}_{\Theta} \{s_i\}_{i \in I} \qquad L_p = L' \text{ with } q{:}r{\downarrow}.L \qquad L'_p = - \text{ with } L}{(\Theta, \mathcal{F}, s[p][r]{\downarrow}.P \mid\mid s[p] : (L_p, \_) \mid\mid_{i \in I} s_i[p] : (L_i, \_)) \to (\Theta, \mathcal{F}, P \mid\mid s[p] : (L'_p, \_) \mid\mid_{i \in I} s_i[p] : (\mathsf{end}_{L_i}, \_))}$$

$$\textsc{RcvFN}\ \frac{L_1 = L' \text{ with } q{:}r?l.L \qquad (q \mapsto l \cdot \bar{l}) \in b \qquad s \rightsquigarrow^{+}_{\Theta} \{s_i\}_{i \in I} \qquad b' = b[q \mapsto \bar{l}]}{\begin{array}{c}(\Theta, \mathcal{F}, s[p][r]?l.P \mid\mid s[p] : (L_1, b) \mid\mid_{i \in I} s_i[p] : (L_i, \_)) \to \\ (\Theta, \mathcal{F}, P \mid\mid s[p] : (- \text{ with } L, b') \mid\mid_{i \in I} s_i[p] : (\mathsf{end}_{L_i}, \_))\end{array}}$$

$$\textsc{Clean}\ \frac{b(q) = l_i \cdot \bar{l} \qquad \big((L = - \text{ with } L' \wedge l_i \notin L') \vee (L = L' \text{ with } L'' \wedge l_i \in L' \wedge \exists l' \in \bar{l}.\ l' \in L'')\big)}{\big((\overline{H}, p) \mid\mid s[p] : (L, b)\big) \to \big((\overline{H}, p) \mid\mid s[p] : (L, b[q \mapsto \bar{l}])\big)}$$

Figure 4.13.: Failure suspicion and failure handling activation.

a *failure notification*, i.e., a message communicated from some other failure handling activity to *convey* the suspected failure; it is similar to Mon. Lastly, Clean models that participants *prioritize* failure handling over the normal activity. This rule simply discards messages associated with the *normal* activity from the front of an input FIFO if p "knows" of a suspected failure, either because the failure handling is already active (the left side of the $\vee$), or a failure notification has arrived in the FIFO (the right side). Prioritizing failure handling is crucial in asynchronous multiparty systems, where there is no causality between the arrival of normal messages and failure notifications; e.g., to avoid being stuck on a blocking action in the normal activity that can no longer be discharged. Recall that label sets between a normal and failure activity are disjoint, providing the intuitive condition for safe failure notifications.

Example 22. *Assume $(\mathcal{F}_0, \Theta', (\nu s : \mathcal{G}_{CM})\, N_2)$ from Example 21. We illustrate the reductions for the failure scenario in Figure 4.9 where p_2 playing w_D is suspected of failure. In Figure 4.9 (c), monitor p_1 activates failure handling in s, stops its activity in s', and sends l_{FailDr} notifications to p_3 and p_4. Let $K = \{1, 3, 4\}$ (i.e., all unsuspected p's) and $J = \{3, 4\}$ (the unsuspected workers).*

$$(\mathcal{F}_0, \Theta', (\nu s : \mathcal{G}_{CM})\, N_2) \xrightarrow{\textsc{Susp}} (\{p_2\}, \Theta', (\nu s : \mathcal{G}_{CM})\, N_2) \xrightarrow{\textsc{Fire},\textsc{Mon},\textsc{MSnd}} (\{p_2\}, \Theta', (\nu s : \mathcal{G}_{CM})\, N_3)$$

$$\begin{aligned} N_3 \ = & \mid\mid_{k \in K} (\overline{H}_k, p_k) \\ & \mid\mid s[p_1] : (- \text{ with } g_{RunDr}(p_1{:}m; W; W), b_1) \mid\mid s'[p_1] : (\mathsf{end}_{L'_1}, b'_1) \mid\mid_{j \in j} s'[p_j] : (L'_j, b'_j) \\ & \mid\mid_{j \in J} s[p_j] : (\ldots \text{ with } p_1{:}m?l_{FailDr} \,.\, g_{RunDr}(p_1{:}m; W; W), b_j[p_1 \mapsto b_j(p_1) \cdot l_{FailDr}]) \mid\mid \ldots \end{aligned}$$

Susp sets p_2 as suspected, i.e., it adds p_2 to $\mathcal{F}_0$. Then Fire fires the handler of p_1 for the w_D suspicion event, i.e., it fires $[w_D{\downarrow}.\, W!l_{FailDr}]\lambda x.\, x[w_D]{\downarrow}.\, x[W]!l_{FailDr}$. We then execute

this handler. First, Mon *sets the subprotocol state of* p_1 *in* s *to* active *failure handling and removes the suspicion type prefix, i.e.,* $-$ with $W!l_{FailwD}.\, g_{RunDr}(p_1{:}m; W; W)$*; and it sets the subprotocol states of* p_1 *in all descendent subsessions of* s *(i.e.,* s'*) to* stopped *(i.e.,* $\mathsf{end}_{L'_1}$*). Next,* MSnd *multisends* l_{FailDr} *to all* unsuspected *workers, i.e.,* p_3 *and* p_4*. Figure 4.9 (c) thus depicts* $(\{p_2\}, \Theta', (\nu s : \mathcal{G}_{CM})\ N_3)$*.*

In Figure 4.9 (d), p_3 *and* p_4 *first each activate their failure handling in* s*.*

$$(\{p_2\}, \Theta', (\nu s : \mathcal{G}_{CM})\ N_3) \xrightarrow{\textsc{Fire},\textsc{Fire},\textsc{RcvFN},\textsc{RcvFN}} (\{p_2\}, \Theta', (\nu s : \mathcal{G}_{CM})\ N_4)$$

$$N_4 = ||_{k\in K}(\overline{H}_k, p_k)||\ \ s[p_1] : (-\text{ with } g_{RunDr}(p_1{:}m; W; W), b_1) \ ||\ s'[p_1] : (\mathsf{end}_{L'_1}, b'_j)$$
$$||_{j\in J}\ s[p_j] : (-\text{ with } g_{RunDr}(p_1{:}m; W; W), b_j)\ ||_{j\in j}\ s'[p_j] : (\mathsf{end}_{L'_j}, b'_j)\ ||\ \ldots$$

p_3 *and* p_4 *each fire their handler on receiving the failure notification* l_{FailDr} *in* s*. Each then reduce their active handler using* RcvFN*: they activate failure handling in their subprotocol states of* s*, remove the branch prefix from the failure activity (i.e., consume the* l_{FailDr}*), and set all descendent subprotocol states to stopped (*$\mathsf{end}_{L'_j}$*). The above assumes* $b_j(p_1) = \varepsilon$*; otherwise,* Clean *would first discard the (now obsolete) normal activity labels from* p_1 *in* p_j*'s buffer.*

Lastly, p_1*,* p_3 *and* p_4 *restart* g_{RunDr}*. We illustrate the* Spawn *step by spawning a new subsession* s''*. This time, it happens that* p_3 *is assigned to play* w_D*. Figure 4.9 (d) depicts* $(\{p_2\}, \Theta'', (\nu s : \mathcal{G}_{CM}) N_5)$ *with* $\Theta'' = \Theta'[s'' \mapsto \{p_1, p_3, p_4\}]$*, where:* $(\{p_2\}, \Theta', (\nu s : \mathcal{G}_{CM}) N_4) \xrightarrow{\textsc{Spawn}} (\{p_2\}, \Theta'', (\nu s : \mathcal{G}_{CM})\ N_5)$

$$N_5\ = ||_{k\in K}((\overline{H}_k, p_k)\ ||\ s[p_k] : (-\text{ with end}, b_k)\ ||\ s'[p_k] : (\mathsf{end}_{L'_k}, b'_k)\ ||\ s''[p_k] : (\ldots, \varepsilon))\ ||\ \ldots$$

4.6. Type System and Properties

We now define the typing rules and show that well-typed systems enjoy *subject reduction*, *session progress*, and *global progress*.

4.6.1. Typing rules

The typing judgments used by our system are of the shapes $\Gamma, \Sigma \vdash N \triangleright \Delta$ and $\vdash (\Theta, \mathcal{F}, (\nu s : \mathcal{G})\, N)$. The latter is for the *top-level system*, i.e., a network under the root session restriction; the former is for all other rules for processes and networks. *Standard environment* Γ maps the root session name to a top-level global type $(s : \mathcal{G})$. *Configuration environment* Σ maps channel values to configuration types $(s[p] : (L, b))$, and participant names to role sets and handler types $(p{:}R : \{L_i\}_{i\in I})$. *Session environment* Δ maps channels to *guard* types $(c : L)$. We write: Δ_{end} for an end-only session environment (i.e., $\Delta_{\mathsf{end}}(c) = \mathsf{end}$, for all c in Δ_{end}); $\Sigma_1 \cdot \Sigma_2$ for the union of two configuration environments with disjoint domains;

$\boxed{\Gamma, \Sigma \vdash N \ \triangleright\ \Delta}$

$$\textsc{TELoop}\ \frac{\forall i \in 1..n\,.\,(H_i = [L_i]\lambda x_i.\,P_i \ \wedge\ s{:}\mathcal{G}, \emptyset \vdash H_i \ \triangleright\ \emptyset) \quad p \in R_{ids} \quad \forall g_z(...) = L \in (\mathcal{G} \upharpoonright R)\,.\,\{L_i\}_{i\in 1..n} \vdash L}{s{:}\mathcal{G}, p{:}R : \{L_i\}_{i\in 1..n} \vdash (H_1 \cdot ... \cdot H_n, p) \ \triangleright\ \Delta_{\mathsf{end}}}$$

$$\textsc{THandler}\ \frac{\Gamma, \emptyset \vdash P \ \triangleright\ x : L \quad \mathsf{wf}(L)}{\Gamma, \emptyset \vdash [L]\lambda x.\,P \ \triangleright\ \emptyset}$$

$$\textsc{TSnd}\ \frac{\Gamma, \Sigma \vdash P \ \triangleright\ c : L, \Delta_{\mathsf{end}}}{\Gamma, \Sigma \vdash c[r]!l.P \ \triangleright\ c : r!l.L, \Delta_{\mathsf{end}}} \qquad \textsc{TRcv}\ \frac{\Gamma, \Sigma \vdash P \ \triangleright\ c : L, \Delta_{\mathsf{end}}}{\Gamma, \Sigma \vdash c[r]?l.P \ \triangleright\ c : r?l.L, \Delta_{\mathsf{end}}}$$

$$\textsc{TMultSnd}\ \frac{\Gamma, \Sigma \vdash P \ \triangleright\ c : L, \Delta_{\mathsf{end}}}{\Gamma, \Sigma \vdash c[R]!l.P \ \triangleright\ c : R!l.L, \Delta_{\mathsf{end}}} \qquad \textsc{TEnd}\ \frac{}{\Gamma, \emptyset \vdash \mathsf{loop} \ \triangleright\ \Delta_{\mathsf{end}}}$$

$$\textsc{TMon}\ \frac{\Gamma, \Sigma \vdash P \ \triangleright\ c : L, \Delta_{\mathsf{end}}}{\Gamma, \Sigma \vdash c[r]{\downarrow}.P \ \triangleright\ c : r{\downarrow}.L, \Delta_{\mathsf{end}}} \qquad \textsc{TSpawn}\ \frac{s : \mathcal{G} \in \Gamma \quad g(\bar{r}; R; \overline{R}) = ... \in \mathcal{G}}{\Gamma, \Sigma \vdash c[\bar{r}; R; \overline{R}](g) \ \triangleright\ c : g(\bar{r}; R; \overline{R}).\mathsf{end}}$$

$$\textsc{TCfg}\ \Gamma, s[p] : (L, b) \vdash s[p] : (L, b) \ \triangleright\ \Delta_{\mathsf{end}} \qquad \textsc{TEnd}\ \Gamma, \emptyset \vdash 0 \ \triangleright\ \Delta_{\mathsf{end}}$$

$$\textsc{TPar}\ \frac{\Gamma, \Sigma_1 \vdash N_1 \ \triangleright\ \Delta_1 \quad \Gamma, \Sigma_2 \vdash N_2 \ \triangleright\ \Delta_2}{\Gamma, \Sigma_1 \cdot \Sigma_2 \vdash N_1 \,||\, N_2 \ \triangleright\ \Delta_1 \cdot \Delta_2}$$

$\boxed{\vdash (\Theta, \mathcal{F}, (\nu s : \mathcal{G})\ N)}$

$$\textsc{TSystem}\ \frac{\mathsf{wf}(\mathcal{G}) \quad s{:}\mathcal{G}, \Sigma \vdash\ N \ \triangleright\ \Delta \quad (\Delta, \Sigma, \mathcal{F}, \Theta)\ \text{coherent}}{\vdash (\Theta, \mathcal{F}, (\nu s : \mathcal{G})\ N)}$$

Figure 4.14.: Typing rules: (top) event loops, static/active handlers, and networks; (bottom) the top-level system.

and $\Delta_1 \cdot \Delta_2$ for the union of Δ_1 and Δ_2, *provided* for $\{i, j\} = \{1, 2\}$, $c{:}L \in \Delta_i$ implies $(c \notin \Delta_j \vee \Delta_j(c) = \mathsf{end})$. Basically, $\Delta_1 \cdot \Delta_2$ is the union if no c is in both Δ_i, unless one/both map to end; else, it is *undefined*. An append $c{:}L, \Delta$ is defined if c is not in Δ; analogously for Γ and Σ.

The top three rows in Figure 4.14 give the rules for *event loops* and *handlers*. TELoop types the event loop for participant p of role set R under $\Gamma = s{:}\mathcal{G}$. It checks each handler is well-typed, and the set of guard types together provides sufficient *coverage* of all subprotocols projected from $\mathcal{G}$ onto R, i.e., $\forall g_z(...) = L \in (\mathcal{G} \upharpoonright R).\{L_i\}_{i\in 1..n} \vdash L$. Intuitively, the event loop must cover every possible event in the overall protocol that may involve p. The coverage judgment $\{L_i\}_{i\in 1..n} \vdash L$ simply asserts that (1) at least one guard type (and thus handler) on the LHS can be matched to L's normal activity, (2) similarly for the failure handling activity, and (3) recursively so for the remaining unmatched portions of each activity, if any. The event loop is then typed with its set of guard types $(p{:}R : \{L_i\}_{i\in I})$

as Σ, and Δ_{end} (the event loop becomes the terminal term in an active handler process, cf. FIRE). We leave the full definition of coverage to Appendix B.6.

Example 23 (Coverage)**.** *Consider* $\{L_i\}_{i\in I} \vdash L$ with L' *where* $L = p{:}a?l_1.q{:}b!l_2$. Coverage *requires that some prefix of* L *matches a guard type (i.e., handler) in* $\{L_i\}_{i\in I}$*, say,* $L_1 = a?l_1$. $\{L_i\}_{i\in I}$ *is then required to provide coverage of the remaining portion of* L*, i.e.,* $b!l_2$*; and similarly for* L'.

THANDLER types a handler under an empty Σ and empty Δ – it checks that the handler *body* uses the channel argument fully according to the guard type. As discussed in Section 4.4.3, it restricts the guard type to a valid handler behavior: we write $\mathsf{wf}(L)$ to simply assert that L is (1) *flat* (all selections/branchings must be *unary* sends/receives), (2) only the *initial* action is potentially blocking (receive, spawn or suspicion), and (3) does *not* contain recursion. TSND TMULTSND, TRCV and TMON are for both static handler code (under THANDLER) and active handler processes (under TPAR). They type a send, multisend, receive, and monitoring action on c, respectively, if the continuation is typeable. TSPAWN types a spawn action on c with the matching spawn type if the top-level $\mathcal{G}$ contains the subprotocol definition. TEND types loop (end of a handler body) by Δ_{end} and $\Sigma = \emptyset$.

The fourth row in Figure 4.14 gives the other rules for networks. TCFG types a configuration. TPAR types a parallel composition of two typeable networks if their Σ_i (resp. Δ_i), $i \in \{1, 2\}$, have disjoint domains. TEND types the terminated network 0. Lastly, TSYSTEM types the top-level system (i.e., root session restriction) together with the reduction environments Θ and $\mathcal{F}$. It checks that the top-level global type is well-formed; network N is typeable under the corresponding $\Gamma = s : \mathcal{G}$, some Σ, and some Δ; and that the tuple $(\Delta, \Sigma, \mathcal{F}, \Theta)$ is *coherent*, discussed next.

Coherence. *Coherence* (or *consistency*) is the central typing invariant in MPSTs that ensures participant interactions remain safe throughout reduction. It is used as an invariant property of runtime networks to establish subject reduction. In standard MPSTs (e.g., [CDYP16]), coherence is based on pairwise *duality* of endpoint types: duality is the intuitive compatibility relation between two participants, where any action (e.g., output) on one side is safely balanced by a corresponding action (e.g., input) on the other. There are two parts to our approach for the present framework. First, we extend the standard coherence as *intrasession* coherence, i.e., for individual (sub)sessions, to cater to our event-driven model and failure handling. Second, we introduce a notion of *intersession* coherence – an invariant property *across* subsessions to ensure their concurrent execution and cross-session failure handling remain safe. It ensures: (1) a subsession only involves participants from its parent subsession; and (2) for unsuspected participants, every stopped configuration has a non-stopped ancestor configuration with an active failure handling, and all descendent configurations of a configuration with an active failure handling

are stopped. The exact definition of intrasession and intersession coherence is a detail for understanding the proofs of our properties (Section 4.6.2), for which the complete technical development is available in the supplement (e.g., Appendix B.6.2).

Definition 21 (Coherence)**.** *Assuming a well-formed top-level $\mathcal{G}$, the tuple $(\Delta, \Sigma, \mathcal{F}, \Theta)$, used to type a system, satisfies* coherence *if (i) it satisfies intrasession coherence for all $s \in \Theta$, and (ii) it satisfies intersession coherence. We may simply say $(\Delta, \Sigma, \mathcal{F}, \Theta)$ is* coherent.

4.6.2. Properties

We can now present the key properties of our system. See the supplement (Appendix B.6) for the more details and proofs. We note the previously discussed assumptions of our system, namely that (1) participants playing the role(s) declared in the *root* subsession are robust, and (2) the runtime infrastructure provides sufficient participants for role assignments during execution. In failure-sensitive distributed systems (1) is commonly achieved by making use of a so-called "application master"; this component is typically made fault-tolerant by managing its critical state with a fault-tolerant coordination service, e.g., ZooKeeper [Hun10], and (2) is a matter of runtime availability of resources.

Subject reduction states that a well-typed system remains well-typed after any reduction step.

Theorem 3 (Subject Reduction)**.** *Let $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G}) N_1)$ such that $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G}) N_1) \rightarrow (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$. Then $\vdash (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$.*

Note that a well-typed system is coherent by TSystem. The proof proceeds as follows. First, as in Honda et al. [HYC16] and Coppo et al. [CDYP16], we define a *typing environment reduction* $((\Delta,\ \Sigma,\ \mathcal{F},\ \Theta) \rightarrow (\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta'))$ that describes how the environments are updated in relation to asynchronous network reduction. In other words, it mimics the interaction dynamics of network reduction at the *local type* level. The proof is by enumerating over all cases of typing environment reduction. Second, we prove *type preservation*: a well-typed network $(\Gamma, \Sigma \vdash N_1 \ \triangleright\ \Delta)$ remains well-typed after a reduction step to N_2. in some cases the proof requires inductively splitting a network into parts that are not necessarily coherent. Subject reduction follows from preservation of coherence and type preservation.

The key session *communication safety* property in our system, including *safe failure handling*, follows as a corollary. For a given session s, we say an unsuspected p: (i) has a *reception error* if it has a receive subprotocol state but the queued label is not accepted by that receive and Clean does not apply; (ii) is *stuck* if it is blocked on a receive or spawn due to waiting for another unsuspected p', and vice versa; and (iii) has a *non-covered*

failure if it is blocked on a receive or spawn due to waiting for a suspected p', and no failure handling in the entire monitoring tree applies.

Corollary 3.1 (Communication Safety)**.** *Let* $\vdash (\Theta, \mathcal{F}, (\nu s : \mathcal{G})\ N))$*. For every session* s *in* Θ *and unsuspected* p *in* s*,* p *has the following properties: (i)* p *does not have a* reception error*; (ii)* p *is not* stuck*; and (iii)* p *does not have a* non-covered failure*.*

Communication safety is a direct consequence of the coherence invariant.

The other key property of our system is *global progress*: progress for an entire (well-typed) top-level system of *multiple, concurrent subsessions* in the presence of concurrent failures. There are two supporting properties. First, our approach adapts that of Coppo et al. [CDYP16]: we define (a) *global type reduction*, which models interaction dynamics at the *global type* level, and (b) a *mirror* relation that relates a global type to typing environments and their configuration types via projection. Fidelity then states that a subsession s in a (coherently) well-typed network reduces *in tandem* with its corresponding global type G, and that the G' at each step mirrors the configuration types of s. Second, we prove *subsession progress* for every *individual* subsession that is "live": i.e., the subsession is not terminated, has no stopped participant (i.e., no participant with a stopped configuration), and has no outer failure (i.e., no failed participant whose failure is handled in an *ancestor* subsession – its role was assigned by the ancestor subsession). Our subsession progress is a generalization of the standard progress property of MPSTs that is limited to a single session and does not consider any notion of failure (cf. Section 4.3.3).

Theorem 4 (Subsession Progress)**.** *Assume an initial system* $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1)$ *and* $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1) \rightarrow^* (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$*. Let* s' *in* Θ_2 *such that* s' *is* not terminated*, has* no stopped participant*, and has* no outer failure*. Then* $(\Theta_2, \mathcal{F}_2, N_2') \rightarrow (\Theta_3, \mathcal{F}_3, N_3)$ *via a reduction in session* s'*, with either* $N_2' = N_2$ *or* $(\Theta_2, \mathcal{F}_2, N_2) \rightarrow^* (\Theta_2, \mathcal{F}_2, N_2')$*.*

The proof builds on subject reduction and fidelity by adapting the approach of Coppo et al. [CDYP16]. By fidelity, for any "live" subsession s', we have a global type G that has reduced in tandem with s'. Let p be any participant active in the first action described in G (i.e., in its active failure handling if applicable, else its normal activity). We show that this first action ensures a reduction step in s' is available. The proof is by enumerating over all possible shapes of the first action, supported by the following. (1) By fidelity, the configuration type of p contains this first action in their subprotocol state, and in the case of a receive action the label is present in the queue. (2) If p does not have an active handler for s', then coverage ensures that p's event loop has a handler with a matching guard type for its subprotocol state (i.e., a handler can be fired for the pending event). (3) An active event handler (for *any* subsession s'') can always reduce back to the event loop

(this corresponds to the $(\Theta_2, \mathcal{F}_2, N_2) \rightarrow^* (\Theta_2, \mathcal{F}_2, N_2')$ clause in the theorem). All together, these ensure that p can always perform a reduction step for this first action.

Global progress states that a system can always progress unless it has *completely* terminated.

Theorem 5 (Global Progress)**.** *Assume an initial system* $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1)$ *and a reduction* $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1) \rightarrow^* (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$*. Then either* Θ_2 *is empty, or without using* Susp *we have* $(\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2) \rightarrow (\Theta_3, \mathcal{F}_3, (\nu s : \mathcal{G})\ N_3)$*.*

A completely terminated system is signified by an empty monitoring tree (i.e., when Θ_2 is empty), implying that every subsession was terminated or stopped, and the configuration terms and monitoring tree entries have been removed via Sess-GC and Root-GC. The proof idea is as follows. Event-driven concurrency is crucial to global progress in our framework: despite concurrent subsessions, progress in any given subsession is *independent* of all other subsessions (i.e., actions in one subsession are never blocked by those in another). We thus consider systems where *every* subsession is not "live" (otherwise subsession progress would apply to one) – in such cases, however, we show that some other reduction step is available, e.g., one of the garbage collection rules. Here, intersession coherence ensures every subsession is safely covered by a failure handling in some ancestor subsession. The side condition regarding Susp ensures that progress does not *abuse* failure suspicion to bypass *actually* stuck sessions.

4.7. Evaluation

Example applications. Section 4.2–4.3 introduced our session-typed CM, which demonstrates how all the various features of our system are needed in a practical fault-tolerant protocol. The table below summarizes further examples from MPST literature that we have specified and implemented (with added failure handling) using our Scala toolchain to demonstrate its expressiveness.

(1) *Core MPSTs (MP interactions, choice, recursion)*
2-Buyers, Streaming [HYC16]
Sutherland-Hodgman [NHYA18]
(2) *Dynamic/parameterized participants*
3-Buyers [CDYP16]
N-stage Pipe [CHJ+19]
N-stage Ring [CHJ+19]

(3) *Application-level exceptions/interrupts*
Two Factor [FLMD19]
Resource Control [DHH+15]
WebCrawler [NY17]
Interruptible 3-Buyers [CGY16]
Basic failure handling (cf. Figure 4.16)
Failure-Aware Streaming [VCE+18]

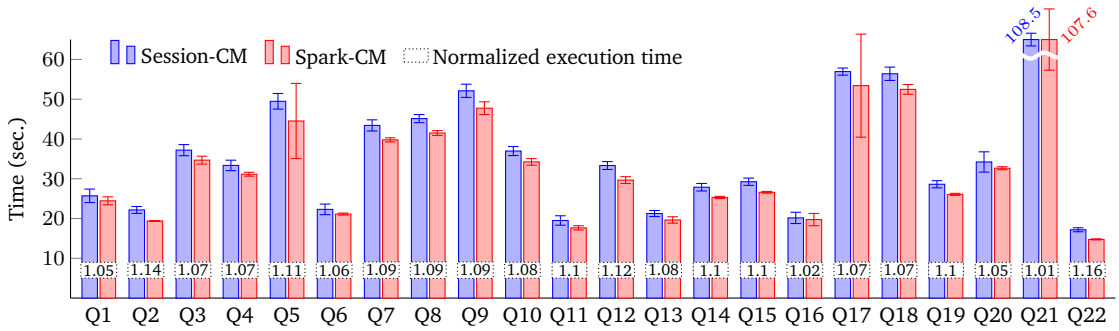

Figure 4.15.: Comparing mean execution times of our Session-CM to the Spark-CM on TPC-H queries. The numbers on the bars give the execution time of our Session-CM normalized to the Spark-CM.

The main takeaways, corresponding to parts (1)–(3), are that our system: (1) subsumes the core communication constructs of standard MPSTs; (2) by necessity supports communication patterns previously limited to relatively exotic MPST features; and (3) can express MPST-based application-level exception and interrupt patterns.

Runtime performance (Spark use case). To demonstrate the practicality of our framework, we compare the performance of the *full* version of our Session-CM (Appendix B.7), running over our session runtime, to Spark-CM. Our Session-CM is not just a "toy" – it is compatible with the other Spark Core components and supports the execution of existing third-party Spark applications *without* any code modification, enabling this experiment. Note, we only reimplement the CM, so our session runtime conducts only the Session-CM internal messaging, not the communications of the Spark application running on top.

The aim is to measure the overheads incurred by our toolchain and session runtime prototype implementation. Our theory introduces additional mechanisms, i.e., potential sources of overhead, mainly: (i) the CFSM used to track the session protocol state and selection of event handlers at runtime, (ii) dynamic checking of linear channel usage, and (iii) infrastructure used to implement subsessions (e.g., additional message queues). We note our prototype is not optimized in general (e.g., some abstraction layers around our message queues could be eliminated), whereas Spark's CM is a mature, industrial-strength component.

We use a Spark implementation[5] of the industry-standard TPC-H benchmark as the benchmark application. TPC-H specifies a set of 22 complex queries on a database of 8

[5]TPC-H queries implemented in Spark, `https://github.com/ssavvides/tpch-spark`.

ST & MPST works	aMP	CSFH	FS	SubS	Par	DP	RR	EDP
This chapter	✓	✓	✓	✓	✓	✓	✓	✓
[APN17]	*sync.*	*mask.*		✓		*ext.*		
[VCE+18]	✓	✓						
[FLMD19]	*binary*							
[DHH+15]	✓	*app.*						
[CGY16]	*sync.*	*app.*						
[CHY08]		*app.*						
[CAP17]								*react.*
[DH12]	*sync.*			✓		*ext.*		
[CHJ+19]	✓				✓			
[HKP+10]	*binary*							✓

aMP Asynchronous MPSTs
CSFH Partial crash-stop failure handling
FS False suspicions of failures
SubS Subsessions
Par Participant parameterization
DA Dynamic role assignment
RR Failed role replacement
EDP Event-driven programming

Figure 4.16.: Comparison to related works; Section 4.8 clarifies where entries are neither "yes" (✓) nor "no" (blank).

tables; we use a database scaling factor of 10 (i.e., database size ∼10GB). Each benchmark run measures the total time (including application startup time) to execute *one* query as an independent application using (a) our Session-CM and session runtime, and (b) the Spark-CM, for scheduling; all other factors are the same. In both cases each Spark application (query) has three servers with identical hardware (Intel Xeon E-2278G CPUs, 64GB RAM) and running Ubuntu 18.04.3 LTS.

Figure 4.15 reports the average (arithmetic mean) of the execution times (y-axis) and standard deviations (error bars) of 10 runs per query. The plot further states the normalized execution time of the Session-CM for each query, i.e., the result of dividing the average execution time of the Session-CM by the average execution time of the Spark-CM. Across all 22 queries, our Session-CM exhibits an average overhead below 10% (and a maximum below 16.5%) compared to Spark-CM. We repeated the experiment allocating *two* servers to a Spark application, where a server running an executor fails after 20s (by killing it) and is successfully replaced, for a query we picked at random (Q18). The average execution times of 5 runs are: 99.19s for Spark-CM (std. dev. of 2.23s), and 109.4s for our Session-CM (std. dev. of 0.44s), i.e., an overhead of $\sim 10\%$.

4.8. Related Work

Failure handling in MPSTs. In session types literature, the two main related works on failure handling in MPSTs, both without performance evaluation, are by Adameit et al.

[APN17] and Viering et al. [VCE+18][6]. Adameit et al. [APN17] extend MPSTs by wrapping interactions in *optional blocks* to model communication *link* failures. They require default values for input actions that may thusly fail, giving a form of failure *masking* (*mask.* in the **CSFH** column of Figure 4.16) where the protocol specification outside of an optional block is agnostic to whether a failure occurs or not. When execution discards the block of some role due to failure, peer roles must also discard their corresponding blocks: such agreement in the presence of failures is only possible with *synchronous* channels (*sync.* in the **aMP** column) or – boiling down to the same – *perfect* failure detection (no false suspicions). This limits applicability to real-world distributed systems that are asynchronous. Their approach includes nested protocols [DH12] for *external* participant invitations during a nested session call (*ext.* in the **DA** column), offering a form of dynamic role assignment. The authors argue that the main example, a rotating coordinator algorithm, can be considered as a distributed asynchronous process; because of its structure, e.g., it uses no choice and outputs have no continuation different than 0 [APN16].

Viering et al. [VCE+18] (and Chapter 3) extend MPSTs with *try-catch* handling for process crash failures. They closely follow traditional MPSTs by directly preserving try-catch structures across global types, local types, and "multithreaded" session π-calculus processes. Their approach works by coupling the reduction (i.e., control flow) of the try-catch construct at the process level to a specific mechanism for detecting distributed process failures based on a *fail-safe* coordinator, e.g., ZooKeeper [Hun10]. Their system assumes *perfect* failure detection. By contrast, our model makes no assumptions on the accuracy [CT96] of failure detection (our communication safety and progress are ensured despite continued actions by falsely suspected peers), and monitors are peers that may be failure-prone themselves. Our framework thus applies to concrete systems based on (e.g.) ZooKeeper, but is also not coupled to any specific oracle infrastructure.

Binary sessions, exceptions, and events. Fowler et al. [FLMD19] add an exception handling process primitive for failures in *binary* sessions. *Type*-level treatment of failure handling behaviors between the remaining (and new) peers, as in this chapter, cannot be studied in a binary setting (*binary* in the **aMP**/**CSFH** columns). Besides the above, the existing session types literature (e.g., [CHY08, CGY16, DHH+15]) has only addressed *application-level* failures (app. in the **CSFH** column). These works consider "exceptional" behaviors in the sense of application logic (e.g., an inappropriate payload value), as opposed to actual participant process failures – the former deal with exception-like protocol control flow over a *normally* functioning session between *fixed* participants, while the latter necessitates reasoning about sessions with *changing* participants due to *failures* and replacements. Our work tackles the latter by modeling protocols with explicit

[6] We provide a performance evaluation for our previous work [VCE+18] in Chapter 3.

specification of participant failure detection/notification, asynchronous failure handling behaviors, and retrying failed interactions with dynamically replaced participants. *None of the works cited above* study fault-tolerance in this regard nor event-driven concurrency for MPSTs.

Hu et al. [HKP+10] present a *binary* session calculus for encoding EDP patterns. They add a non-blocking primitive for *polling* channels for messages and a session typecase construct based on dynamic typing [ACPP91], while we model event loops and handlers as first-class concepts. Their work does not consider failures (as mentioned, specification of fault-tolerant, multiparty communication patterns cannot be studied in a binary setting), and does not consider the challenges that we address for MPSTs: (partial) projections, coherence, or fidelity. Cano et al. [CAP17] present a related binary calculus (*without* types) for reactive sessions (*react.* in **EDP**). Coppo et al. [CDYP16] develop an *additional* interaction type system on top of MPSTs to analyze global progress; our system offers global progress for concurrent (sub)sessions within MPSTs. Our notions of subprotocols and subsessions are inspired by the nested protocols of Demangeon and Honda [DH12]. We exploit subsessions to (1) incorporate lightweight practical participant parametericity and dynamic role assignment, and (2) reason about the runtime structure of failure monitoring between subsessions. Their work does not consider failures, and their progress property is restricted to a single session.

Outside of session types. This chapter (and more broadly MPSTs) promotes a methodology for safe development of message passing applications based on top-down protocol specification and distributed endpoint implementation. There exists a wide range of techniques and tools that focus more, in comparison to this chapter, on *verification* than software *development*, in the context of distributed systems and algorithms with failure handling. They can be considered on a scale from mostly "manual" to automated techniques. In this regard, the verification aspect of our toolchain can be considered as fully automated (for our specific MPST-based properties) after the user supplies the protocol specification as a global type.

At the former end of the scale are verification approaches based on interactive proof tools such as Verdi [WWP+15] and IronFleet [HHK+15]. Recent approaches [SWT18, TLM+18, vGKB+19] employ modularization of proofs or transformation of asynchronous programs into synchronous versions to reduce the user effort. These approaches allow establishing user-defined functional correctness properties for distributed systems, but can require substantial poof engineering effort. The formal specifications written by the user can become quite complex [CCE+21].

Semi-automated verification approaches like PSync [DHZ16] and Ivy [PMP+16] can automate parts of the verification by restricting the underlying model. PSync is a domain-

specific language based on the *heard-of* model [CBS09] and structures programs into sequentially executed rounds. Ivy verifies safety properties for algorithms; it introduces a modeling language based on EPR logic [PdMB10] for which the checking of invariants is decidable. It interactively guides a user via counterexamples to a valid invariant.

At the other end of the scale are fully automated verification techniques that trade off the supported class of properties and algorithms/systems, and perform full or partial verification. Model checking tools can transparently check distributed system implementations [YCW+09], be combined with a tailored programming language [KAB+07], and automatically verify restricted classes of distributed algorithms [KLVW17, MGJ+19]. Applications of these tools to concrete implementations of distributed systems may be best effort (i.e., incomplete).

4.9. Conclusion, Limitations and Future Work

We presented an MPST framework for practical fault-tolerant distributed programming. We formalized our approach, proving communication safety and global progress, and demonstrated its practicality by implementing and evaluating a session-typed CM for Apache Spark.

Our present formulation inherits some of the standard conservative restrictions of MPSTs. For example, our global type branch is a *directed choice*, meaning the immediate action in every choice case is between the same two roles/role sets (e.g., $m \to W$). We believe it is possible to port techniques for relaxing such restrictions from recent works into our setting, e.g., by augmenting global type expressiveness using MPST-based behavioral type properties [SY19], or by incorporating explicit connection actions [HY17] into subsession initiation.

As future work, we are considering further reimplementation studies of components from related middleware systems in the Apache middleware family (e.g., Storm, Fink, Kafka). We are also considering further performance evaluations, e.g., scalability in relation to data set size and number of worker processes, as part of improving our toolchain and runtime implementation.

5. Conclusion and Future Work

This thesis introduces two formal models, based on MPSTs, for verified partial failure handling, supporting the specification and verification of protocols that deal with crash-stop failures. This demonstrates that behavior type disciplines, based on MPSTs, can statically verify relevant properties of distributed applications that deal with partial failures. We now conclude the thesis and then discuss some interesting avenues for further research.

5.1. Conclusion

This thesis extends MPSTs to support the specification and verification of fault-tolerant distributed applications - applications that are challenging to implement correctly and where lightweight verification could be highly beneficial to developers. Our type systems ensure relevant properties such as progress and the absence of communication errors. We show that our two typing disciplines can specify and verify protocols and implementations that handle partial failures. We provide toolchains for both frameworks that enable lightweight verification of failure-aware distributed applications. Moreover, we show that our implementations provide good runtime performance. This thesis demonstrates that type systems, based on MPSTs, can verify relevant properties of failure-aware distributed applications and provide developers of distributed applications with lightweight verification tools. We now draw a more detailed conclusion for each model separately.

Lightweight coordinator-based partial failure handling. Our first model, which we present in Chapter 3, adds typing support for crash-stop failure handling in a lightweight coordinator model – a common model of many real-life systems. The model carefully exposes potential problems in distributed applications due to partial failures, such as inconsistent endpoint behaviors and orphan messages. Furthermore, to ensure that our model is practical, it involves the coordination service sparingly because interactions with the coordination service are more expensive than regular messages. Our typing discipline addresses these challenges by building on the mechanisms of MPSTs, e.g., well-formedness of global types for sound failure handling specifications, modeling of

asynchronous permutations between regular messages and failure notifications in sessions, and the type-directed mechanisms for determining correct and orphaned messages in the event of failures. We adapt the coherence invariant of session typing environments (i.e., endpoint consistency) to consider failed roles and orphan messages. The type system statically ensures subject reduction in the presence of failures. Informally, even under concurrent failures, we statically ensure that distributed applications remain safe.

We provide a prototype implementation for our model, featuring Apache ZooKeeper as its coordination service. The evaluation shows that a session-type LR model has a runtime performance comparable to failure agnostic baselines in non-failure cases, thus validating the design goal of minimizing the involvement of the coordination service outside of failure handling.

Practical fault-tolerant protocols. Our second model enables the specification and verification of practical fault-tolerant protocols in MPSTs. It supports partial failures due to crash-stop failures, even under non-perfect failure detection, and provides language features to specify, implement, and verify practical asynchronous fault-tolerant distributed applications. It is the first work offering an event-driven programming model for MPSTs.

As an MPST theory, the event-driven model facilitates a tractable integration of a host of features needed to express fault-tolerant communication patterns that can replace failed processes and restart failed protocol segments. We show that our type system statically ensures subject reduction, protocol fidelity, progress, and global progress for concurrent sessions in the presence of failures and their handling. In other words, even under concurrent failures, we statically ensure that a distributed application with advanced concepts, such as replacing failed participants will progress safely. It is worth highlighting that traditionally MPST systems do not provide global progress for interleaved sessions, even in a setting without failures. We exploit the EDP characteristics to develop our global progress property.

We demonstrate the practicality of our approach by implementing a prototype of our framework and realizing the Session-CM, which can schedule real Apache Spark applications. Apache Spark has an average overhead of around 10% in both the non-failure and failure evaluation on an industry-standard benchmark using the Session-CM instead of Spark's default CM.

5.2. Future Work

We now briefly discuss some interesting directions for future work.

Expressiveness of global types. Our models build on traditional MPSTs [HYC16, CDYP16] and inherit their restrictions on well-formed global types. Some recent works increase the expression of MPST protocols, allowing, in particular, for more flexible branching patterns. MPST-based behavioral type properties [SY19] and model checking [HY17] can increase the expressiveness of global types. Enabling more flexible protocols is an orthogonal question, and we are convinced that the proposed failure handling in both models will play well together with more flexible protocols. Nonetheless, we believe that it is a worthwhile direction to pursue, primarily with the aim of making MPSTs for distributed applications more accessible to a broader audience.

Local data state. We tackle a significant challenge in this thesis: ensuring consistent communication even under concurrent failures. In our experience, this can also simplify the development of applications that ensure consistent data state under concurrent failures. Nonetheless, we think it could be interesting to combine MPSTs based failure handling with concepts that ensure data consistency. Potential options include: extending our model with checkpoints; integrating data state into the types; and establishing and verifying safe storage patterns that depend on reliable storage, e.g., ZooKeeper [Hun10].

Failure models. There are other failure models besides the standard crash-stop failure model. Crash-stop with recovery and network partitions generalize crash-stop failures and play an important role in distributed middleware systems [ATAA18]. Furthermore, Byzantine failures [LSP82] could be an exciting model, especially with the advent of Intel SGX and blockchains.

Verification of custom functional properties. Our works establish properties like subject reduction, which provides communication safety and progress. It could be interesting to study whether our approach can be combined with techniques such as interactive theorem proving, e.g., Coq [BC13] or SMT solving, e.g., Z3 [dMB08] to verify custom functional properties of protocols and their implementations, on top of the ensured typing properties. Using these tools in combination with our model could be advantageous. Our frameworks ensure desired properties and restrict the space of possible communications. Proving behavior properties of distributed algorithms was also a motivating factor for the work of Adameit et al. [APN17].

Further features. The model that we present in Chapter 4, provides a wide range of features and is expressive enough to express complex middleware systems such as a cluster manager. Nevertheless, there are additional features that would be desirable for

the purpose of implementing failure tolerant middleware systems. Such features include failure-free addition and removal of participants to and from a session to adjust to changes in resource demands (explicit connection actions [HY17] could be an option for that); and external interaction, i.e., interaction with external parties and not just between participants in a session. External interactions can affect the communication in a session and commonly occurs in middleware systems.

Further empirical evaluation. We implemented practical algorithms in our models, e.g., a distributed logistic regression model and practical systems such as a CM for Apache Spark. These are essential steps, but we believe that the study of additional middleware systems that deal with failure is desirable. This would demonstrate the expressiveness of our models and identify potential shortcomings.

Optimization. The focus of this work was to ensure safe partial failure handling. Performance was not a primary objective. We believe that more efficient synchronization algorithms that ensure safe failure handling, and optimization techniques, such as the combination of coordination messages with normal data messages, could improve the runtime performance of our model.

Bibliography

[A+16] Davide Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[ACT00] M.K. Aguilera, W. Chen, and S. Toueg. Failure Detection and Consensus in the Crash Recovery Model. *Distributed Computing*, 13(2):99–125, April 2000.

[Apa06] Apache. Hadoop, 2006. `https://hadoop.apache.org`.

[Apa08] Apache Software Foundation. Apache HBase, 2008. `http://hbase.apache.org/`.

[APN16] Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures (technical report). *CoRR*, abs/1607.07286, 2016.

[APN17] Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session Types for Link Failures. In *FORTE '17*, volume 10321, pages 1–16. Springer, 2017.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[ATAA18] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 51–68. USENIX Association, 2018.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCD+08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[BCT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *10th International Workshop on Distributed Algorithms (WDAG'96)*, Lecture Notes in Computer Science, pages 105–122. Springer, 1996.

[BDV+19] Andi Bejleri, Elton Domnori, Malte Viering, Patrick Eugster, and Mira Mezini. Comprehensive multiparty session types. *Art Sci. Eng. Program.*, 3(3):6, 2019.

[Bir17] Kenneth P. Birman. Byzantine Clients, 2017.

[Bur06] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI '06*, pages 335–350. USENIX Association, 2006.

[BVSE17] Marcel Blöcher, Malte Viering, Stefan Schmid, and Patrick Eugster. The grand CRU challenge. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems, HotConNet@SIGCOMM 2017, Los Angeles, CA, USA, August 25, 2017*, pages 7–11. ACM, 2017.

[CAP17] Mauricio Cano, Jaime Arias, and Jorge A. Pérez. Session-based concurrency, reactively. In *FORTE '17*, volume 10321 of *Lecture Notes in Computer Science*, pages 74–91. Springer, 2017.

[CBD+11] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In Roberto Bruni and Vladimiro Sassone, editors, *Trustworthy Global Computing - 6th International Symposium, TGC 2011, Aachen, Germany, June 9-10, 2011. Revised Selected Papers*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2011.

[CBS09] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[CCE+21] Nathan Chong, Byron Cook, Jonathan Eidelman, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar

Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow at amazon web services. *Softw. Pract. Exp.*, 51(4):772–797, 2021.

[CDA16] Minas Charalambides, Peter Dinges, and Gul A. Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.*, 115-116:100–126, 2016.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*, pages 205–218. USENIX Association, 2006.

[CDYP16] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 26(2):238–302, 2016.

[CGY16] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *MSCS*, 26(2):156–205, 2016.

[CHJ+19] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *PACMPL*, 3(POPL):29:1–29:30, 2019.

[CHTCB96] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the Impossibility of Group Membership. In *PODC '96*, pages 322–330. ACM, 1996.

[CHY08] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Interactional Exceptions in Session Types. In *CONCUR '08*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[CKV01] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, December 2001.

[CLM+16] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR '16*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[CP16] Luís Caires and Jorge A. Pérez. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *FORTE '16*, volume 9688 of *LNCS*, pages 74–95. Springer, 2016.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.

[CVB+16] Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A Type Theory for Robust Failure Handling in Distributed Systems. In *FORTE '16*, volume 9688, pages 96–113. Springer, 2016.

[CYH09a] Marco Carbone, Nobuko Yoshida, and Kohei Honda. Asynchronous session types: Exceptions and multiparty interactions. In *Formal Methods for Web Services*, pages 187–212, 2009.

[CYH09b] Marco Carbone, Nobuko Yoshida, and Kohei Honda. Asynchronous Session Types: Exceptions and Multiparty Interactions. In *SFM 2009*, volume 5569 of *LNCS*, pages 187–212. Springer, 2009.

[DH12] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR '12*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012.

[DHH+15] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical Interruptible Conversations. *Formal Methods in System Design*, 46(3):197–225, 2015.

[DHZ16] Cezara Dragoi, Thomas Henzinger, and Damien Zufferey. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *POPL '16*, pages 400–415. ACM, 2016.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic Multirole Session Types. In *POPL '11*, pages 435–446. ACM, 2011.

[DY12] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.

[DYBH12] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012.

[FLMD19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *PACMPL*, 3(POPL):28:1–28:29, 2019.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43. ACM, 2003.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.

[GR17] Simon Gay and Antonio Ravara, editors. *Behavioural Types: from Theory to Tools*. River Publishers series in automation, control and robotics. River Publishers, June 2017.

[GS01] Rachid Guerraoui and André Schiper. The Generic Consensus Service. *IEEE Trans. Software Eng.*, 27(1):29–41, 2001.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[HKP+10] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *ECOOP '10*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010.

[HKZ+11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, April 2011.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[Hun10] Patrick Hunt. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX '10*. USENIX Association, 2010.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[HY16] Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation . In *FASE '16*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016.

[HY17] Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE '17*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9:1–9:67, 2016.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[IYY17] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: A Session-Based Library with Polarities and Lenses. In *COORDINATION '17*, volume 10319 of *LNCS*, pages 99–118. Springer, 2017.

[KAB+07] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI '07*, volume 42, pages 179–188. ACM, 2007.

[KLVW17] Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 719–734. ACM, 2017.

[KNR11] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. In *NetDB '11*, 2011.

[KY14] Dimitrios Kouzapas and Nobuko Yoshida. Globally Governed Session Semantics. *LMCS*, 10(4), 2014.

[LM16] Sam Lindley and J. Garrett Morris. Embedding Session Types in Haskell. In *Haskell '16*, pages 133–145. ACM, 2016.

[LSP82] Leslie Lamport, Robert Shostak, and Marshal Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[LWH+11] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *SOSP '11*, pages 279–294. ACM, 2011.

[MFYZ21] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In Aaron Smith, Delphine Demange, and Rajiv Gupta, editors, *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*, pages 94–106. ACM, 2021.

[MGJ+19] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

[MI14] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[MY15] Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.

[NdFCY15] Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by default - safe MPI code generation based on session types. In Björn Franke, editor, *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9031 of *Lecture Notes in Computer Science*, pages 212–232. Springer, 2015.

[NHYA18] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 128–138. ACM, 2018.

[NN93] Flemming Nielson and Hanne Riis Nielson. From CML to process algebras (extended abstract). In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 1993.

[NY17] Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC '17*, pages 98–108. ACM, 2017.

[OY17] Dominic Orchard and Nobuko Yoshida. Session types with linearity in haskell. In António Ravara Simon Gay, editor, *Behavioural Types: from Theory to Tools*, Lecture Notes in Computer Science, page 219–241. River Publishers, 2017.

[Pad17] Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.

[PdMB10] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reason.*, 44(4):401–424, 2010.

[PMP+16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016.

[PT08] Riccardo Pucella and Jesse A. Tov. Haskell Session Types with (Almost) No Class. In *Haskell '08*, pages 25–36. ACM, 2008.

[SDHY17a] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *ECOOP 2017*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[SDHY17b] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP '17*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*, pages 1–10. IEEE Computer Society, 2010.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.

[SNS+11] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. *SIGPLAN Not.*, 46(10):713–732, October 2011.

[SQZ+13] K. C. Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj, and Patrick Eugster. Efficient sessions. *Sci. Comput. Program.*, 78(2):147–167, 2013.

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018.

[SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[SY19] Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *PACMPL*, 3(POPL):30:1–30:29, 2019.

[TLM+18] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In Jeffrey S. Foster and Dan Grossman, editors, *PLDI '18*, pages 662–677. ACM, 2018.

[VCE+18] Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. A typing discipline for statically verified crash failure handling in distributed systems. In *ESOP '18*, volume 10801 of *Lecture Notes in Computer Science*, pages 799–826. Springer, 2018.

[VCS08] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *ESOP '08*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

[vGKB+19] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, 2019.

[VHEZ21] Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021.

[VMD+13a] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16. ACM, 2013.

[VMD+13b] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13*, pages 5:1–5:16. ACM, 2013.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI '15*, pages 357–368. ACM, 2015.

[XRR+95] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery. In *FTCS '95*, pages 499–508. IEEE Computer Society, 1995.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In Jennifer Rexford and Emin Gün Sirer, editors, *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 213–228. USENIX Association, 2009.

[YDBH10] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.

[ZCD+12a] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, pages 2–2, 2012.

[ZCD+12b] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI '12*, pages 15–28, 2012.

[ZFH+20] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020.

A. Appendix for Chapter 3

A.1. Additional Examples: Examples of MPST Types and Processes with Coordinator-based Failure Handling

We give versions of the main motivating examples in standard MPST literature [HYC16] extended to support partial failures.

Two-buyers protocol

The Two-Buyers Protocol [HYC16, § 2.3,3.4] derives from a Web services use case. In the original protocol specification, the roles Buyer1 (`B1`) and Buyer2 (`B2`) carry out a joint transaction to buy a book from an online Seller (`S`). The role of `B1` is to offer to pay some share of the total price to `B2`. `B2` makes the final *choice* whether to proceed with the purchase (by paying the remaining amount) or not.

Figure A.1 extends the original global type to a fault tolerant version of this application protocol, bearing in mind the asymmetry of the `B1` and `B2` roles. We shall assume `S` is robust.

In the initial exchanges, given by the inner try-block, `B2` sends `S` the name of the book (l_1), `S` sends `B1` and `B2` the price (l_2), and `B1` sends `B2` the amount it is willing to contribute (l_3) (as in the original protocol). If `B1` crashes during this part, then `S` acknowledges this event by resending the price to `B2` (l_7) in the inner handling enviroment—asynchrony and inherent concurrency of the interactions between `S` and the Buyers and this potential failure means that `S` cannot be certain about the order of the relevant messages arriving at `B2`.

Whether or not `B1` crashes, `B2` and `S` proceed to the choice of `B2` to buy (`ok`) or not buy (`quit`) the book (this segment is also as in the original protocol). However, if `B2` crashes at any point of the protocol, then the protocol must simply end for `S`, and also for `B1` if it is live, as given by the outer handling environment.

$$
\begin{array}{l}
G_{\mathsf{TB}} = \ (\\
\quad (\\
\quad\quad \mathtt{B2}\to\mathtt{S}\ l_1(\mathtt{string}).\\
\quad\quad \mathtt{S}\to\mathtt{B1}\ l_2(\mathtt{int}).\\
\quad\quad \mathtt{S}\to\mathtt{B2}\ l_3(\mathtt{int}).\\
\quad\quad \mathtt{B1}\to\mathtt{B2}\ l_4(\mathtt{int}).\\
\quad\quad \mathtt{B2} \to \mathtt{S} \begin{cases} \mathtt{ok}_1().\mathtt{B2}\to\mathtt{S}\ l_5(\mathtt{string}).\mathtt{S}\to\mathtt{B2}\ l_6(\mathtt{date}).\mathsf{end} \\ \mathtt{quit}_1().\mathsf{end} \end{cases}\\
\quad \blacktriangleright\\
\quad\quad \{\mathtt{B1}\} : \mathtt{S}\to\mathtt{B2}\ l_7(\mathtt{int}).\\
\quad\quad\quad \mathtt{B2} \to \mathtt{S} \begin{cases} \mathtt{ok}_2().\mathtt{B2}\to\mathtt{S}\ l_8(\mathtt{string}).\mathtt{S}\to\mathtt{B2}\ l_9(\mathtt{date}).\mathsf{end} \\ \mathtt{quit}_2().\mathsf{end} \end{cases}\\
\quad)^2.\mathsf{end}\\
\blacktriangleright\\
\quad \{\mathtt{B2}\} : \mathsf{end},\\
\quad \{\mathtt{B1}, \mathtt{B2}\} : \mathsf{end}\\
)^1.\\
\mathsf{end}
\end{array}
$$

Figure A.1.: The Two-Buyers Protocol [HYC16, § 3.4] extended with partial failure handling.

Projection to local types

The local type projection (Definition 10) of G_{TB} onto $\mathtt{B1}$, $G_{\mathsf{TB}}\restriction\mathtt{B1}$, is:

$$
\begin{array}{l}
(\\
\quad \big(\mathtt{S}?l_2(\mathtt{int}).\mathtt{B2}!l_4(\mathtt{int}).\mathsf{end} \blacktriangleright \{\mathtt{B1}\} : \mathsf{end}\big)^{(2,\emptyset)}.\mathsf{end}\\
\blacktriangleright \{\mathtt{B2}\} : \mathsf{end}, \{\mathtt{B1}, \mathtt{B2}\} : \mathsf{end})^{(1,\emptyset)}.\mathsf{end}
\end{array}
$$

The projection onto $\mathtt{B2}$, $G_{\mathsf{TB}}\restriction\mathtt{B2}$, is:

$$
\begin{array}{l}
\Big(\\
\quad \Big(\\
\qquad \mathtt{S}!l_1(\mathtt{string}).\mathtt{S}?l_3(\mathtt{int}).\mathtt{B1}?l_4(\mathtt{int}). \\
\qquad \mathtt{S}!\begin{cases} \mathtt{ok}_1().\mathtt{S}!l_5(\mathtt{string}).\mathtt{S}?l_6(\mathtt{date}).\mathsf{end} \\ \mathtt{quit}_1().\mathsf{end} \end{cases} \\
\quad \blacktriangleright \\
\qquad \{\mathtt{B1}\} : \mathtt{S}?l_7(\mathtt{int}). \\
\qquad\quad \mathtt{S}!\begin{cases} \mathtt{ok}_2().\mathtt{S}!l_8(\mathtt{string}).\mathtt{S}?l_9(\mathtt{date}).\mathsf{end} \\ \mathtt{quit}_2().\mathsf{end} \end{cases} \\
\quad \Big)^{(2,\emptyset)} \\
\blacktriangleright \{\mathtt{B2}\} : \mathsf{end}, \{\mathtt{B1},\mathtt{B2}\} : \mathsf{end}\Big)^{(1,\emptyset)}.\mathsf{end}
\end{array}
$$

The projection onto $\mathtt{S}$, $G_{\mathsf{TB}}\restriction\mathtt{S}$, is:

$$
\begin{array}{l}
\Big(\\
\quad \Big(\\
\qquad \mathtt{B2}?l_1(\mathtt{string}).\mathtt{B1}!l_2(\mathtt{int}).\mathtt{B2}!l_3(\mathtt{int}). \\
\qquad \mathtt{B2}?\begin{cases} \mathtt{ok}().\mathtt{B2}?l_5(\mathtt{string}).\mathtt{B2}!l_6(\mathtt{date}).\mathsf{end} \\ \mathtt{quit}().\mathsf{end} \end{cases} \\
\quad \blacktriangleright \\
\qquad \{\mathtt{B1}\} : \mathtt{B2}!l_7(\mathtt{int}).\mathsf{end} \\
\qquad\quad \mathtt{B2}?\begin{cases} \mathtt{ok}().\mathtt{B2}?l_8(\mathtt{string}).\mathtt{B2}!l_9(\mathtt{date}).\mathsf{end} \\ \mathtt{quit}().\mathsf{end} \end{cases} \\
\quad \Big)^{(2,\emptyset)} \\
\blacktriangleright \{\mathtt{B2}\} : \mathsf{end}, \{\mathtt{B1},\mathtt{B2}\} : \mathsf{end}\Big)^{(1,\emptyset)}.\mathsf{end}
\end{array}
$$

Processes

We give example of well-typed processes implementing each of roles.

For $\mathtt{B1}$:

$$
\begin{array}{l}
a[\mathtt{B1}](y).y : \Big(\\
\quad \big(\mathtt{S}?l_2(x_2).\mathtt{B2}!l_4(x_2 \div 2).0 \blacktriangleright \{\mathtt{B1}\} : 0\big)^{(2,\emptyset)}.0 \\
\blacktriangleright \{\mathtt{B2}\} : 0, \{\mathtt{B1},\mathtt{B2}\} : 0\Big)^{(1,\emptyset)}.0
\end{array}
$$

For $\mathtt{B2}$:

$$
\begin{array}{l}
a[\mathtt{B2}](y).y : \\
\big(\\
\quad \big(\\
\qquad \mathsf{def}\ X(z) = \mathsf{if}\ \ (z < 100)\ \ \mathtt{S}!\mathtt{ok}_1().\mathtt{S}!l_5(\text{“addr”}).\mathtt{S}?l_6(x_6).0 \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathtt{S}!\mathtt{quit}_1().0 \\
\qquad\quad \mathsf{in}\ \ \mathtt{S}!l_1(\text{“title”}).\mathtt{S}?l_2(x_2).\mathtt{B1}?l_4(x_4).X\langle x_2 - x_4\rangle\big) \\
\quad \blacktriangleright\{\mathtt{B1}\} : \\
\qquad \mathsf{def}\ X(z) = \mathsf{if}\ \ (z < 100)\ \ \mathtt{S}!\mathtt{ok}_2().\mathtt{S}!l_8(\text{“addr”}).\mathtt{S}?l_9(x_9).0 \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathtt{S}!\mathtt{quit}_2().0 \\
\qquad\quad \mathsf{in}\ \ \mathtt{S}?l_7(x_7).X\langle x_7\rangle \\
\quad \big)^{(2,\emptyset)}.0 \\
\blacktriangleright\{\mathtt{B2}\} : 0, \{\mathtt{B1}, \mathtt{B2}\} : 0\big)^{(1,\emptyset)}.0
\end{array}
$$

For S, assuming some helper functions getPrice and getData on data:

$$
\begin{array}{l}
a[\mathtt{S}](y).y : \\
\big(\\
\quad \big(\\
\qquad \mathsf{def}\ X() = \mathtt{B2}?\{\mathtt{ok}_1().\mathtt{B2}?l_5(x_5).\mathtt{S}?l_6(\mathsf{getDate}(x_5)).0, \mathtt{quit}_1().0\} \\
\qquad\quad \mathsf{in}\ \ \mathtt{B2}?l_1(x_1).\mathtt{B1}!l_2(\mathsf{getPrice}(x_1)).\mathtt{B2}!l_4(\mathsf{getPrice}(x_1)).X\langle\rangle\big) \\
\quad \blacktriangleright\{\mathtt{B1}\} : \\
\qquad \mathsf{def}\ X() = \mathtt{B2}?\{\mathtt{ok}_2().\mathtt{B2}?l_8(x_8).\mathtt{S}?l_9(\mathsf{getDate}(x_8)).0, \mathtt{quit}_2().0\} \\
\qquad\quad \mathsf{in}\ \ \mathtt{B1}!l_7(99).X\langle\rangle \\
\quad \big)^{(2,\emptyset)}.0 \\
\blacktriangleright\{\mathtt{B2}\} : 0, \{\mathtt{B1}, \mathtt{B2}\} : 0 \\
\big)^{(1,\emptyset)}.0
\end{array}
$$

Streaming protocol

The original Streaming Protocol [HYC16, § 3.4] demonstrated a *recursive* global type for a continuous stream of messages, where two producer roles (DP, KP) independently send to a middleman role (K) in a join pattern, followed by K forwarding a message to a consumer role (C). (The role names are taken from the original protocol.)

Figure A.2 extends the protocol to handle the potential failures of the DP and KP; we assume the other two roles are robust. The idea is if just one of the producers crashes, the now fault tolerant protocol should attempt to continue with the other producer. We keep the “once-unfolded” specification from the original protocol definition in the try-block, but use the shorter folded versions in the handling activities.

$$
\begin{array}{l}
(\\
\quad \mu t.\\
\quad \mathtt{DP}{\rightarrow}\mathtt{K}\ l_1(\mathtt{bool}).\\
\quad \mathtt{KP}{\rightarrow}\mathtt{K}\ l_2(\mathtt{bool}).\\
\quad \mathtt{K}{\rightarrow}\mathtt{C}\ l_3(\mathtt{bool}).\\
\quad \mathtt{DP}{\rightarrow}\mathtt{K}\ l_1(\mathtt{bool}).\\
\quad \mathtt{KP}{\rightarrow}\mathtt{K}\ l_2(\mathtt{bool}).\\
\quad \mathtt{K}{\rightarrow}\mathtt{C}\ l_3(\mathtt{bool}).\\
\quad t\\
\blacktriangleright\\
\quad \{\mathtt{DP}\}: \quad \mu t'.\mathtt{KP}{\rightarrow}\mathtt{K}\ l_4(\mathtt{bool}).\mathtt{K}{\rightarrow}\mathtt{C}\ l_5(\mathtt{bool}).t'\\
\quad \{\mathtt{KP}\}: \quad \mu t''.\mathtt{DP}{\rightarrow}\mathtt{K}\ l_6(\mathtt{bool}).\mathtt{K}{\rightarrow}\mathtt{C}\ l_7(\mathtt{bool}).t''\\
\quad \{\mathtt{KP},\mathtt{KP}\}: \quad \mathsf{end}\\
)^1
\end{array}
$$

Figure A.2.: The Stream Protocol [HYC16, § 3.4] extended with partial failure handling.

This example also gives some intuition for the complexity in the design of our failure handling constructs: *without* any explicit choice construct, adding failure handling naturally introduces a *safe* notion of choice into the protocol between distinct paths, and also allows the normally recursive protocol to end (if both producers crash).

A.2. Proofs

A.2.1. Preservation of coherence: supporting definitions and lemmas

To reason about asynchrony in Δ, we define message types permutation, which is very similar to Definition 6.

Definition 22 (Permutable Message Types)**.** *We define* $\mathtt{m}_i \cdot \mathtt{m}_j \curvearrowright \mathtt{m}_j \cdot \mathtt{m}_i$*,* $i \neq j$*, saying* $\mathtt{m}_i \cdot \mathtt{m}_j$ *can be permuted to* $\mathtt{m}_j \cdot \mathtt{m}_i$*, if* none *of the following conditions holds:*

- $\mathtt{m}_i = \langle p, q, l(S)\rangle$ *and* $\mathtt{m}_j = \langle p, q, l'(S')\rangle$ *for some* l, l', S, S'*.*
- $\mathtt{m}_i = \langle \psi, q\rangle^{\phi}$ *and* $\mathtt{m}_j = \langle\!\langle q, F\rangle\!\rangle$ *for some* ϕ, F*.*

The following simple lemmas and definitions will be used in the proof for Theorem 1 (Preservation of Coherence).

Lemma 1. *If $L \downharpoonleft p = \mathcal{E}[p\ w\ \{l_i(S_i).L_i\}_{i\in I}]$ and $w \in \{!, ?\}$ for some $\mathcal{E}$ and $L_i, i \in I$, then for any $p' \neq p$ we have for any $j, k \in I$, $j \neq k$, $\mathcal{E}[L_j] \downharpoonleft p' = \mathcal{E}[L_k] \downharpoonleft p'$.*

Proof. Immediately by Definition 10. □

The next lemma states that, given a session environment, if each of its endpoint's types is projected from a well-formed global type, then this session environment is coherent.

Lemma 2. *Given G is well-formed and $\Delta_s = \{s[p_1] : L_1, ..., s[p_n] : L_n, s : \emptyset\}$ and $\{p_1, ..., p_n\} = roles(G)$ and $L_i = G\upharpoonright p_i$ for $i \in \{1..n\}$. Then $\Gamma, G : (\emptyset, \emptyset) \vdash \Delta_s$ is coherent.*

Proof. By the structure of a well-formed G, the proof is immediate by Definition 10 (Projection), Definition 15 (Duality), and Definition 16 (Coherence). □

And, after a failure occurs, if a coordinator has not yet issued failure notifications to endpoints in a coherent Δ for handling this failure, the types of all non-failed endpoints are still dual to each other in Δ.

Lemma 3. *Assuming $\Delta = \Delta_0, \{s : \mathtt{h}\}$ is coherent and $\Delta \rightarrow_L \Delta_0 \setminus \{s[p] : L\}, \{s : remove(\mathtt{h}, p) \cdot \langle\!\langle \psi, p \rangle\!\rangle\} = \Delta'$. Then $\forall s[p] : L \in \Delta$, if $s[q] : L' \in \Delta$ then $s[p] : L \downharpoonleft q - (\mathtt{h})_{q\rightarrow p} \bowtie s[q] : L' \downharpoonleft p - (\mathtt{h})_{p\rightarrow q}$.*

Proof. Assume the non-failed endpoints are $\{s[p_i] : L_i\}_{i\in I}$. Since Δ is coherent, $\{s[p_i] : L_i\}_{i\in I}$ in Δ are dual to each other after considering the effect of $\mathtt{h}$ on each of them. By the rules defined in Definition 14, we know $\langle\!\langle \psi, p \rangle\!\rangle$ will not affect any non-failed endpoints. So the endpoints of $\{s[p_i] : L_i\}_{i\in I}$ in Δ' are still dual to each other after considering the effect of $remove(\mathtt{h}, p) \cdot \langle\!\langle \psi, p \rangle\!\rangle$ on each of them. □

Lemma 4. *Let $L - \mathtt{ht} = L_1$, and $L - \mathtt{ht} = L_2$ then $L_1 = L_2$*

Proof. Proof by induction over the length of $\mathtt{ht}$. □

A.2.2. Preservation of coherence: proof

Theorem 1 (Preservation of Coherence)**.** *$\Gamma \vdash \Delta$ coherent and $\Gamma = G :_s (F, h_d), \Gamma'$ and $G :_s (F, h_d) \vdash \Delta \rightarrow_L G :_s (F', h'_d) \vdash \Delta'$ imply that $\Gamma', G :_s (F', h'_d) \vdash \Delta'$ is coherent.*

Proof. We prove the statement by mechanically proving each case. The cases can involve outer $[\![\mathtt{Str}]\!]$ which we handle implicitly.

For convenience we sometimes write $s[p] : L \downharpoonleft p - (\mathtt{h})_{p\rightarrow q} = s[p] : L \downharpoonleft p$ for $(\mathtt{h})_{p\rightarrow q} \neq \emptyset$ if $(\mathtt{h})_{p\rightarrow q}$ contains only failure notification and they have no effect on $s[p] : L \downharpoonleft p$ We show interesting and pragmatic cases.

(I) Case $[\![\mathsf{Snd}]\!]$, there exists $s[p] : L \in \Delta, L = \mathcal{E}[q!\{l_i(S_i).T_i\}_{i\in I}]$.

Without loss of generality assume

$$\Delta = \Delta_0, s[p] : L, s : \mathtt{h}$$

such that

$$\begin{array}{l} G : (F_q, h_d) \vdash \Delta_0, s[p] : L, s : \mathtt{h} \\ \quad \to_L G : (F_q, h_d) \vdash \Delta_0, s[p] : \mathcal{E}[L_k], s : \mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle \end{array}$$

for some $k \in I$. Let $\Delta' = \Delta_0, s[p] : \mathcal{E}[L_k], s : \mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle$.

Then we have the following cases:

(A) No notification in $\mathtt{h}$ makes the send obsolete, i.e. $\mathtt{h}$ contains no failure notifications that activate a handling environment in an enclosing try-handle.

In Δ, by Definition 14 (The Effect of $\mathtt{ht}$) and Definition 16 (2). (a), for any $s[q'] : L' \in \Delta$ we have

$$\begin{array}{rcl} s[p] : L \downharpoonleft q' - (\mathtt{h})_{q'\to p} & = & s[p] : \mathcal{E}[q!\{l_i(S_i).L_i\}_{i\in I}] \downharpoonleft q' - (\mathtt{h})_{q'\to p} \\ & \bowtie & s[q'] : L' \downharpoonleft p - (\mathtt{h})_{p\to q'} \end{array}$$

In Δ, if $q' = q$, we have

$$L \downharpoonleft q - (\mathtt{h})_{q\to p} = \mathcal{E}[q!\{l_i(S_i).L_i\}_{i\in I}] \downharpoonleft q - (\mathtt{h})_{q\to p} = \mathcal{E}''[q!\{l_i(S_i).L_i'''\}_{i\in I}]$$

for some $\mathcal{E}''$ and $L_i''', i \in I$. Note that this case assumes $\mathtt{h}$ contains no message which will trigger failure handling at p or q. By Definition 15 (Duality), we have $L' \downharpoonleft p - (\mathtt{h})_{p\to q} = \mathcal{E}'[p?\{l_i(S_i).L_i'\}_{i\in I}]$ for some $\mathcal{E}'$ and

$$\forall i \in I.\ s[p] : \mathcal{E}''[L_i'''] \bowtie s[q] : \mathcal{E}'[L_i'].$$

By Definition 14 (The Effect of $\mathtt{ht}$) and Definition 15 (Duality)

$$\begin{array}{rl} & s[p] : \mathcal{E}[L_k] \downharpoonleft q - (\mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle)_{q\to p} \\ = & s[p] : \mathcal{E}[L_k] \downharpoonleft q \\ = & s[p] : \mathcal{E}''[L_k'''] \\ \bowtie & s[q] : \mathcal{E}'[L_k'] \\ = & s[q] : L' \downharpoonleft p - (\mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle)_{p\to q} \end{array}$$

Since the types of other endpoints are not changed, $\Gamma \vdash \Delta'$ is coherent.

(B) $\mathtt{h}$ contains failure notifications that make the send obsolete.

Without loss of generality let $F \subseteq Fset(\mathtt{h}, p)$ be the failure set which triggers the out most (activatable) try-handle at p.

Without loss of generality assume $\Delta = \Delta_0, s[p] : L, s[q] : L', s : \mathtt{h}$ and assume $L = \mathcal{E}[q!\{l_i(S_i).L_i\}_{i\in I}]$ and

$$\begin{array}{l} G : (F_q, h_d) \vdash \Delta_0, s[p] : L, s[q] : L', s : \mathtt{h} \rightarrow_L \\ \quad G : (F_q, h_d) \vdash \Delta_0, s[p] : \mathcal{E}[L_k], s[q] : L', s : \mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle \end{array}$$

In Δ we observe: $L = \mathcal{E}'''[\big(\mathcal{E}''[q!\{l_i(S_i).L_i\}_{i\in I}] \blacktriangleright F : L_F, H^{\downarrow}\big)^{(\kappa, F_s{}')}]$, where L_F is the handler body for F, and $F = \cup\{A \mid A \in \mathit{dom}(F : L_F, H^{\downarrow}) \wedge A \subseteq Fset(\mathtt{h}, p)\}$ (notice that $\langle \psi, p\rangle^{(\kappa, F_s{}')} \notin \mathtt{h}$ because endpoint $s[p]$ who is still taking actions in ϕ), then by Definition 16 and Definition 15 (Duality) and Definition 14 (The Effect of $\mathtt{ht}$), we have

$$\begin{array}{rl} & s[p] : L \downharpoonleft q - (\mathtt{h})_{q\rightarrow p} \\ = & s[p] : \mathcal{E}'''[\big(\mathcal{E}''[q!\{l_i(S_i).L_i\}_{i\in I}] \blacktriangleright F : L_F, H^{\downarrow}\big)^{(\kappa, F_s{}')}] \downharpoonleft q - (\mathtt{h})_{q\rightarrow p} \\ = & s[p] : \mathcal{E}_5[\big(L'_F \blacktriangleright F : L_F, H^{\downarrow}\big)^{(\kappa, F)}] \\ \bowtie & s[q] : L' \downharpoonleft p - (\mathtt{h})_{p\rightarrow q} \\ = & s[q] : \mathcal{E}_4[\big(L''_F \blacktriangleright H^{\downarrow\prime}\big)^{(\kappa, F)}] \end{array}$$

where $L'_F \bowtie L''_F$ by Definition 15.

Now $l_k \notin \mathit{labels}(\mathcal{E}_4[\big(L''_F \blacktriangleright H^{\downarrow\prime}\big)^{(\kappa, F)}])$ due to the occurrence of failure (case failure makes send obsolete) and Definition 4.

In Δ' we observe

$$\begin{array}{rl} & s[p] : \mathcal{E}[L_k] \downharpoonleft q - (\mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle)_{q\rightarrow p} \\ = & s[p] : \mathcal{E}_5[\big(L'_F \blacktriangleright F : L_F, H^{\downarrow}\big)^{(\kappa, F)}] \\ \bowtie & s[q] : \mathcal{E}_4[\big(L''_F \blacktriangleright H^{\downarrow\prime}\big)^{(\kappa, F)}] \\ = & s[q] : \mathcal{E}_4[\big(L''_F \blacktriangleright H^{\downarrow\prime}\big)^{(\kappa, F)}] - p?l_k(S_k) \\ = & s[q] : L' \downharpoonleft p - (\mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle)_{p\rightarrow q} \\ = & s[q] : L' \downharpoonleft p - (\mathtt{h})_{p\rightarrow q} \end{array}$$

because $\langle p, q, l_k(S_k)\rangle$ can be removed by Definition 4 (Well-formedness).(3) and rule $[\![\mathtt{Cln}]\!]$ since $l_k \notin \mathit{labels}(\mathcal{E}_4[\big(L''_F \blacktriangleright H^{\downarrow\prime}\big)^{(\kappa, F)}])$ (i.e., $\langle p, q, l_k(S_k)\rangle$ is a message sent out after failures which can be handled by $s[p]$ and $s[q]$ occur).

(C) A done notification makes the send obsolete. This case cannot occur. Assume this notification exists: $\exists\langle\psi, p'\rangle^{\phi} \in \mathtt{h}$ for some p' and $\phi = (\kappa, F_s)$.

Let the try-handle of κ in G be $\left(G_0 \blacktriangleright H^{\uparrow}\right)^{\kappa}$. If $\langle\psi, p_i\rangle^{\phi} \in \mathtt{h}$, then $p_i \in \mathit{roles}(\left(G_0 \blacktriangleright H^{\uparrow}\right)^{\kappa})$.

Without loss of generality assume $\Delta = \Delta_0, s[p] : L, s[q] : L', s : \mathtt{h}$ and assume $L = \mathcal{E}[q!\{l_i(S_i).L''_i\}_{i\in I}]$

$$\begin{array}{l} G : (F_q, h_d) \vdash \Delta_0, s[p] : L, s[q] : L', s : \mathtt{h} \rightarrow_L \\ \quad G : (F_q, h_d) \vdash \Delta_0, s[p] : \mathcal{E}[L''_k], s[q] : L', s : \mathtt{h} \cdot \langle p, q, l_k(S_k)\rangle \end{array}$$

By case (done makes the send obsolete) we have that $[\![\mathtt{ClnDone}]\!]$ does not apply to $\langle\psi, p'\rangle^{\phi}$.

By $[\![\mathtt{SndDone}]\!]$, $[\![\mathtt{IssueDone}]\!]$ and $[\![\mathtt{ClnDone}]\!]$ not applicable we have:

$$L = \mathcal{E}[\left(\underline{\mathsf{end}} \blacktriangleright H^{\downarrow}\right)^{\phi}.L'''']$$

Therefore, we have a contradiction as by case we have: $L = \mathcal{E}[q!\{l_i(S_i).L''_i\}_{i\in I}]$

(II) Case $[\![\mathtt{Crash}]\!]$, there exists some $s[p] : L \in \Delta$ that fails, and a failure notification $\langle\!\langle\psi, \{p\}\rangle\!\rangle$ is sent to the coordinator. Definition 16 (duality) is only required between alive participants and the failure notification only effects Definition 14 (The Effect of $\mathtt{ht}$) once they are forwarded by the coordinator.

(III) Case $[\![\mathtt{F}]\!]$. The coordinator adds a notification $\langle\!\langle\widetilde{p}, F\rangle\!\rangle$ into the queue type to notify the endpoint acting as roles $\widetilde{p}$ that F occurs.

This case, particularly, gives reasons for

a) Definition 4 (Well-formedness).(5) ensures either F will be handled or there is no more interactions involving $\forall q \in F$.

b) Definition 4 (Well-formedness). (1),(2) work for avoiding any confusion among partners for handling some F.

c) With the above reasons, a coordinator needs to keep a well-formed G as a global guidance.

Without loss of generality assume

- $\Delta = \Delta_0, s : \langle\!\langle\psi, F\rangle\!\rangle \cdot \mathtt{h}''$,
- $\Delta' = \Delta_0, s : \mathtt{h}'' \cdot \langle\!\langle\tilde{p}, F\rangle\!\rangle$,
- $\mathtt{h} = \langle\!\langle\psi, F\rangle\!\rangle \cdot \mathtt{h}''$

- $\mathtt{h}' = \mathtt{h}'' \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle$

$$G :_s (F_q, h_d) \vdash \Delta_0, s : \langle\!\langle \psi, F \rangle\!\rangle \cdot \mathtt{h}'' \rightarrow_L \\ G :_s (F_q \cup F, d) \vdash \Delta_0, s : \mathtt{h}'' \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle$$

G is well-formed $\langle\!\langle p, F \rangle\!\rangle \in \mathtt{h}$ means $\forall q \in F$ are non-robust, so a well-formed G shall handle it (somewhere) by Definition 4.5. For convenience we call all participant which occur in a try-handle partners for that try-handle.

Pick any p, q with $s[p] : L_p, s[q] : L_q \in \Delta$.

In Δ we have:

$$\begin{array}{rl} & s[p] : L_p \upharpoonright q - (\langle\!\langle \psi, F \rangle\!\rangle \cdot \mathtt{h}'')_{q \rightarrow p} \\ = & s[p] : L_p \upharpoonright q - (\mathtt{h}'')_{q \rightarrow p} \\ = & s[p] : \mathcal{E}_p[L'_p] \\ \bowtie & s[q] : L_q \upharpoonright p - (\langle\!\langle \psi, F \rangle\!\rangle \cdot \mathtt{h}'')_{p \rightarrow q} \\ = & s[q] : L_q \upharpoonright p - (\mathtt{h}'')_{p \rightarrow q} \\ = & s[q] : \mathcal{E}_q[L'_q] \end{array}$$

Let κ be the outer most try-handler in G that contains a handler triggered by a subset of $Fset(\mathtt{h}', p)$, say F', that is present in the partial types of p and q after considering done notifications in $\mathtt{h}'$ (dually forbiddings the case where its only present in either p or q). Without loss of generality assume $F \subseteq F'$ (i.e., the failure handling gets activated by the new failure), otherwise we have nothing to show.

Let both $\mathcal{E}_p[L'_p]$ and $\mathcal{E}_q[L'_q]$ have a try-handle of $(\kappa, {F_s}'')$ for some ${F_s}''$, at L'_p and L'_q respectively.

In Δ' we have to show that:

$$s[p] : L_p \upharpoonright q - (\mathtt{h}'' \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle)_{q \rightarrow p} \bowtie s[q] : L_q \upharpoonright p - (\mathtt{h}'' \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle)_{p \rightarrow q}$$

Let $\mathtt{h}_F \subseteq \mathtt{h}$ and let $\mathtt{h}_F$ contain all failure notification of $\mathtt{h}$. We have

$$\begin{array}{rl} & s[p] : L_p \upharpoonright q - (\mathtt{h}'' \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle)_{q \rightarrow p} \\ = & s[p] : (L_p \upharpoonright q - (\mathtt{h}'')_{q \rightarrow p}) - (\mathtt{h}_F \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle)_{q \rightarrow p} \\ = & s[p] : (\mathcal{E}_p[\big(L''_p \blacktriangleright H_p\big)^{(\kappa, {F_s}'')}.L'''_p]) - (\mathtt{h}_F \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle)_{q \rightarrow p} \\ = & s[p] : \mathcal{E}_p[\big(L_{pF} \blacktriangleright H_p\big)^{(\kappa, F')}.L'''_p] \\ \bowtie & s[q] : L_q \upharpoonright p - (\mathtt{h}'' \cdot \langle\!\langle \tilde{p}, F \rangle\!\rangle)_{p \rightarrow q} \end{array}$$

Thus overall, $\Gamma', G : ({F_q}', h_d) \vdash \Delta'$ is coherent.

(IV) Case $[\![\texttt{SndDone}]\!]$, there exists $s[p] : L \in \Delta, L = \mathcal{E}[\big(\mathsf{end} \blacktriangleright H^{\downarrow}\big)^{\phi}.L']$ and the reduction sends done notification $\langle p, \psi\rangle^{\phi}$ to the coordinator.

This notification (at this moment), which is abstracted as $\langle p, \psi\rangle^{\phi}$, does not affect any endpoints and end and $\underline{\mathsf{end}}$ are dual (Definition 15). The coordinator $G : (F_q, h_d)$ will eventually handle this notification (see Cases (VI) and (VII)).

(V) Case $[\![\texttt{Rcv}]\!]$. The receiving case is trivial because coherence has been ensured when the message/notification was sent out. We only prove the standard receiving case $[\![\texttt{Rcv}]\!]$, where an endpoint receives a message from the queue. We do not need to consider here that the sender failed, since by $[\![\texttt{Crash}]\!]$ we know that there is no message to be consumed (i.e., $[\![\texttt{Rcv}]\!]$ is not applicable).

Without loss of generality assume $s[p] : \mathcal{E}[q?\{l_i(S_i).L_i\}_{i\in I}] \in \Delta$ and $\mathtt{h} = \langle q, p, l_k(S_k)\rangle \cdot \mathtt{h}'$ and $k \in I$ and $\Delta' = \Delta_0, s[p] : \mathcal{E}[L_k], s : \mathtt{h}'$.

By Definition 16.(2) and Definition 14 (The Effect of $\mathtt{ht}$), for any $p' \neq q$, with $s[p'] : L' \in \Delta$, we have

$$\begin{array}{rl} & s[p] : \mathcal{E}[q?\{l_i(S_i).L_i\}_{i\in I}] \upharpoonright p' - (\langle q, p, l_k(S_k)\rangle \cdot \mathtt{h}')_{p'\to p} \\ = & s[p] : \mathcal{E}[L_1] \upharpoonright p' - (\mathtt{h}')_{p'\to p} \\ \bowtie & s[p'] : L' \upharpoonright p - (\langle q, p, l_k(S_k)\rangle \cdot \mathtt{h}')_{p\to p'} \\ = & s[p'] : L' \upharpoonright p - (\mathtt{h}')_{p\to p'} \end{array}$$

Thus in Δ', we immediately have ($\mathcal{E}[L_k] \upharpoonright p' = \mathcal{E}[L_1] \upharpoonright p'$)

$$s[p] : \mathcal{E}[L_k] \upharpoonright p' - (\mathtt{h}')_{p'\to p} \quad \bowtie \quad s[p'] : L' \upharpoonright p - (\mathtt{h}')_{p\to p'}$$

Thus $\Gamma \vdash \Delta'$ is coherent.

(VI) Case $[\![\texttt{CollectDone}]\!]$. This case is trivial because only the coordinator collects done notifications sent from participants; no endpoints are affected.

(VII) Case $[\![\texttt{IssueDone}]\!]$. By $[\![\texttt{IssueDone}]\!]$,

$$G : (F_q, h_d) \vdash \Delta = \Delta_0, s : \mathtt{h} \to_L G :_s (F_q, h'_d) \vdash \Delta' = \Delta_0, s : \mathtt{h} \cdot \mathtt{h}'$$

such that $\forall \mathtt{m} \in \mathtt{h}', \mathtt{m} = \langle \psi, p'\rangle^{(\kappa, F_s)}$ and, let $\big(\ldots \blacktriangleright H^{\uparrow}\big)^{\kappa} \in G$, we have $p' \in \mathit{roles}(G, (\kappa, F))$.

Pick any p, q with $s[p] : L_p, s[q] : L_q \in \Delta$.

Without loss of generality $p, q \in \mathit{roles}(G, (\kappa, F))$. (If either of them is not in the try-handler then the try-handler is projected out in the partial types)

In Δ we have:

$$\begin{array}{rl} & s[p] : L_p \upharpoonright q - (\mathtt{h})_{q \to p} \\ = & s[p] : L'_p \\ \bowtie & s[q] : L_q \upharpoonright p - (\mathtt{h})_{p \to q} \\ = & s[q] : L'_q \end{array}$$

Either (a) $L'_p = \mathcal{E}_p[(\underline{\mathsf{end}} \blacktriangleright ..)^{(\kappa, F_s)}.L_{p2}]$ and $L'_q = \mathcal{E}_q[(\underline{\mathsf{end}} \blacktriangleright ..)^{(\kappa, F_s)}.L_{q2}]$, or (b) $\kappa \notin L'_q \wedge \kappa \notin L'_p$ (because of failure activation)

Let $\mathtt{h}_F$ be all failure notifications in $\mathtt{h}$

(a) In Δ' we have:

$$\begin{array}{rl} & s[p] : L_p \upharpoonright q - (\mathtt{h} \cdot \mathtt{h}')_{q \to p} \\ = & s[p] : L'_p - (\mathtt{h}_F \cdot \mathtt{h}')_{q \to p} \\ = & s[p] : \mathcal{E}_p[L_{p2}] - (\mathtt{h}_F)_{q \to p} \\ = & s[p] : \mathcal{E}_p[L_{p3}] \\ \bowtie & s[q] : L_q \upharpoonright p - (\mathtt{h} \cdot \mathtt{h}')_{p \to q} \\ = & s[q] : L'_q - (\mathtt{h}_F \cdot \mathtt{h}')_{p \to q} \\ = & \mathcal{E}_q[L_{q2}] - (\mathtt{h}_F)_{p \to q} \\ = & \mathcal{E}_q[L_{q3}] \end{array}$$

Note that in L'_p the failure notifications $(\mathtt{h}_F)_{q \to q}$ can not trigger any failure handling. After the consumption of $(\mathtt{h}_F \cdot \mathtt{h}')_{q \to p}$ either (i) L_{p2} contains a try-handle which gets activated by the failure notification or (ii) the base case (only failure notification left in the queue and no other rule is applicable) gets triggered.

(b) In Δ' we have (Definition 14 (The Effect of `ht` will remove the added done notifications):

$$\begin{array}{rl} & s[p] : L_p \upharpoonright q - (\mathtt{h} \cdot \mathtt{h}')_{q \to p} \\ = & s[p] : L_p \upharpoonright q - (\mathtt{h})_{q \to p} \\ \bowtie & s[q] : L_q \upharpoonright p - (\mathtt{h} \cdot \mathtt{h}')_{p \to q} \\ = & s[q] : L_q \upharpoonright p - (\mathtt{h})_{p \to q} \end{array}$$

(VIII) Case $[\![\mathtt{TryHdl}]\!]$. In Case $[\![\mathtt{F}]\!]$, we prove that the failure notifications sent out by the coordinator with a well-formed G can trigger endpoints who are able to handle failures; moreover, Definition 16(Coherence).(2) ensures that the handler for any F are coherent in the sense that every participant has a dual interacting party to handle failures.

(IX) Case $[\![\texttt{RcvDone}]\!]$. In Case (VII) the affect of the issued done notification is already considered.

□

A.2.3. Subject reduction: supporting lemmas and definitions

Lemma 5 (Inversion Lemma)**.**

(1) *If* $\Gamma \vdash c : 0 \triangleright \Delta$*, then* $\Delta = \{c : \mathsf{end}\}$.

(2) *If* $\Gamma \vdash c : \mathsf{if}\ e\ \eta_1\ \mathsf{else}\ \eta_2 \triangleright \Delta$*, then* $\Gamma \vdash e : \mathsf{bool}$ *and* $\forall i \in \{1, 2\}$ *we have* $\Gamma \vdash c : \eta_i \triangleright \Delta$.

(3) *If* $\Gamma \vdash a[p](y).P \triangleright \Delta$*, then* $\Delta = \emptyset$ *and* $\Gamma \vdash a : \langle G \rangle$ *and* $\Gamma \vdash P \triangleright \{y : G\upharpoonright p\}$.

(4) *If* $\Gamma \vdash c : p!\ l(e).\eta \triangleright \Delta$*, then* $\Delta = \{c : L\}$ $l = l_k$ *and* $k \in I$ *and* $L = p!\ \{l_i(S_i).L'_i\}_{i \in I}$ *and* $\Gamma \vdash e : S_k$ *and* $\Gamma \vdash c : \eta \triangleright \{c : L'_k\}$.

(5) *If* $\Gamma \vdash c : p?\ \{l_i(e_i).\eta_i\}_{i \in I} \triangleright \Delta$*, then* $\Delta = \{c : L\}$ *and* $L = p?\ \{l_i(S_i).L_i\}_{i \in I}$ *and* $\forall i \in I.\ \Gamma, x_i : S_i \vdash c : \eta_i \triangleright \{c : L_i\}$.

(6) *If* $\Gamma \vdash c : \underline{0} \triangleright \Delta$*, then* $\Delta = \{c : L\}$ *and* $L = \underline{\mathsf{end}}$

(7) *If* $\Gamma \vdash c : X\langle e \rangle \triangleright \Delta$*, then* $\Delta = \{c : L\}$ *and* $\Gamma = \Gamma', X : S\ L$ *and* $\Gamma' \vdash e : S$.

(8) *If* $\Gamma \vdash c : \mathsf{def}\ D\ \mathsf{in}\ \eta_2 \triangleright \Delta$ *and* $X(x) = \eta_1 \in D$*, then* $\Delta = \{c : L\}$ *and* $\Gamma, X : S\ \mu t.L' \vdash c : \eta_2 \triangleright \{c : L\}$ *and* $\Gamma, X : S\ t, x : S \vdash c : \eta_1 \triangleright \{c : L'\}$.

(9) *If* $\Gamma \vdash c : \left(\eta \blacktriangleright H\right)^{\phi}.\eta' \triangleright \Delta$*, then* $\Delta = \{c : L\}$ *and* $L = \left(L' \blacktriangleright H^{\downarrow}\right)^{\phi}.L''$ *and* $\Gamma \vdash c : \eta \triangleright \{c : L'\}$ *and* $\Gamma \vdash c : \eta' \triangleright \{c : L''\}$ *and* $dom(H) = dom(H^{\downarrow})$ *and* $\forall F \in dom(H)$ *we have* $\Gamma \vdash c : H(F) \triangleright \{c : H^{\downarrow}(F)\}$.

(10) *If* $\Gamma \vdash s : \emptyset \triangleright \Delta$*, then* $\Delta = \{s : \emptyset\}$.

(11) *If* $\Gamma \vdash 0 \triangleright \Delta$*, then* $\Delta = \emptyset$.

(12) *If* $\Gamma \vdash s : h \cdot \langle p, q, l(e) \rangle \triangleright \Delta$*, then* $\Delta = \{s : \mathtt{h} \cdot \langle p, q, l(S) \rangle\}$ *and* $\Gamma \vdash s : h \triangleright \{s : \mathtt{h}\}$ *and* $\Gamma \vdash e : S$.

(13) *If* $\Gamma \vdash s : h \cdot \langle p_1, p_2 \rangle^{\phi} \triangleright \Delta$*, then* $(p_1, p_2) \in \{(p, \psi), (\psi, p)\}$ *for some* p *and* $\Delta = \{s : \mathtt{h} \cdot \langle p_1, p_2 \rangle^{\phi}\}$ *and* $\Gamma \vdash s : h \triangleright \{s : \mathtt{h}\}$.

(14) *If* $\Gamma \vdash s : h \cdot \llbracket q, F \rrbracket \rhd \Delta$*, then* $\Delta = \{s : \mathtt{h} \cdot \llbracket q, F \rrbracket\}$ *and* $q \in \{p, \psi\}$ *and* $\Gamma \vdash s : h \rhd \{s : \mathtt{h}\}$.

(15) *If* $\Gamma \vdash N_1 \mid N_2 \rhd \Delta$*, then* $\Gamma \vdash N_1 \rhd \Delta_1$ *and* $\Gamma \vdash N_2 \rhd \Delta_2$ *and* $\Delta = \Delta_1, \Delta_2$ *such that* $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$.

(16) *If* $\Gamma \vdash (\nu s)N \rhd \Delta$*, then* $\Gamma \vdash N \rhd \Delta'$ *and and* $\Delta = \Delta' \setminus \Delta'_s$ *and* $\Gamma \vdash \Delta'_s$ *is coherent.*

(17) *If* $\Gamma \vdash \Psi \blacklozenge N \rhd \Delta$*, then* $\Gamma = \Gamma', \Psi$ *and* $\Gamma' \vdash N \rhd \Delta$.

Proof. By induction on derivations. □

Lemma 6 (Substitution Lemma)**.**

I *If* $\Gamma, x : S \vdash N \rhd \Delta$ *and* $\Gamma \vdash v : S$*, then* $\Gamma \vdash N\{v/x\} \rhd \Delta$.

II *If* $\Gamma \vdash N \rhd \Delta, y : L$*, then* $\Gamma \vdash N\{s[p]/y\} \rhd \Delta, s[p] : L$.

Proof.

I The proof is by induction on derivation of $\Gamma, x : S \vdash N \rhd \Delta$.

II The proof is by induction on derivation of $\Gamma \vdash N \rhd \Delta, y : L$.

□

Lemma 7 (Types of Queues)**.**

1. *If* $\Gamma \vdash s : \langle p, q, l(v) \rangle \cdot h \rhd \Delta$*, then* $\Delta = \{s : \langle p, q, l(S) \rangle \cdot \mathtt{h}\}$ *and* $\Gamma \vdash s : h \rhd \{s : \mathtt{h}\}$.
2. *If* $\Gamma \vdash s : \llbracket \dagger, F \rrbracket \cdot h \rhd \Delta$*, then* $\dagger \in \{p, \psi\}$ *for some* p *and* $\Delta = \{s : \llbracket \dagger, F \rrbracket \cdot \mathtt{h}\}$ *and* $\Gamma \vdash s : h \rhd \{s : \mathtt{h}\}$.
3. *If* $\Gamma \vdash s : \langle \dagger, \dagger' \rangle^{\phi} \cdot h \rhd \Delta$*, then* $(\dagger, \dagger') \in \{(p, \psi), (\psi, p)\}$ *for some* p *and* $\Delta = \{s : \langle \dagger, \dagger' \rangle^{\phi} \cdot \mathtt{h}\}$ *and* $\Gamma \vdash s : h \rhd \{s : \mathtt{h}\}$.

Proof. For all cases, the first step follows from Lemma 5.(10). The induction step follows from Lemma 5.(12). □

Lemma 8. *Let* $\Gamma \vdash c : E[\eta] \rhd \{c : L\}$ *and* $\Gamma' \vdash c : \eta \rhd \{c : L'\}$*, then* $L = \mathcal{E}[L']$ *for some* $\mathcal{E}$.

Proof. The proof is by structural induction on contexts E.

- $E = [\,]$, then immediately $\mathcal{E} = [\,]$.

- $E = \mathsf{def}\ D\ \mathsf{in}\ E'$ and $X(x) = \eta' \in D$ and $\Gamma \vdash c : \eta \triangleright \{c : L'\}$. By applying Lemma 5.(8) to
$$\Gamma \vdash c : \mathsf{def}\ D\ \mathsf{in}\ E'[\eta] \triangleright \{c : L\}$$
we have
$$\Gamma, X : S\ \mu t.L'' \vdash c : E'[\eta] \triangleright \{c : L\} \text{ and } \Gamma, X : S\ t, x : S \vdash c : \eta' \triangleright \{c : L''\}$$
Since $\Gamma, X : S\ \mu t.L'' \vdash c : \eta \triangleright \{c : L'\}$, by induction, we have $L = \mathcal{E}[L']$.
- $E = \left(E' \blacktriangleright H\right)^{\phi}.\eta'$ and $\Gamma \vdash c : \eta \triangleright \{c : L'\}$. By applying Lemma 5.(9) to
$$\Gamma \vdash c : \left(E'[\eta] \blacktriangleright H\right)^{\phi}.\eta' \triangleright \{c : L\}$$
we have
$$L = \left(L'' \blacktriangleright H^{\downarrow}\right)^{\phi}.L''' \text{ and } \Gamma \vdash c : E'[\eta] \triangleright \{c : L''\}$$
By induction, we have $L'' = \mathcal{E}'[L']$ for some $\mathcal{E}'$. So we have
$$L = \left(\mathcal{E}'[L'] \blacktriangleright H^{\downarrow}\right)^{\phi}.L''' = \left(L'' \blacktriangleright H^{\downarrow}\right)^{\phi}.L'''$$
such that $\mathcal{E} = \left(\mathcal{E}' \blacktriangleright H^{\downarrow}\right)^{\phi}.L'''$.

□

Subject Congruence

The property of subject congruence states that if N is well-typed by some session environment, then an N' that is structurally congruent to N is well-typed by the same session environment up to message type reordering.

Theorem 6 (Subject Congr.)**.** *$\Gamma \vdash N \triangleright \Delta$ and $N \equiv N'$ imply $\Gamma \vdash N' \triangleright \Delta'$ where $\Delta \equiv \Delta'$.*

Proof. Both proofs are by induction on $\equiv$. We only list some interesting cases.

- $\dfrac{h \equiv h'}{s : h \equiv s : h'}$.
Given $h \equiv h'$, we prove that $\Gamma \vdash s : h \triangleright \Delta$ implies $\Gamma \vdash s : h' \triangleright \Delta'$ and $\Delta \equiv \Delta'$.
Let $h \equiv h'$ and $\Gamma \vdash s : h \triangleright \Delta$. The equivalence $h \equiv h'$ should come from one of the following cases: (1) $h \equiv h \cdot \emptyset \equiv h'$ or (2) $h \equiv \emptyset \cdot h \equiv h'$ or (3) $h \equiv h_1 \cdot (h_2 \cdot h_3) \equiv$

$(h_1 \cdot h_2) \cdot h_3 \equiv h'$ or (4) $h \equiv h_1 \cdot m \cdot m' \cdot h_2 \equiv h_1 \cdot m' \cdot m \cdot h_2 \equiv h'$ by given $m \cdot m' \curvearrowright m' \cdot m$.

If the structural congruence comes from cases (1), (2), or (3), since the messages in h and h' are the same and they are in the same order, then by Lemma 7, $\Gamma \vdash s : h' \rhd \Delta$.

If the structural congruence comes from case (4), given $m \cdot m' \curvearrowright m' \cdot m$, then by Lemma 5.(12), Lemma 5.(13), Lemma 5.(14) and Lemma 7,

$$\Gamma \vdash s : h_1 \cdot m \cdot m' \cdot h_2 \rhd \{s : \mathtt{h}_1 \cdot \mathtt{m} \cdot \mathtt{m}' \cdot \mathtt{h}_2\}$$

imply $\Gamma \vdash s : h_1 \rhd \{s : \mathtt{h}_1\}$ and $\Gamma \vdash s : m \rhd \{s : \mathtt{m}\}$ and $\Gamma \vdash s : m' \rhd \{s : \mathtt{m}'\}$ and $\Gamma \vdash s : h_2 \rhd \{s : \mathtt{h}_2\}$.

By $\lfloor$T-m$\rfloor$ or $\lfloor$T-D$\rfloor$ or $\lfloor$T-$F\rfloor$, we derive

$$\Gamma \vdash s : h_1 \cdot m' \cdot m \cdot h_2 \rhd \{s : \mathtt{h}_1 \cdot \mathtt{m}' \cdot \mathtt{m} \cdot \mathtt{h}_2\}.$$

By Definition 22 (Permutable Message Types),

$$\{s : \mathtt{h}_1 \cdot \mathtt{m}' \cdot \mathtt{m} \cdot \mathtt{h}_2\} \equiv \{s : \mathtt{h}_1 \cdot \mathtt{m} \cdot \mathtt{m}' \cdot \mathtt{h}_2\}$$

Thus we conclude this case.

- $\dfrac{N \equiv N'}{\Psi \blacklozenge N \equiv \Psi \blacklozenge N'}$.

  By Lemma 5.(17), then $G : (F_q, h_d) = \Psi$ and $\Gamma = \Gamma', G : (F_q, h_d)$ and $\Gamma' \vdash N \rhd \Delta$.

  By induction on $\equiv$, we have $\Gamma \vdash N' \rhd \Delta'$. By $\lfloor$T-sys$\rfloor$, we have

  $$\Gamma \vdash \Psi \blacklozenge N' \rhd \Delta'$$

  We conclude this case.

□

A.2.4. Subject reduction: proof

Theorem 2 (Subject Reduction).

(a). $\Gamma \vdash N \rhd \Delta$ *with* $\Gamma \vdash \Delta$ *coherent and* $N \to N'$ *imply that* $\exists \Delta'$ *such that* $\Gamma' \vdash N' \rhd \Delta'$ *and* $\Gamma \vdash \Delta \to_L \Gamma' \vdash \Delta'$ *or* $\Delta \equiv \Delta'$ *and* $\Gamma' \vdash \Delta'$ *coherent.*

(b). $\Gamma \vdash N \rhd \emptyset$ *and* $N \to N'$ *imply that* $\Gamma \vdash N' \rhd \emptyset$.

Proof. Proof by structural induction over the evaluation relation and relying on Theorem 1 for preservation of coherence.

Given $\Gamma' \vdash \Delta'$ coherent does not ensure $\Gamma \vdash \Delta$ coherent where $\Gamma \subseteq \Gamma'$ and $\Delta \subseteq \Delta'$. In the cases for simplicity we just assume $\Gamma \vdash \Delta$ coherent, since we know it comes from Γ' and Δ' ($\Delta' = \Delta, \Delta''$) such that $\Gamma' \vdash \Delta'$ coherent.

The proof is mostly standard and we only cover some interesting cases. We show (a) first, (b) follows then immediately.

- **(Snd)**. Assume $\Gamma \vdash s[p] : E[q!\, l(e).\eta] \mid s : h \rhd \Delta$ and $e \Downarrow v$ and $\Gamma \vdash \Delta$ is coherent.

  By applying Lemma 5.(15), we have

  $$\Delta = \Delta_1, \Delta_2 \text{ and } \Gamma \vdash s[p] : E[q!\, l(e).\eta] \rhd \Delta_1 \text{ and } \Gamma \vdash s : h \rhd \Delta_2 \tag{A.1}$$

  By applying Lemma 7 and Equation A.1 and $\lfloor$T-m$\rfloor$, we have

  $$\Delta_2 = \{s : \mathtt{h}\} \text{ and } \Gamma \vdash v : S \text{ and } \Gamma \vdash s : h \cdot \langle p, q, l(v)\rangle \rhd \{s : \mathtt{h} \cdot \langle p, q, l(S)\rangle\} \tag{A.2}$$

  By applying Lemma 5.(4) and Lemma 8 to Equation A.1, we have

  $$\Delta_1 = \{s[p] : \mathcal{E}[q!\, \{l_i(S_i).L'_i\}_{i\in I}]\} \;\; l = l_k \text{ and } \Gamma \vdash s[p] : E[\eta] \rhd \{s[p] : \mathcal{E}[L'_k]\} \tag{A.3}$$

  By $\lfloor$T-pa$\rfloor$ and Equation A.2 and Equation A.3, we derive

  $$\Gamma \vdash s[p] : E[\eta] \mid s : h \cdot \langle p, q, l(v)\rangle \rhd \{s[p] : \mathcal{E}[L'_k], s : \mathtt{h} \cdot \langle p, q, l(S)\rangle\} \tag{A.4}$$

  Let $\Delta' = \{s[p] : \mathcal{E}[L'_k], s : \mathtt{h} \cdot \langle p, q, l(S)\rangle\}$. By $[\![\mathsf{Snd}]\!]$, we have

  $$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

  and by Theorem 1 $\Gamma \vdash \Delta'$ is coherent. We conclude this case.

- **(TryHdl)**. Assume $\Gamma \vdash s[p] : E[(\eta \blacktriangleright F : \eta', H)^{(\kappa, F_s{}')}.\eta''] \mid s : h \rhd \Delta$ and $F = \cup\{A \mid A \in dom(H) \wedge F_s \subset A \subseteq Fset(h, p)\}$ and $\Gamma \vdash \Delta$ is coherent.

  By applying Lemma 5.(15), we have

  $$\begin{aligned} &\Delta = \Delta_1, \Delta_2 \text{ and} \\ &\Gamma \vdash s[p] : E[(\eta \blacktriangleright F : \eta', H)^{(\kappa, F_s{}')}.\eta''] \rhd \Delta_1 \text{ and} \\ &\Gamma \vdash s : h \rhd \Delta_2 \end{aligned} \tag{A.5}$$

By applying Lemma 8 and Lemma 5.(9) to Equation A.5, we have

$$\Delta_1 = \{s[p] : L\} \tag{A.6}$$
$$\Gamma \vdash s[p] : E[\big(\eta \blacktriangleright F : \eta', H\big)^{(\kappa, F_s{}')}.\eta''] \rhd \{s[p] : L\}$$
$$L = \mathcal{E}[\big(L' \blacktriangleright F : L'', H^{\downarrow}\big)^{(\kappa, F_s{}')}.L''']$$
$$\Gamma \vdash s[p] : \eta \rhd \{s[p] : L'\} \text{ and } \Gamma \vdash s[p] : \eta'' \rhd \{s[p] : L'''\} \text{ and}$$
$$\Gamma \vdash s[p] : \eta' \rhd \{s[p] : L''\}$$
$$dom(H) = dom(H^{\downarrow}) \text{ and}$$
$$\forall F \in dom(H).\ \Gamma \vdash s[p] : H(F) \rhd \{c : H^{\downarrow}(F)\} \tag{A.7}$$

By applying Lemma 7 to Equation A.5, we have

$$\Delta_2 = \{s : \mathtt{h}\} \text{ and } \Gamma \vdash s : h \rhd \{s : \mathtt{h}\} \tag{A.8}$$

By $\lfloor$`T-th`$\rfloor$ and Equation A.6, we have

$$\Gamma \vdash s[p] : E[\big(\eta' \blacktriangleright F : \eta', H\big)^{(\kappa, F)}.\eta''] \rhd \{s[p] : \mathcal{E}[\big(L'' \blacktriangleright F : L'', H^{\downarrow}\big)^{(\kappa, F)}.L''']\} \tag{A.9}$$

By $\lfloor$`T-pa`$\rfloor$ and Eqs. A.6, A.8, A.9, we derive

$$\Gamma \vdash s[p] : E[\big(\eta' \blacktriangleright F : \eta', H\big)^{(\kappa, F)}.\eta''] \mid s : h \rhd \{s[p] : \mathcal{E}[\big(L'' \blacktriangleright F : L'', H^{\downarrow}\big)^{(\kappa, F)}.L'''], s : \mathtt{h}\} \tag{A.10}$$

Let $\Delta' = \{s[p] : \mathcal{E}[\big(T'' \blacktriangleright F : T'', H^{\downarrow}\big)^{(\kappa, F)}.T'''], s : \mathtt{h}\}$. By $[\![$`TryHdl`$]\!]$ we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

and by Theorem 1, $\Gamma \vdash \Delta'$ is coherent. We conclude this case.

- **(Cln)**. Assume $\Gamma \vdash s[p] : E[\eta] \mid s : \langle q, p, l(v)\rangle \cdot h \rhd \Delta$ and $l \notin labels(E[\eta])$ and Δ is coherent. (Note that, Definition 14 (The Effect of `ht`) defines that this message $\langle q, p, l(v)\rangle$ will not affect any endpoints in Δ).
  By applying Lemma 5.(15), we have

$$\Delta = \Delta_1, \Delta_2 \text{ and } \Gamma \vdash s[p] : E[\eta] \rhd \Delta_1 \text{ and } \Gamma \vdash s : \langle q, p, l(v)\rangle \cdot h \rhd \Delta_2 \tag{A.11}$$

By applying Lemma 7 and Equation A.11, we have

$$\Delta_2 = \{s : \langle q, p, l(v)\rangle \cdot \mathtt{h}\} \text{ and } \Gamma \vdash v : S \text{ and } \Gamma \vdash s : h \rhd \{s : \mathtt{h}\} \tag{A.12}$$

By $\lfloor$T-pa$\rfloor$ and Equation A.11 and Equation A.12, we have

$$\Gamma \vdash s[p] : E[\eta] \mid s : h \rhd \Delta_1, \{s : \mathtt{h}\} \tag{A.13}$$

Let $\Delta' = \Delta_1, \{s : \mathtt{h}\}$. By $[\![\mathtt{Cln}]\!]$, we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

and by Theorem 1 $\Gamma \vdash \Delta'$ is coherent. We conclude this case.

- **(RcvDone)**. Assume $\Gamma \vdash s[p] : E[(\underline{0} \blacktriangleright H)^{\phi}.\eta] \mid s : h \rhd \Delta$ and $\langle \psi, p\rangle^{\phi} \in h$.
  By applying Lemma 5.(15), we have

$$\begin{aligned} &\Delta = \Delta_1, \Delta_2 \text{ and} \\ &\Gamma \vdash s[p] : E[(\underline{0} \blacktriangleright H)^{\phi}.\eta'] \rhd \Delta_1 \text{ and} \\ &\Gamma \vdash s : h \rhd \Delta_2 \end{aligned} \tag{A.14}$$

By applying Lemma 8 and Lemma 5.(6) to Equation A.14, we have

$$\Delta_1 = \{s[p] : L\} \tag{A.15}$$

$$\begin{aligned} &L = \mathcal{E}[(\underline{\mathsf{end}} \blacktriangleright H^{\downarrow})^{\phi}.L'] \text{ and} \\ &\Gamma \vdash s[p] : \underline{0} \rhd \{s[p] : \underline{\mathsf{end}}\} \text{ and } \Gamma \vdash s[p] : \eta' \rhd \{s[p] : L'\} \text{ and} \\ &dom(H) = dom(H^{\downarrow}) \text{ and } \forall F \in dom(H).\ \Gamma \vdash H(F) \rhd H^{\downarrow}(F) \end{aligned} \tag{A.16}$$

By Lemma 7 and $\lfloor$T-D$\rfloor$ and Equation A.14 and the condition that $\langle \psi, p\rangle^{\phi} \in h$, we have

$$\Delta_2 = \{s : \mathtt{h}\} \text{ and } \mathtt{h} = \mathtt{h}' \cdot \langle \psi, p\rangle^{\phi} \cdot \mathtt{h}'' \tag{A.17}$$

By applying Lemma 8 to Equation A.15, we have

$$\Gamma \vdash s[p] : E[\eta'] \rhd \{s[p] : \mathcal{E}[L']\} \tag{A.18}$$

By applying Equation A.17, we derive

$$\Gamma \vdash s : h \setminus \langle \psi, p\rangle^{\phi} \rhd \{s : \mathtt{h}' \cdot \mathtt{h}''\} \tag{A.19}$$

By $\lfloor$T-pa$\rfloor$ and Eqs. A.18, A.19, we derive

$$\Gamma \vdash s[p] : E[\eta'] \mid s : h \setminus \langle p, \psi \rangle^{\phi} \rhd \{s[p] : \mathcal{E}[L'], s : \mathtt{h}' \cdot \mathtt{h}''\}$$

Let $\Delta' = \{s[p] : \mathcal{E}[L'], s : \mathtt{h}' \cdot \mathtt{h}''\}$. By $[\![$RcvDone$]\!]$ we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

and by Theorem 1, $\Gamma \vdash \Delta'$ is coherent.

- **(Rcv)**. Assume $\Gamma \vdash s[p] : E[q?\{l_i(x_i) : \eta_i\}_{i \in I}] \mid s : \langle q, p, l(v) \rangle \cdot h \rhd \Delta$ and $\Gamma \vdash \Delta$ is coherent.

  By applying Lemma 5.(15), we have $\Delta = \Delta_1, \Delta_2$

$$\Gamma \vdash s[p] : E[q?\{l_i(x_i) : \eta_i\}_{i \in I}] \rhd \Delta_1 \text{ and } \Gamma \vdash s : \langle q, p, l(v) \rangle \cdot h \rhd \Delta_2 \tag{A.20}$$

  By applying Lemma 7 and Equation A.20, we have

$$\Delta_2 = \{s : \langle q, p, l(v) \rangle \cdot \mathtt{h}\} \text{ and } \Gamma \vdash v : S \text{ and } \Gamma \vdash s : h \rhd \{s : \mathtt{h}\} \tag{A.21}$$

  By applying Lemma 5.(5) and Lemma 8 to Equation A.20 we have

$$\begin{aligned} &\Gamma \vdash s[p] : E[q?\{l_i(x_i) : \eta_i\}_{i \in I}] \rhd \{s[p] : \mathcal{E}[q?\ \{l_i(S_i).L_i\}_{i \in I}]\} \\ &\forall i \in I.\ \Gamma, x_i : S_i \vdash s[p] : E[\eta_i] \rhd \{s[p] : \mathcal{E}[L_i]\} \end{aligned} \tag{A.22}$$

  By coherence, applying Lemma 17 (Substitution) and Equation A.22,

$$\Gamma \vdash s[p] : E[\eta_k\{v_k/x_k\}] \rhd \{s[p] : \mathcal{E}[L_k]\} \tag{A.23}$$

  By $\lfloor$T-pa$\rfloor$ and Equation A.21 and Equation A.23, we derive

$$\Gamma \vdash s[p] : E[\eta_k\{v_k/x_k\}] \mid s : h \cdot \langle q, p, l(v) \rangle \rhd \{s[p] : \mathcal{E}[L_k], s : \mathtt{h}\}$$

  Let $\Delta' = \{s[p] : \mathcal{E}[L_k], s : \mathtt{h}\}$. By $[\![$Rcv$]\!]$, we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

  and by Theorem 1 $\Gamma \vdash \Delta'$ is coherent. We conclude this case.

- **(Rec)**. Assume $\Gamma \vdash s[p] : E[\mathsf{def}\ X(x) = \eta\ \mathsf{in}\ X\langle e\rangle] \rhd \Delta$.

  By applying Lemma 8 and Lemma 5.(8), we have

$$\begin{aligned} &\Delta = \{s[p] : L''\} \\ &L'' = \mathcal{E}[L] \\ &\Gamma, X : S\ \mu t.L' \vdash s[p] : X\langle e\rangle \rhd \{s[p] : L\} \\ &\Gamma, X : S\ t, x : S \vdash s[p] : \eta \rhd \{s[p] : L'\} \end{aligned} \tag{A.24}$$

  By applying Lemma 5.(7) to Equation A.24, we have

$$\Gamma' = \Gamma, X : S\ \mu t.L' \text{ and } \Gamma \vdash e : S \tag{A.25}$$

  By 5.(7) to Equation A.24 and Equation A.25, we have

$$\Gamma, X : S\ \mu t.L' \vdash s[p] : X\langle e\rangle \rhd \{s[p] : \mu t.L'\} \tag{A.26}$$

  By applying Lemma 17.I to Equation A.24 with Equation A.25 and Equation A.26, we have

$$\Gamma \vdash s[p] : E[\mathsf{def}\ X(x) = \eta\ \mathsf{in}\ \eta\{v/x\}] \rhd \Delta \text{ where } e \Downarrow v$$

  Thus we conclude this case.

- **(ClnDone)**. The proof is similar to the case **(Cln)**

- **(Crash)**. Assume $\Gamma \vdash s[p] : \eta \mid N \mid s : h \rhd \Delta$.

  By applying Lemma 5.(15) two times, we have

$$\begin{aligned} \Delta = \Delta_1, \Delta_2, \Delta_3 \text{ and } \Gamma \vdash s[p] : \eta \rhd \Delta_1 \text{ and} \\ \Gamma \vdash N \rhd \Delta_2 \text{ and } \Gamma \vdash s : h \rhd \Delta_3 \end{aligned} \tag{A.27}$$

  By applying Lemma 7 and $\lfloor$T-m$\rfloor$ to Equation A.27, we have

$$\Delta_3 = \{s : \mathtt{h}\} \text{ and } \mathtt{h} = \mathtt{h}_0 \cdot \mathtt{m}_1 \cdot \mathtt{h}_1 ... \mathtt{h}_{n-1} \cdot \mathtt{m}_n \cdot \mathtt{h}_n \text{ where } p \in \mathtt{m}_i, i \in \{1..n\} \tag{A.28}$$

  Let $msg(h, p)$ collect the messages to and from p. By applying Lemma 7 and $\lfloor$T-m$\rfloor$ to Equation A.28, we have

$$\Gamma \vdash s : \mathit{remove}(h, p) \rhd \{s : (\mathit{remove}(\mathtt{h}, p)\} \tag{A.29}$$

By $\lfloor$T-pa$\rfloor$ and $\lfloor$T-$F\rfloor$ to Equation A.27 and Equation A.29, we derive

$$\Gamma \vdash N \mid s : \mathit{remove}(h, p) \cdot \langle\!\langle \psi, p \rangle\!\rangle \rhd \tag{A.30}$$
$$\Delta_2, \{s : \mathit{remove}(\mathtt{h}, p) \cdot \langle\!\langle \psi, p \rangle\!\rangle\} \tag{A.31}$$

Let $\Delta' = \Delta_2, \{s : \mathtt{h} \setminus \mathit{msg}(\mathtt{h}, p) \cdot \langle\!\langle \psi, p \rangle\!\rangle\}$. By $[\![\mathtt{Crash}]\!]$ we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

and by Theorem 1, $\Gamma \vdash \Delta'$ is coherent. We conclude this case.

- **(SndDone)**. Assume $\Gamma \vdash s[p] : E[(0 \blacktriangleright H)^{\phi}.\eta'] \mid s : h \rhd \Delta$
  By applying Lemma 5.(15), we have

$$\begin{aligned} &\Delta = \Delta_1, \Delta_2 \text{ and} \\ &\Gamma \vdash s[p] : E[(0 \blacktriangleright H)^{\phi}.\eta'] \rhd \Delta_1 \text{ and} \\ &\Gamma \vdash s : h \rhd \Delta_2 \end{aligned} \tag{A.32}$$

  By applying Lemma 8 and Lemma 5.(9) and Lemma 5.(1) to Equation A.32, we have

$$\Delta_1 = \{s[p] : L\} \tag{A.33}$$
$$\begin{aligned} &L = \mathcal{E}[(\mathsf{end} \blacktriangleright H^{\downarrow})^{\phi}.L'] \text{ and} \\ &\Gamma \vdash s[p] : 0 \rhd \{s[p] : \mathsf{end}\} \text{ and } \Gamma \vdash s[p] : \eta' \rhd \{s[p] : L'\} \text{ and} \\ &\mathit{dom}(H) = \mathit{dom}(H^{\downarrow}) \text{ and} \\ &\forall F \in \mathit{dom}(H).\ \Gamma \vdash s[p] : H(F) \rhd \{s[p] : H^{\downarrow}(F)\} \end{aligned} \tag{A.34}$$

  By applying Lemma 7 and Equation A.32, we have

$$\Delta_2 = \{s : \mathtt{h}\} \tag{A.35}$$

  By $\lfloor$T-yd$\rfloor$ and $\lfloor$T-th$\rfloor$ and Equation A.33, we have

$$\Gamma \vdash s[p] : E[(\underline{0} \blacktriangleright H)^{\phi}.\eta'] \rhd \{s[p] : \mathcal{E}[(\underline{0} \blacktriangleright H^{\downarrow})^{\phi}.L']\} \tag{A.36}$$

  By $\lfloor$T-D$\rfloor$ and Equation A.35, we derive

$$\Gamma \vdash s : h \cdot \langle p, \psi \rangle^{\phi} \rhd \{s : \mathtt{h} \cdot \langle p, \psi \rangle^{\phi}\} \tag{A.37}$$

By $\lfloor$T-pa$\rfloor$ and Eqs. A.33, A.35, A.36, A.37, we derive

$$\Gamma \vdash s[p] : E[(\underline{0} \blacktriangleright H)^{\phi}.\eta'] \mid s : h \cdot \langle p, \psi \rangle^{\phi} \rhd \\ \{s[p] : \mathcal{E}[(\underline{0} \blacktriangleright H^{\downarrow})^{\phi}.L'], s : \mathtt{h} \cdot \langle p, \psi \rangle^{\phi}\}$$

Let

$$\Delta' = \{s[p] : \mathcal{E}[(\underline{0} \blacktriangleright H^{\downarrow})^{\phi}.L'], s : \mathtt{h} \cdot \langle p, \psi \rangle^{\phi}\}.$$

By $[\![$SndDone$]\!]$ we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h_d) \vdash \Delta'$$

and by Theorem 1, $\Gamma \vdash \Delta'$ is coherent. We conclude this case.

- **(F)**. Assume $\Gamma \vdash G : (F_q, h_d) \blacklozenge N \mid s : \langle\!\langle \psi, F' \rangle\!\rangle \cdot h \rhd \Delta$
  By applying Lemma 5.(17), we have

$$G : (F_q, h_d) \in \Gamma \text{ and } \Gamma' \vdash N \mid s : \langle\!\langle \psi, F' \rangle\!\rangle \cdot h \rhd \Delta \\ \text{such that } \Gamma = \Gamma', G : (F_q, h_d) \tag{A.38}$$

  By applying Lemma 5 and Lemma 7 and Equation A.38, we have

$$\Delta = \Delta_1, \Delta_2 \\ \Delta_2 = \{s : \langle\!\langle \psi, F' \rangle\!\rangle \cdot \mathtt{h}\} \text{ and } \Gamma' \vdash s : h \rhd \{s : \mathtt{h}\} \\ \Gamma' \vdash N \rhd \Delta_2 \tag{A.39}$$

  By $\lfloor$T-sys$\rfloor$,$\lfloor$T-pa$\rfloor$ and $\lfloor$T-$F\rfloor$ and Equation A.39, we derive

$$\Gamma', G : (F_q \cup F', h_d) \vdash G : (F_q \cup F', h_d) \blacklozenge N \mid \\ s : h \cdot \langle\!\langle \mathit{roles}(G) \setminus (F_q \cup F'), F' \rangle\!\rangle \cdot \rhd \\ \Delta_2, \{s : \mathtt{h} \cdot \langle\!\langle \mathit{roles}(G) \setminus (F_q \cup F'), F' \rangle\!\rangle\} \tag{A.40}$$

  Let

$$\Delta' = \Delta_2, \{s : \mathtt{h} \cdot \langle\!\langle \mathit{roles}(G) \setminus (F_q \cup F'), F' \rangle\!\rangle\}$$

  By $[\![$F$]\!]$ we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q \cup F', h_d) \vdash \Delta'$$

  and by Theorem 1, $\Gamma', G : (F_q \cup F', h_d) \vdash \Delta'$ is coherent. We conclude this case.

- **(CollectDone)**. The proof is trivial.

- **(IssueDone)**. Assume $\Gamma \vdash\ G : (F_q, h_d) \blacklozenge N \mid s : h \rhd\ \Delta$ and $\Gamma = \Gamma', G : (F_q, h_d)$ and there exists ϕ such that $\mathit{roles}(h_d, \phi) \supseteq \mathit{roles}(G, \phi) \setminus F_q$ and $\forall F \in hdl(G, \phi).(F \not\subseteq F_q)$

  By applying Lemma 5, we have

$$\begin{aligned} \Gamma' \vdash\ s : h \rhd\ \Delta_2 \\ \Delta = \{s : \mathtt{h}\} \\ \Gamma' \vdash\ N \rhd\ \Delta_1 \end{aligned} \tag{A.41}$$

  By applying Lemma 7 and Equation A.41 and $\lfloor$T-D$\rfloor$, we have

$$\Gamma' \vdash\ s : h \cdot \langle \psi, \mathit{roles}(G, \phi) \setminus (F_q) \rangle^{\phi} \rhd\ \{s : \mathtt{h} \cdot \langle \psi, \mathit{roles}(G, \phi) \setminus (F_q) \rangle^{\phi}\} \tag{A.42}$$

  By $\lfloor$T-sys$\rfloor$, $\lfloor$T-Pa$\rfloor$, Equation A.41 and Equation A.42, we derive

$$\begin{aligned} \Gamma', G : (F_q, \mathit{remove}(h_d, \phi)) \vdash\ G : (F_q, \mathit{remove}(h_d, \phi)) \blacklozenge N \mid \\ s : h \cdot \langle \psi, \mathit{roles}(G, \phi) \setminus (F_q) \rangle^{\phi} \\ \rhd \Delta_1, \{s : \mathtt{h} \cdot \langle \psi, \mathit{roles}(G, \phi) \setminus (F_q) \rangle^{\phi}\} \end{aligned} \tag{A.43}$$

  Let

$$\Delta' = \Delta_1, \{s : \mathtt{h} \cdot \langle \psi, \mathit{roles}(G, \phi) \setminus (F_q) \rangle^{\phi}\}$$

  By $[\![$IssueDone$]\!]$ we have

$$G : (F_q, h_d) \vdash \Delta \rightarrow_L G : (F_q, h'_d) \vdash\ \Delta'$$

  where $h'_d = \mathit{remove}(h_d, \phi)$ and by Theorem 1, $\Gamma, G : (F_q, h'_d) \vdash\ \Delta'$ is coherent. We conclude this case.

- **(Link)**. Assume $\Gamma \vdash\ a[p_1](y_1).P_1 \mid ... \mid a[p_n](y_n).P_n \rhd\ \Delta$ and $a : G$ and $\mathit{roles}(G) = \{p_1, ..., p_n\}$.

  By applying Lemma 5.(15) n times, we have

$$\Delta = \{\Delta_i\}_{i \in \{1..n\}} \text{ and } \forall i \in \{1..n\}.\ \Gamma \vdash\ a[p_i](y_i).P_i \rhd\ \Delta_i \tag{A.44}$$

By applying Lemma 5.(3) on Equation A.44, we have

$$\Delta_i = \emptyset \Rightarrow \Delta = \emptyset \tag{A.45}$$
$$\forall i \in \{1..n\}.\ \Gamma \vdash\ P_i \rhd\ \{y_i : G\upharpoonright p_i\} \text{ and } \Gamma \vdash\ a : \langle G \rangle$$

By applying $\lfloor$T-pa$\rfloor$ n times on Equation A.45, we derive

$$\Gamma \vdash\ P_1 \mid ... \mid P_n \rhd\ \{y_1 : G\upharpoonright p_1, ..., y_n : G\upharpoonright p_n\} \tag{A.46}$$

By applying Lemma 17.II to Equation A.46, we derive

$$\Gamma \vdash\ P_1\{s[p_1]/y_1\} \mid ... \mid P_n\{s[p_n]/y_n\} \rhd\ \{s[p_1] : G\upharpoonright p_1, ..., s[p_n] : G\upharpoonright p_n\} \tag{A.47}$$

By applying $\lfloor$T-pa$\rfloor$ n and $\lfloor$T-$\emptyset\rfloor$ to Equation A.47, we derive

$$\begin{aligned}\Gamma \vdash\ P_1\{s[p_1]/y_1\} \mid ... \mid P_n\{s[p_n]/y_n\} \mid s : \emptyset \rhd \\ \{s[p_1] : G\upharpoonright p_1, ..., s[p_n] : G\upharpoonright p_n, s : \emptyset\}\end{aligned} \tag{A.48}$$

By applying $\lfloor$T-sys$\rfloor$ to Equation A.48, we have

$$\begin{aligned}\Gamma, G : (\emptyset, \emptyset) \vdash\ G : (\emptyset, \emptyset) \blacklozenge P_1\{s[p_1]/y_1\} \mid ... \mid P_n\{s[p_n]/y_n\}\ \mid\ s : \emptyset \rhd \\ \{s[p_1] : G\upharpoonright p_1, ..., s[p_n] : G\upharpoonright p_n, \{s : \emptyset\}\}\end{aligned} \tag{A.49}$$

By applying Lemma 2 to Equation A.49, we have

$$\Gamma, G : (\emptyset, \emptyset) \vdash\ \{s[p_1] : G\upharpoonright p_1, ..., s[p_n] : G\upharpoonright p_n, \{s : \emptyset\}\} \quad \text{coherent} \tag{A.50}$$

By applying $\lfloor$T-s$\rfloor$ to Equation A.50 we derive

$$\Gamma, G : (\emptyset, \emptyset) \vdash\ (\nu s)(G : (\emptyset, \emptyset) \blacklozenge P_1\{s[p_1]/y_1\} \mid ... \mid P_n\{s[p_n]/y_n\}\ \mid\ s : \emptyset) \rhd\ \emptyset$$

Thus $\Delta' = \emptyset = \Delta$. We conclude this case.

For (b).

The proof is immediately because $\Gamma \vdash\ \emptyset$ is always coherent. □

B. Appendix for Chapter 4

B.1. Appendix: Overview

The contents of this supplementary appendix are structured as follows:

- Appendix B.2 provides the *full collection* of global types, local types (and CFSM representations), and processes for the Session-CM running example (cf. Sections 4.2 and 4.3).
- Appendix B.3 provides a detailed description of our system and failure model (cf. Section 4.3.3).
- Appendix B.4 corresponds to Section 4.4.1 (Global Types) and Section 4.4.2 (Local Types and Projection). It provides:
  - the formal definition of *global type well-formedness*;
  - and the projection of a *subprotocol onto a role set* ($g \upharpoonright R$) omitted from the main sections.
- Appendix B.5 corresponds to Section 4.4.3 (Endpoint Processes and Systems). It provides:
  - the omitted structural (parallel Par, restriction StrR and structural congruence StrR – all standard);*garbage collection* (Sess-GC and Root-GC) rules for system reduction;
  - and some minor auxiliary definitions ($\mathsf{fire}(L_1, L_2, b, \mathcal{F})$, $g^{\overline{q};p_0}\upharpoonright p$ and $s \rightsquigarrow^{+}_{\Theta}$).
- Appendix B.6 corresponds to Section 4.6.1 (Type System) and Section 4.6.2 (Properties). It provides
  - the definition of *coverage* ($\{L_i\}_{i\in 1..n} \vdash L$);
  - the full definition of *coherence* and the supporting infrastructure for the proofs;

– and the **proofs** of our properties: *subject reduction* (Appendix B.6.3), *fidelity* (Appendix B.6.4) and progress (Appendix B.6.5 – *subsession progress* and *global progress*).

- Appendix B.7 provides and discusses the global type of the *full version* of the Session-CM use case (as used in the performance evaluations of Section 4.7). The version of Session-CM presented in Sections 4.2 and 4.3 and Appendix B.2 is abridged for brevity in Chapter 4.

B.2. Session-CM: All Global Types, Local Types and Processes

This section provides all the global types, local types and example endpoint processes for our session-typed Spark cluster manager (Session-CM) example as presented in Sections 4.2 and 4.3 using the formal notation from Sections 4.4.1 to 4.4.3. We also provide the CFSM representations for all the local types as used internally by our Scala toolchain, as explained in Section 4.3.1.

B.2.1. Overview

Our framework starts from a user specification of the protocol as a *top-level global type*. The Scala version of the global type for Session-CM was given in Figure 4.2, and the formal notation version in Example 14. For completeness, we shall repeat the latter in Appendix B.2.2.

The following summarises how some of the names correspond between the two.

Scala notation	`RunDr`	`RunEx`	`Masters`	`m`	`Workers`	`wD`	`wE`	`InitDr` (etc.)
Formal notation	g_{RunDr}	g_{RunEx}	M	m	W	w_D	w_E	l_{InitDr} (etc.)

Here is a high-level outline of how local subprotocols are derived from the global subprotocols for each participant kind based on the generic role set and assigned role behaviors.

Global subprotocol	M participant kind		
	M generic role set	m role	
$g_{RunDr}(m; w_D; W)$	no explicit behavior	g_{RunDr_m}	
$g_{RunEx}(m, w_D; w_E; W)$	no explicit behavior	g_{RunEx_m}	

	W participant kind		
	W generic role set	w_D role	w_E role
$g_{RunDr}(m; w_D; W)$	g_{RunDr_W}	$g_{RunDr_{w_D}}$	n/a
$g_{RunEx}(m, w_D; w_E; W)$	g_{RunEx_W}	$g_{RunEx_{w_D}}$	$g_{RunEx_{w_E}}$

The top-level local type for each participant kind (i.e., M and W) corresponds to the collection of local subprotocols in each of those two columns above. More specifically, the top-level local types are calculated as follows (which shows where each local subprotocol comes from).

Top-level local type	Projection onto participant kind (Definition 18)	Projection onto role sets.. (Appendix B.4.1)	..*and* roles (Definition 19)
$\mathcal{L}_M$	$= \mathcal{G}_{\text{CM}} \upharpoonright M$	$= \{$	$g_{RunDr} \upharpoonright m,\ g_{RunEx} \upharpoonright m\}$
		$= \{$	$g_{RunDr_m},\ g_{RunEx_m}\}$
$\mathcal{L}_W$	$= \mathcal{G}_{\text{CM}} \upharpoonright W$	$= \{\, g_{RunDr} \upharpoonright W$	$g_{RunDr} \upharpoonright w_D,$
		$g_{RunEx} \upharpoonright W,$	$g_{RunEx} \upharpoonright w_D, g_{RunEx} \upharpoonright w_E\}$
		$= \{\, g_{RunDr_W},\ g_{RunEx_W}$	$g_{RunDr_{w_D}},\ g_{RunEx_{w_D}},\ g_{RunEx_{w_E}}\}$

Endpoint processes, i.e., event loops, are implemented for each participant kind – and static typing ensures the handlers of each event loop safely *cover* its local type. The definitions of every local subprotocol (and the internal CFSM representation used by our Scala toolchain) and example handlers for the event loops are given in the following subsections.

	M participant kind	W participant kind
Local subprotocols	Appendix B.2.3	Appendix B.2.3
(and CFSM representations)	Figure 4.4, Figure B.8	Figure B.6, Figure B.7, Figure B.9, Figure B.10, Figure B.11
Endpoint processes (i.e., event loops and handlers)	Appendix B.2.4	Appendix B.2.4

$$
\begin{array}{l}
g_{RunDr}(m; w_D; W) = \\
\quad m \to w_D\, l_{InitDr}\,. \\
\quad w_D \to m\, l_{Ack}\,. \\
\quad \mu \mathtt{t}\,. \\
\quad m \to w_D\{ \\
\qquad l_{AddEx} : g_{RunEx}(m, w_D; W; W)\,.\,\mathtt{t}, \\
\qquad l_{Ok} : \mathsf{end}\} \\
\mathsf{with} \\
\quad w_D @ m\,. \\
\quad m \to W\, l_{FailDr}\,. \\
\quad g_{RunDr}(m; W; W)\,. \\
\quad \mathsf{end}
\end{array}
$$

$$
\begin{array}{l}
g_{RunEx}(m, w_D; w_E; W) = \\
\quad m \to w_E\, l_{InitEx}\,. \\
\quad w_E \to m\, l_{ExDone}\,. \\
\quad m \to w_D\, l_{ExFinished}\,.\ \mathsf{end} \\
\mathsf{with} \\
\quad w_E @ m\,. \\
\quad m \to W\, l_{FailEx}\,. \\
\quad g_{RunEx}(m, w_D; W; W)\,. \\
\quad \mathsf{end}
\end{array}
$$

Figure B.1.: Global type of the Session-CM.

B.2.2. Global type of the Session-CM

We define the top-level global type of the Session-CM in Figure B.1. It aligns closely with the protocols defined in Figure 4.2. The main difference is that this global type has no messages with payload and it uses M and W for the role sets of the masters respectively workers.

B.2.3. Local types of the Session-CM

Master

We give the top-level local type of the M participant kind (`Masters` in Figure 4.2) in Figure B.2, projected from the global type via projection, see Definition 18. The top-level local type contains the protocols g_{RunDr_m} and g_{RunEx_m}, i.e. the projection of g_{RunDr} to the role m (`m` in Figure 4.2) and the projection of g_{RunEx} to m. There is no protocol for the generic role set M as no protocol uses the role set M. Both protocols contain the action describe in the global protocol whenever they involve m (which in this example are all actions).

Worker

We give the top-level local type of the participant kind W (`Workers` in Figure 4.2) in Figure B.3, projected from the global type via projection, see Definition 18. The top-level local type contains the protocols $g_{RunDr_{w_D}}$,g_{RunDr_W}, $g_{RunEx_{w_D}}$, $g_{RunEx_{w_E}}$ and g_{RunEx_W} i.e. the projection of g_{RunDr} to the role w_D and the generic role set W (`wD` and `Workers` in

$$
\begin{array}{l}
g_{RunDr_m}(m; w_D; W) = \{ \\
\quad w_D!l_{InitDr} \,. \\
\quad w_D?l_{Ack} \,. \\
\quad \mu \mathtt{t} \,.\, W!\{ \\
\qquad l_{AddEx} : g_{RunEx}(m, w_D; W; W) \,.\, \mathtt{t}, \\
\qquad l_{Ok} : \mathsf{end}\} \\
\mathsf{with} \\
\quad w_D{\downarrow} \,. \\
\quad W!l_{FailewD} \,. \\
\quad g_{RunDr}(m; W; W) \,. \\
\quad \mathsf{end}\}
\end{array}
\qquad
\begin{array}{l}
g_{RunEx_m}(m, w_D; w_E; W) = \{ \\
\quad w_E!l_{InitEx} \,. \\
\quad w_E?l_{ExDone} \,. \\
\quad w_D!l_{ExFinished} \,.\, \mathsf{end} \\
\mathsf{with} \\
\quad w_E{\downarrow} \,. \\
\quad W!FailwEx \,. \\
\quad g_{RunEx}(m, w_D; W; W) \,. \\
\quad \mathsf{end}\}
\end{array}
$$

Figure B.2.: Top-level local type of the participant kind M of the Session-CM.

Figure 4.2), and the projection of g_{RunEx} to the roles w_D, w_E and the generic role set W. **Note**, the failure handlings for w_D in g_{RunDr} (i.e., in $g_{RunDr_{w_D}}$) and w_E in g_{RunEx} (i.e., in $g_{RunDr_{w_E}}$) are "empty", despite the involvement of their role set W there, because a role canont participate in its own failure handling.

B.2.4. Endpoint process of the Session-CM

Endpoint process of Master

We provide an endpoint process which implements the participant kind M (`Masters` in Figure 4.2) in Figure B.4. This was first explained for g_{RunDr} in Example 16, corresponding very closely with Figure 4.6 (d). The event loop below contains the *full* set of handlers, i.e., including those for g_{RunEx}. Points worth highlighting are: the handlers contain no recursion, instead recursion is handled implicitly; Example 20, Chapter 4 provides further details on that; and that the selection in the protocol g_{RunDr} is expressed as two send handlers.

Endpoint process of Worker

We provide an endpoint process which implements the role set W (`Workers` in Figure 4.2) in Figure B.5. Points worth highlighting are: that branching in the protocol g_{RunDr} is expressed as two receiving handlers; and that e.g., $[m?l_{AddEx}]\lambda x \,.\, x[m]?l_{AddEx} \,.\, \mathsf{loop}$ implements the receiving of l_{AddEx} both for when p plays w_D or when p is just part of the role set W in the current session.

$$
\begin{array}{l}
g_{RunDr_W}(m; w_D; W) = \\
\quad \mu \mathtt{t} \,. \\
\quad m?\{ \\
\qquad l_{AddEx} : g_{RunEx}(m, w_D; W; W) \,.\, \mathtt{t}, \\
\qquad l_{Ok} : \mathsf{end}\} \\
\mathsf{with} \\
\quad m?l_{FailDr} \,. \\
\quad g_{RunDr}(m; W; W) \,. \\
\quad \mathsf{end}
\end{array}
\qquad
\begin{array}{l}
g_{RunDr_{w_D}}(m; w_D; W) = \\
\quad m?l_{InitDr} \,. \\
\quad m!l_{Ack} \,. \\
\quad \mu \mathtt{t} \,. \\
\quad m?\{ \\
\qquad l_{AddEx} : g_{RunEx}(m, w_D; W; W) \,.\, \mathtt{t}, \\
\qquad l_{Ok} : \mathsf{end}\} \\
\mathsf{with} \\
\quad \mathsf{end}
\end{array}
$$

$$
\begin{array}{l}
g_{RunEx_W}(m, w_D; w_E; W) = \\
\quad \mathsf{end} \\
\mathsf{with} \\
\quad m?l_{FailEx} \,. \\
\quad g_{RunEx}(m, w_D; W; W) \,. \\
\quad \mathsf{end}
\end{array}
\qquad
\begin{array}{l}
g_{RunEx_{w_D}}(m, w_D; w_E; W) = \\
\quad m?l_{ExFinished} \,.\, \mathsf{end} \\
\mathsf{with} \\
\quad m?l_{FailEx} \,. \\
\quad g_{RunEx}(m, w_D; W; W) \,. \\
\quad \mathsf{end}
\end{array}
\qquad
\begin{array}{l}
g_{RunEx_{w_E}}(m, w_D; w_E; W) = \\
\quad m?l_{InitEx} \,. \\
\quad m!l_{ExDone} \,. \\
\quad \mathsf{end} \\
\mathsf{with} \\
\quad \mathsf{end}
\end{array}
$$

Figure B.3.: Top-level local type of the participant kind W of the Session-CM.

$$
\left(
\begin{array}{c}
\texttt{/*RunDr normal activity*/} \\
[w_D!l_{InitDr}]\lambda x \,.\, x[w_D]!l_{InitDr} \,.\, \mathsf{loop} \\
[w_D?l_{Ack}]\lambda x \,.\, x[w_D]?l_{Ack} \,.\, \mathsf{loop} \\
[W!l_{AddEx}]\lambda x \,.\, x[W]!l_{AddEx} \,.\, \mathsf{loop} \\
[W!l_{Ok}]\lambda x \,.\, x[W]!l_{Ok} \,.\, \mathsf{loop} \\
[g_{RunEx}(m, w_D; W; W)]\lambda x \,.\, x[m, w_D; W; W](g_{RunEx}) \,.\, \mathsf{loop} \\
\texttt{/*RunDr failure handling*/} \\
[w_D{\downarrow} \,.\, W!l_{FailDr}]\lambda x \,.\, x[w_D]{\downarrow} \,.\, x[W]!l_{FailDr} \,.\, \mathsf{loop} \\
[g_{RunDr}(m; W; W)]\lambda x \,.\, x[m, w_D; W; W](g_{RunDr}) \,.\, \mathsf{loop} \\
\texttt{/*RunEx normal activity*/} \\
[w_E!l_{InitEx}]\lambda x \,.\, x[w_E]!l_{InitEx} \,.\, \mathsf{loop} \\
[w_E?l_{ExDone} \,.\, w_D!l_{ExFinished} \,.\,]\lambda x.Px[w_E]?l_{ExDone} \,.\, x[w_D]!l_{ExFinished} \,.\, \mathsf{loop} \\
\texttt{/*RunEx failure handling*/} \\
[w_E{\downarrow} \,.\, W!l_{FailwDr}]\lambda x \,.\, x[w_E]{\downarrow} \,.\, x[W]!l_{FailDr} \,.\, \mathsf{loop} \\
[g_{RunEx}(m; W; W)]\lambda x \,.\, x[m, w_D; W; W](g_{RunEx}) \,.\, \mathsf{loop}
\end{array}
\right) , \; p
$$

Figure B.4.: Endpoint process which implements the role set M of the Session-CM.

$$
\left(\begin{array}{c}
\texttt{/*RunDr normal activity*/} \\
[m?l_{InitDr}]\lambda x\,.\,x[m]?l_{InitDr}\,.\,\mathsf{loop} \\
[m!l_{Ack}]\lambda x\,.\,x[m]!l_{Ack}\,.\,\mathsf{loop} \\
[m?l_{AddEx}]\lambda x\,.\,x[m]?l_{AddEx}\,.\,\mathsf{loop} \\
[m?l_{Ok}]\lambda x\,.\,x[m]?l_{Ok}\,.\,\mathsf{loop} \\
[g_{RunEx}(m, w_D; W; W)]\lambda x\,.\,x[m, w_D; W; W](g_{RunEx})\,.\,\mathsf{loop} \\
\texttt{/*RunDr failure handling*/} \\
[m?l_{FailDr}]\lambda x\,.\,x[m]?l_{FailDr}\,.\,\mathsf{loop} \\
[g_{RunDr}(m; W; W)]\lambda x\,.\,x[m, w_D; W; W](g_{RunDr})\,.\,\mathsf{loop} \\
\texttt{/*RunEx normal activity*/} \\
[m?l_{InitEx}]\lambda x\,.\,x[m]?l_{InitEx}\,.\,\mathsf{loop} \\
[m!l_{ExDone}]\lambda x.Px[m]!l_{ExDone}\,.\,\mathsf{loop} \\
[m?l_{ExFinished}]\lambda x.Px[m]?l_{ExFinished}\,.\,\mathsf{loop} \\
\texttt{/*RunEx failure handling*/} \\
[m?l_{FailwEx}]\lambda x\,.\,x[W]?l_{FailEx}\,.\,\mathsf{loop} \\
[g_{RunEx}(m; W; W)]\lambda x\,.\,x[m, w_D; W; W](g_{RunEx})\,.\,\mathsf{loop}
\end{array}\right),\ p
$$

Figure B.5.: Endpoint process which implements the role set W of the Session-CM.

B.2.5. CFSM Representations of the local types of Session-CM

CFSM representations of `RunDr` local types. The CFSM representations of role m in `RunDr` were explained in Section 4.3.1. Figures B.6 and B.7 depict the two pairs of state machines for the protocol `RunDr` of the participant kind `Workers`. We have two pairs since `RunDr` involves the role `wD` belonging to `Workers`, and the generic role set behavior of (unassigned) `Workers`.

Figure B.6 depicts the state machine for the named role `wD`. It represents the following behavior: In the normal activity (a), `wD` first performs a receive from, then a send to, `m`. It then follows the choice of `m` between the spawn loop, i.e., receiving `Ack` from, `m`, and

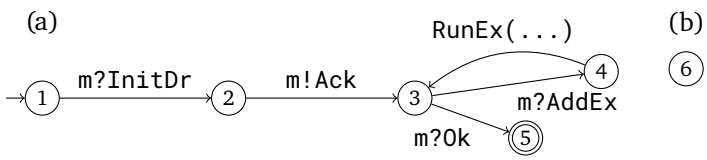

Figure B.6.: CFSM pair for the local type of role `wD` in `RunDr`, i.e., $g_{RunDr_{w_D}}$: (a) normal activity, and (b) failure handling. **Note**: (b) is empty because `wD` is not part of the failure handling for itself.

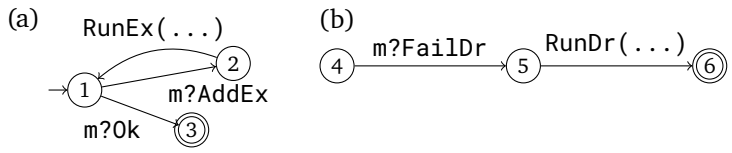

Figure B.7.: CFSM pair for the local type of the generic role set `Worker` in `RunDr`, i.e., g_{RunDr_W}: (a) normal activity, and (b) failure handling.

spawning `RunEx`, or the terminal case. **Note**, the failure handling (b), is empty since `wD` is not part of the failure handling for itself.

Figure B.7 depicts the state machine for the generic role set `Workers`. It represents the following behavior: In the normal activity (a), `Workers` first follows the choice of `m` between the spawn loop, i.e., receiving `Ack` from, `m`, and spawning `RunEx`, or the terminal case. In the failure handling (b), `Workers` receives the failure notification, then joins the spawning of the replacement `RunDr` subsession.

CFSM representations of `RunEx` local types. The CFSM representations for the local types of `RunEx` are for: role `m` (Figure B.8), role `wD` Figure B.10, role `wE` Figure B.9, and generic role set Figure B.11.

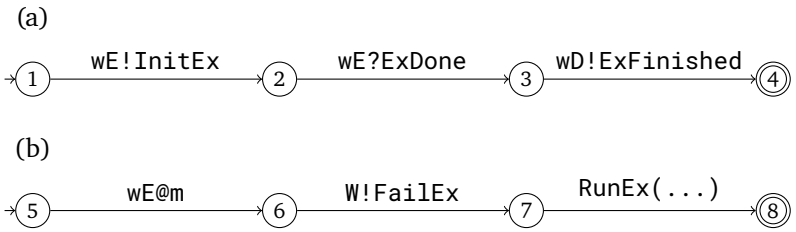

Figure B.8.: CFSM pair for the local type of `m` in `RunEx`, i.e., g_{RunEx_m}: (a) normal activity, and (b) failure handling.

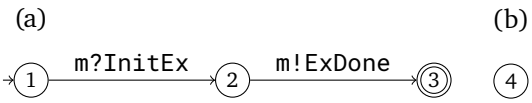

Figure B.9.: CFSM pair for the local type of `wE` in `RunEx`, i.e., $g_{RunEx_{w_E}}$: (a) normal activity, and (b) failure handling. **Note**: (b) is empty because `wE` is not part of the failure handling for itself.

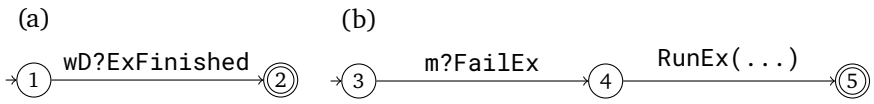

Figure B.10.: CFSM pair for the local type of `wD` in `RunEx`, i.e., $g_{RunEx_{w_D}}$: (a) normal activity, and (b) failure handling.

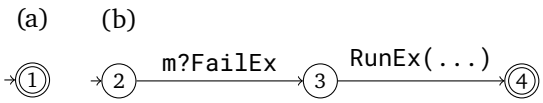

Figure B.11.: CFSM pair for the local type of generic role set `Workers` in `RunEx`, i.e., g_{RunEx_W}: (a) normal activity, and (b) failure handling.

B.3. System and Failure Model

This section clarifies our assumptions regarding distributed systems. First, we adopt the standard asynchronous semantics for session communication: (1) **Asynchronous system** [FLP85]: Application processes and the network are asynchronous, meaning that there are no upper bounds on processes' relative speeds or message transmission delays. (2) **Reliable message transport** [BCT96]: Messages transmitted between two peers are eventually delivered and in order of dispatch.

The above are achieved in practice by, e.g., TCP-based channels, MPI, various messaging middlewares, and "local" transports such as message buffers in shared memory (retransmissions etc. can overcome transient message losses). Message delivery can of course fail due to participants failing by *crashing*. A process which crashes stops performing any actions. To overcome the impossibility of agreement in asynchronous systems with such process crash stop failures [FLP85], our model distinguishes *failure-prone* roles (the default) from *robust* roles. A robust role is one that may be considered as failure-resilient from the perspective of the protocol or application. In this sense, standard (i.e., non-fault-tolerant) MPSTs are given by considering all roles robust, whereas our formalism has a minimal requirement of one robust role for our (progress) properties to hold.

Similarly, failure detection – crucial for fault-tolerance – in general *cannot* be achieved reliably in the considered system and failure model [CT96], due to the impossibility of distinguishing between a failed process and a slow(ly communicating) one. Our system is therefore built to deal with *false (failure) suspicions* to overcome these limitations: (3) **Peer-based failure detection and failure notification**: The failure of a failure-prone

role is determined by *one* of its (itself potentially failure-prone) peers. We say the latter *monitors* the former. Non-monitor roles only enter a failure handling activity if *explicitly* notified. (4) **Asynchronous and unreliable failure detection**: A monitor may deem a role as failed and initiate the corresponding *failure handling activity* at *any* point, i.e., non-deterministically and concurrently to any other interaction in the session. This judgment does *not* depend on whether the participant playing the role in question has actually failed or not. (5) **Consistent and monotonic failure suspicion**: A role behaves consistently in a session according to its current knowledge of peer failures, either following its normal activity or a failure handling activity. Once a role suspects a peer has failed, this belief is never reverted, regardless of the actual status of the suspected peer.

Items (1) and (2) and are standard in literature on fault-tolerant distributed systems [FLP85]. Items (3) and (4) reflect key contributions of Chapter 4 for MPSTs. Item (3) means our system completely abstracts from any particular failure detection mechanism, e.g., a heartbeat failure detectors could be used. (3) and (4) also mean that our system does not hide any implicit synchronization regarding failure detection or notification, and does not depend on failure detection being reliable. An important consequence is that our model permits false suspicion scenarios – a monitor and other peers may proceed as if some role has failed, while the role itself, and possibly some other set of peers, continue operating as normal. In practice a false suspicion is dealt with by *program-controlled crash*, which consists in communicating decisions to disregard supposedly failed processes also to those processes, prompting them to terminate themselves upon false suspicion [CHTCB96, CKV01], justifying (5). To be clear, our model is also fully compatible with reliable failure detection.

The above considerations are crucial to the applicability of our approach, and tie in with our observations of distributed programming and EDP in practice.

B.4. Global Types, Local Types, and Projection

This appendix section corresponds to Section 4.4.1 (Global Types, Local Types and Projection). It provides the omitted projection of a protocol to a role set (cf. Section 4.4.2).

Well-formedness. Global type syntax does not necessarily yield a sensible, i.e., safely realizable, distributed protocol. Firstly, as standard in MPSTs, our system only considers protocols for which projection (a partial function) is defined for every participant.

Before defining Well-Formedness, we define a helper predicate to check if the monitor informs all roles and roles sets correctly.

Definition 23 ($r; \overline{r}; \overline{R} \vdash G$)**.** *$r$ correctly notifying the roles $\overline{r}$ and role sets $\overline{R}$ in G, is defined as:*

$$\frac{r' \in \overline{r} \qquad r; \overline{r} \setminus r'; \overline{R} \vdash G}{r; \overline{r}; \overline{R} \vdash r \to r'\, l.\, G} \qquad \frac{R \in \overline{R} \qquad \forall i \in I. r; \overline{r} \setminus R; \overline{R} \setminus R \vdash G_i}{r; \overline{r}; \overline{R} \vdash r \to R\, l.\, G} \qquad r; \emptyset; \emptyset \vdash G$$

The following Well-Formedness definition checks that a *top-level* global type $\mathcal{G}$ ensures the following conditions for every subprotocol in $\mathcal{G}$: (1) the the global type following a monitor declaration ($r_1 @ r_2$) must start with the monitor (r_2) sending some label (i.e., failure notification) to all roles, except the monitored role (r_1), and roles sets in the subprotocol; (2) the set of message labels occurring in the normal activity is disjoint from that of the failure handling activity; (3) the set of subprotocol names occurring in the normal activity is disjoint from that of the failure handling activity; (4) spawned subprotocols are defined in the top-level global type; (5) and interactions and spawns only contain roles and role sets present in the arguments of the subprotocol.

Definition 24 (Well-Formedness)**.** *A* top-level *global type $\mathcal{G}$ is well-formed if it is projectable and all the following conditions hold for every subprotocol $g(\overline{r}; r_1; \overline{R}) = G_1$* with *$r_1 @ r\,.\, G_2$ in $\mathcal{G}$:*

- *(1) $r \in \overline{r}$ and $r; \overline{r} \setminus r; \overline{R} \vdash G_2$*
- *(2) $\forall r \to z\{l_i : G_i\}_{i \in I} \in G_1. \forall r' \to z'\{l_j : G_j\}_{j \in J} \in G_2.\ \{l_i\}_{i \in I} \cap \{l_j\}_{j \in J} = \emptyset$*
- *(3) $\forall g(\overline{r}; R; \overline{R}) \in G_1. \forall g'(\overline{r}'; R'; \overline{R}') \in G_2.\ g \neq g'$*
- *(4) $\forall g(r_1, .., r_n; R; R_1, .., R_m) \in G_i$ ($i \in \{1, 2\}$). $\exists g(r'_1, .., r'_n; r_0; R'_1, .., R'_m) = ...$ in $\mathcal{G}$ and $\exists R''_1, .., R''_n$ such that: $r_1 \in R''_1, r'_1 \in R''_1, .., r_n \in R''_n, r'_n \in R''_n$ and $r_0 \in R$ and $R_1 = R'_1, .., R_m = R'_m$*
- *(5) (a) $\forall r \to z\{l_i : G_i\}_{i \in I} \in G_j$ ($j \in \{1, 2\}$). $r, z \in \{\overline{r}; r_1; \overline{R}\}$ and (b) $\forall g(\overline{r}'; R'; \overline{R}') \in G_i$ ($i \in \{1, 2\}$). $\forall z \in \{\overline{r}'; R'; \overline{R}'\}.\ z \in \{\overline{r}; r_1; \overline{R}\}$*

B.4.1. Projection

We now define the projection of a global type to a role set.

Definition 25 (Projection $g \upharpoonright R$)**.** *Projection of a global type G to a role set R is defined as follows:*

$$
\begin{array}{rcl}
\big(g(\bar{r}; r_1; \overline{R}) = \\ G_1 \text{ with } r_1@r_2.G_2\big) \upharpoonright R & ::= & g_R(\bar{r}; r_1; \overline{R}) = G_1 \upharpoonright R \text{ with } G_2 \upharpoonright R \quad \textit{if } R \in \overline{R} \\
r \to z\{l_i : G_i\}_{i \in I} \upharpoonright R & ::= & \begin{cases} r?\{l_i : G_i \upharpoonright R\}_{i \in I} & \textit{if } R = z \\ G_1 \upharpoonright R & \textit{else if } \forall i, j \in I \; G_i \upharpoonright R = G_j \upharpoonright R \end{cases} \\
g(\bar{r}; R'; \overline{R}).G \upharpoonright R & ::= & \begin{cases} g(\bar{r}; R'; \overline{R}).(G \upharpoonright R) & \textit{if } R \in \overline{R} \textit{ or } R = R' \\ G \upharpoonright R & \textit{otherwise} \end{cases} \\
\mu t.G \upharpoonright R & ::= & \begin{cases} \mu t.(G \upharpoonright R) & \textit{if } G \upharpoonright R \neq t \\ \text{end} & \textit{otherwise} \end{cases} \qquad t \upharpoonright R ::= t \qquad \text{end} \upharpoonright R ::= \text{end}
\end{array}
$$

The first rule projects a protocol to the role set R if R is part of the role set parameters. The body of the protocol is the default activity G_1 projected onto R. Role sets are never the monitor nor are monitored; thus, for the failure handling activity only G_2, is projected to R. The next rule projects a choice to a role set R by either creating a branching statement $r?\{l_i : G_i \upharpoonright R\}_{i \in I}$ if R is the receiver (role sets may only occur at the receiving side) or continues by projecting G_1 to R if all branching cases projected to R are the same. We chose the traditional MPSTs restriction around branching (cf. [HYC16]) instead of a branch using merging (cf. [CYH09a]) to streamline formal development. The next rule projects a spawn; it is kept if R is either used to assign a role (R) or is one of the role set parameters ($\overline{R}$). Projection continues with G. Projection of recursion and end are as expected.

B.5. Endpoint Processes and Systems

This appendix section corresponds to Section 4.4.3 (Processes and Networks) and Section 4.5 (Operational Semantics).

B.5.1. Operational semantics

In the following we define the relation $\mathsf{fire}(L_1, L_2, b, \mathcal{F})$ used in the Fire reduction (cf. Figure 4.10).

Definition 26 (Valid guard type)**.** *A local type L is a* valid guard type *if (i) it is flat (the label sets of all selection and branching types in L contain exactly one label), (ii) it contains no* with *type, (iii) it contains no recursive types (i.e., no $\mu\mathtt{t}.$ or $\mathtt{t}$), and (iv) it contains no participant names.*

Definition 27 ($L \asymp L'$)**.** *The match predicate $L \asymp L'$ holds if and only if $L - L'$ is defined for these types.*

Definition 28 ($\mathsf{fire}(L, L_g, b, \mathcal{F})$)**.** *The $\mathsf{fire}(L, L_g, b, \mathcal{F})$ predicate, which checks that L_g will not block given the buffer b, the fail set $\mathcal{F}$, and L for binding of roles to participant names, is defined as:*

$$\mathsf{fire}(L, L_g, b, \mathcal{F}) \quad = \quad \begin{cases} \mathit{true} & \text{if } L_g = r!l\,.\,L' \\ \mathit{true} & \text{else if } L_g = r?l\,.\,L'_g \ \wedge\ b(p) = l\,.\,\bar{l} \text{ where } p{:}r \in L \\ \mathit{true} & \text{else if } L_g = r{\downarrow}\,.\,L'_g \ \wedge\ p \in \mathcal{F} \text{ where } p{:}r \in L \\ \mathit{false} & \text{otherwise} \end{cases}$$

Session and spawn semantics In the following we formally define relations used in Spawn, Sess-GC and Root-GC (cf. Figure 4.12) that we omitted in the main section.

Definition 29 (Projection $g^{\bar{q};p_0}\restriction p$)**.** *Given $\mathcal{G}$, $g(\bar{r}; r_0; \overline{R}) = \ldots \in \mathcal{G}$, p, and R such that $p \in R_{ids}$, and $\mathcal{L} = \mathcal{G} \restriction R$, then $g^{\bar{q};p_0}\restriction p$ is defined as:*

$$g^{\bar{q};p_0}\restriction p \quad = \quad \begin{cases} L\{\overline{q{:}r}, p_0{:}r_0/\bar{r}, r_0\} & \text{if } \bar{q} = q_1 \cdot \ldots \cdot q_n,\ p = q_j \text{ and } g_{r_j}(\bar{r}; r_0; \overline{R}) = L \in \mathcal{L} \\ L\{\overline{q{:}r}, p_0{:}r_0/\bar{r}, r_0\} & \text{else if } p = p_0 \text{ and } g_{r_0}(\bar{r}; r_0; \overline{R}) = L \in \mathcal{L} \\ L\{\overline{q{:}r}, p_0{:}r_0/\bar{r}, r_0\} & \text{else if } p \notin \bar{q} \cdot q_j \ \wedge\ g_R(\bar{r}; r_0; \overline{R}) = L \in \mathcal{L} \end{cases}$$

Definition 30. *The $\mathsf{done}(L)$ predicate is valid when $L = \mathsf{end}$ with L_2, $L = -$ with end, or $L = \mathsf{end}_{L_2}$*

Failure handling semantics. In the following we formally define the relations used in Mon and and RcvFN (cf. Figure 4.12) that we omitted from the main section.

Definition 31 ($s \rightsquigarrow^{+}_{\Theta}$)**.** *$s \rightsquigarrow^{+}_{\Theta}$ is defined as:*

$$s \rightsquigarrow^{+}_{\Theta} \quad = \quad \begin{cases} \widetilde{s} \cup (\bigcup_{s_i \in \widetilde{s}} (s_i \rightsquigarrow^{+}_{\Theta})) & \text{if } \Theta(s) = (\_, \widetilde{s}) \\ \emptyset & \text{else} \end{cases}$$

Omitted reduction rules

Figure B.12 provides reduction rules omitted from Figures 4.10, 4.12 and 4.13. The rules Par, StrR, and StrR in Figure B.12 are standard structural reduction rules. StrR uses a (simpler version of) standard structural equivalence.

$$\boxed{(\Theta, \mathcal{F}, N) \to (\Theta', \mathcal{F}', N')}$$

$$\textsc{Par}\ \frac{(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta_1', \mathcal{F}_1', N_1')}{(\Theta_1, \mathcal{F}_1, N_1 \mid\mid N_2) \to (\Theta_1, \mathcal{F}_1, N_1' \mid\mid N_2)}$$

$$\textsc{Res}\ \frac{(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta_2, \mathcal{F}_2, N_2)}{(\Theta_1, \mathcal{F}_1, (\nu(s : \mathcal{G}))\ N_1) \to (\Theta_2', \mathcal{F}_2', (\nu(s : \mathcal{G}))\ N_2)}$$

$$\textsc{StrR}\ \frac{N_1 \equiv N_1' \qquad (\Theta_1, \mathcal{F}_1, N_1') \to (\Theta_2, \mathcal{F}_2, N_2') \qquad N_2' \equiv N_2}{(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta_2, \mathcal{F}_2, N_2)}$$

Figure B.12.: Reduction rules omitted from Figures 4.10, 4.12 and 4.13

Definition 32 (Structural equivalence)**.** *StrR exercises structural equivalence for networks, defined by the rules below. Our calculus uses only the standard rules for rearranging parallel compositions and* 0*. We do not have rules for restriction, as we assume exactly one top-level restriction.*

$$N_1 \mid\mid N_2 \equiv N_2 \mid\mid N_1 \qquad (N_1 \mid\mid N_2) \mid\mid N_3 \equiv N_1 \mid\mid (N_2 \mid\mid N_3) \qquad N \mid\mid 0 \equiv N$$

B.6. Type System and Properties

B.6.1. Typing judgments

We now define coverage that is used in the typing rule TELoop (cf. Figure 4.14). We use $\{L_i\}_{i \in I} \vdash g$ as a short hand for $\{L_i\}_{i \in I} \vdash L$ where $g(\overline{r}; r; \overline{R}) = L \in \mathcal{L}$.

Definition 33 (Coverage $\{L_i\}_{i \in I} \vdash L$)**.** *Let* $\mathcal{L}$ *be the the minimal closed set under the relation* $\{(L_1, L_2) \mid L_j \in \{L_i\}_{i \in I}$ *and* $\mathsf{unf}(L_1) - L_j = L_2\}$ *that contains* L*. The event loop* $\{L_i\}_{i \in I}$ *covers the local type* L *if for all* L' *in* $(\{\mathsf{unf}(L_i) \mid L_1 \text{ with } L_2 \in \mathcal{L} \wedge L_i \neq \mathsf{end} \wedge i \in \{1, 2\}\} \cup \{\mathsf{unf}(L) \mid - \text{ with } L \in \mathcal{L} \wedge L \neq \mathsf{end}\})$ *exactly one of following is true:*

- *If* $L' = y?\{l_j : L_j\}_{j \in J}$*, then for all* $j \in J$ *it exists* $L'' \in \{L_i\}_{i \in I}$ *such that* $y?\{l_j : L_j\} \asymp L''$*.*
- *If* $L' = y!\{l_i : L_i\}_{i \in I}$*, then for all* $j \in J$ *it exists* $L'' \in \{L_i\}_{i \in I}$ *such that* $y!\{l_j : L_j\} \asymp L''$*.*
- *If* $L' = R!\{l_i : L_i\}_I$*, then for all* $j \in J$ *it exists* $L'' \in \{L_i\}_{i \in I}$ *such that* $R!\{l_j : L_j\} \asymp L''$*.*
- *If* $L' = y{\downarrow}.L$*, then it exists* $L'' \in \{L_i\}_{i \in I}$ *such that* $y{\downarrow}.L \asymp L''$*.*

- *If $L' = g(\overline{y}, R, \overline{R}).L$, then it exists $L'' \in \{L_i\}_{i \in I}$ such that $g(\overline{y}, R, \overline{R}).L \asymp L''$.*

The definition builds a minimal closed set using the relation $L_1 - L_2 = L_3$. For every element in that set, the definition ensures that all non-end, normal activities, and failure handling activities are covered by at least one event handler. For branching types and selection types, it further ensures that every case is covered. Furthermore, it ensures that for spawn types, there is a handler that only implements the spawn (by $L \asymp L'$ that builds on Definition 20).

Definition 34 ($\mathsf{wf}(L)$)**.** *A guard type L is well-formed ($\mathsf{wf}(L)$) if all the following conditions are valid.*

- *L is a valid guard type and*
- *if L contains a suspicion type, receive type or spawn type, then that type must be L*

Definition 34, defines that a well-formed guard type is flat and contains at most one type describing a blocking action, which must be the first action in the guard type.

B.6.2. Coherence

We now definition omitted definitions for coherence (cf. Section 4.6.1).

Coherence is the central typing invariant that ensures participant interactions remain safe throughout reduction – i.e., invariant properties of runtime networks used to establish subject reduction. There are two parts to our approach. The coherence (or consistency) property in MPSTs (e.g., [CDYP16]) is based on pairwise *duality* of endpoint types with consideration of input queue contents – duality is the intuitive compatibility relation between two participants, where an output on one side is balanced by an input on the other. First, we extend this concept as *intrasession* coherence, i.e., for individual (sub)sessions, to cater to our event-driven model, failure handling, and our multisend branching. Second, we introduce a notion of *intersession* coherence – properties *across* subsessions to ensure their concurrent execution and cross-session failure handling remain safe.

$L \upharpoonright p$ and partial types T A *partial type T* contains only the parts of L involving one other participant and a type for multisend receives.

Definition 35 (Partial Types)**.**

$$T \quad ::= \quad \{T \text{ with } T\} \mid \{-\text{with } T\} \mid !\{l_i : T_i\}_{i \in I} \mid ?\{l_i : T_i\}_{i \in I} \mid q\underline{?}\{l_i : T_i\}_{i \in I} \mid g.T \mid \mu \mathtt{t}.T \mid \mathtt{t} \mid \mathsf{end}$$

Auxiliary definition For ease of presentation, we assume all elements in a global type are uniquely numbered, and the numbers are carried to the local and partial types. The numbers allows us to identify local and partial types that are associated with a multisend send type (i.e., $r \to R\{l_i : G_i\}_{i \in I}$).

Partial projection $L \Downarrow p$ projects local type L to participant p, yielding a *partial type* T containing only the parts of L involving p.

Definition 36 (Partial Projection ($L \Downarrow p$)). *Partial projection of a local type L to a participant p is defined in Figure B.13.*

Partial projection is mostly standard ([CDYP16]) and closely follows the structure of (normal) projection. We therefore only explain some interesting rules. Partial projection of a with type or an active with type normally performs partial projection of the normal activity and failure handling activity. To simplify the next definitions, partial projection "activates" failure handling in the partial types if the queue contains a label that was sent in the failure handling activity (see duality for the justification). If we partial project to the monitored participant then partial projection sets the failure handling activity to end, because Definition 19 sets the failure handling activity of a monitored role to end.

The partial projection of a branching type has one nonstandard case (cf. [CDYP16]). If we partial project a branching type ($q{:}r?\{l_i : L_i\}_{i \in I}$) associated with (was projected from) a multisend to R ($r \to R\{l_i : G_i\}_{i \in I}$) and the partial projection is towards a participant p that belongs to R (i.e.,$r, p \in R_{ids}$) then we create a multisend receive partial type ($q\underline{?}\{l_i : L_i \Downarrow p\}_{i \in I}$). The dual of a partial multisend receive type is another partial multisend receive type. While we do not have an output is balanced by an input we have that both sides wait for (or have received) a multisend label, i.e., if any participant of R received a label, say $l_j \in \{l_i : L_i\}_{i \in I}$ then all other (unsuspected) participant of R also received the label l_j.

Session remainder for partial types and queue Before defining the session remainder for partial types we define a helper definition that removes partial multisend receive types for which we have a multisend label in the local queue.

Definition 37 ($T \setminus b$). *The removal of multisend labels b from a partial type T is defined as*

$$
\begin{array}{rcl}
\mathsf{end}_L \restriction p & ::= & L \restriction p \\[2ex]
L \text{ with } L' \restriction p & ::= & \begin{cases} \{-\text{with } L' \restriction p\} & \text{if any failure handing label} \\ & \quad \text{is queued in the configuration} \\ \{L \restriction p \text{ with } \mathsf{end}\} & \text{else if } p \text{ is the monitored participant} \\ \{L \restriction p \text{ with } L' \restriction p\} & \text{else} \end{cases} \\[2ex]
- \text{ with } L \restriction p & ::= & \begin{cases} \{-\text{with } \mathsf{end}\} & \text{if } p \text{ is the monitored participant} \\ \{-\text{with } L \restriction p\} & \text{else} \end{cases} \\[2ex]
p{:}r'?\{l_i : L_i\}_{i \in I} \restriction p & ::= & \begin{cases} ?\{l_i : L_i \restriction p\}_{i \in I} & \text{if } p = q \\ q\underline{?}\{l_i : L_i \restriction p\}_{i \in I} & \text{else if associated with } r \to R\{l_i : G_i\}_{i \in I} \\ & \quad \text{and } p \in R_{ids} \\ L_1 \restriction p & \text{else if } L_i \restriction p = L_j \restriction p \end{cases} \\[2ex]
q{:}r{\downarrow}.L \restriction p & ::= & \begin{cases} \mathsf{end} & \text{if } p = q \\ L \restriction p & \text{else} \end{cases} \\[2ex]
q{:}r!\{l_i : L_i\}_{i \in I} \restriction p & ::= & \begin{cases} !\{l_i : L_i \restriction p\}_{i \in I} & \text{if } p = q \\ L_1 \restriction p & \text{else if } L_i \restriction p = L_j \restriction p \end{cases} \\[2ex]
R!\{l_i : L_i\}_{i \in I} \restriction p & ::= & \begin{cases} !\{l_i : L_i \restriction p\}_{i \in I} & \text{if } p \in R_{ids} \\ L_1 \restriction p & \text{else if } L_i \restriction p = L_j \restriction p \end{cases} \\[2ex]
g(\overline{q{:}r}, R, \overline{R}).L \restriction p & ::= & \begin{cases} g.(L \restriction p) & \text{if } p \in \mathsf{pid}_\Theta(\overline{q{:}r}, R, \overline{R}) \\ L \restriction p & \text{else} \end{cases}
\end{array}
$$

$$
\mu t.L \restriction p = \begin{cases} \mu t.(L \restriction p) & \text{if } (L \restriction p) \neq t \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad t \restriction p = t \qquad \mathsf{end} \restriction p = \mathsf{end}
$$

Figure B.13.: Partial projection.

follows:

$$\frac{\forall i \in \{l_i\}.\ T_i \setminus b = T_i'}{!\{l_i : T_i\}_{i\in I} \setminus b = !\{l_i : T_i'\}_{i\in I}} \qquad \frac{\forall i \in \{l_i\}.\ T_i \setminus b = T_i'}{?\{l_i : T_i\}_{i\in I} \setminus b = ?\{l_i : T_i'\}_{i\in I}} \qquad \frac{\textit{no other rule applicable}}{T \setminus b = T}$$

$$\frac{b(q) = l_j \cdot \bar{l} \qquad l_j \in \{l_i\} \qquad b' = b[q \mapsto \bar{l}] \qquad T_j \setminus b' = T'}{q\underline{?}\{l_i : T_i\}_{i\in I} \setminus b = T'}$$

$$\frac{T\{\mu t.T/t\} \setminus b = T' \qquad T\{\mu t.T/t\} \neq T'}{\mu t.T \setminus b = T'} \qquad \frac{T \setminus b = T'}{g.T \setminus b = g.T'}$$

$$\frac{T_1 \setminus b = T_1'}{T_1 \text{ with } T_2 \setminus b = T_1' \text{ with } T_2} \qquad \frac{b' = \textit{clean default labels from } b \qquad T \setminus b' = T'}{- \text{ with } T \setminus b = - \text{ with } T'}$$

Definition 38 (multiSndLabels(b, R))**.** *Extracting the multisend labels sent to R from b is defined as follows:*

$$\begin{aligned} \textit{multiSndLabels}(b, R) &= [p_1 \mapsto m_{lbl}(b(p_1)), ..., p_n \mapsto m_{lbl}(b(p_n))] \quad \textit{where}\{p_1, .., p_n\} = \textit{dom}(b) \\ m_{lbl}(\bar{l}) &= \begin{cases} \epsilon & \textit{if } \bar{l} = \epsilon \\ m_{lbl}(\bar{l}') & \textit{else if } \bar{l} = l \cdot \bar{l}' \ \wedge\ l \textit{ is not a mulitsend label to } R \\ l \cdot m_{lbl}(\bar{l}') & \textit{else if } \bar{l} = l \cdot \bar{l}' \end{cases} \end{aligned}$$

We now define the partial type remainder definitions.

Definition 39 ($T - b(q)$)**.** *Let T be p's partial type and (a) $b' = \textit{multiSndLabels}(b, R)$ if $p, q \in R_{ids}$ else (b) $b' = []$; then the partial type remainder $T - b(q)$ first calculates $T \setminus b' = T'$*

and then calculates $T' - \bar{l}$ via the rules defined below.

$$\frac{T_j - \bar{l} = T' \qquad l_j \in \{l_i\}}{?\{l_i : T_i\}_{i \in I} - l_j \cdot \bar{l} = T'} \qquad \frac{T\{\mu t.T/t\} - \bar{l} = T'}{\mu t.T - \bar{l} = T'} \qquad \frac{}{T - \epsilon = T}$$

$$\frac{T_1 - l_1 = T' \qquad T' \text{ with } T_2 - \bar{l} = T''}{T_1 \text{ with } T_2 - l_1 \cdot \bar{l} = T''} \qquad \frac{T_2 - l_1 = T' \qquad - \text{ with } T' - \bar{l} = T''}{T_1 \text{ with } T_2 - l_1 \cdot \bar{l} = T''}$$

$$\frac{T - l_1 = T_1 \qquad - \text{ with } T_1 - \bar{l} = T'}{- \text{ with } T - l_1 \cdot \bar{l} = T'} \qquad \frac{l_1 \notin T \qquad - \text{ with } T - \bar{l} = T'}{- \text{ with } T - l_1 \cdot \bar{l} = T'}$$

$$\frac{l_1 \in T_1 \qquad \exists l' \in \bar{l}.\ l' \in T_2 \qquad T_1 \text{ with } T_2 - \bar{l} = T''}{T_1 \text{ with } T_2 - l_1 \cdot \bar{l} = T''}$$

Notice that Definition 39 (-) is not defined for spawn. If b and q is clear from context we sometimes write $T - \bar{l}$ instead of $T - b(q)$ for $b(q) = \bar{l}$

Definition 40 (Duality)**.** *$T \bowtie T'$ of partial types is the minimal symmetric relation that satisfies:*

$$T \bowtie T' \wedge T_H \bowtie T'_H \Rightarrow T \text{ with } T_H \bowtie T' \text{ with } T'_H \qquad T_H \bowtie T'_H \Rightarrow T' \text{ with } T'_H \bowtie - \text{ with } T_H$$
$$T_H \bowtie T'_H \Rightarrow - \text{ with } T'_H \bowtie - \text{ with } T_H \qquad T \bowtie T' \Rightarrow g.T \bowtie g.T'$$
$$\forall i \in I.\ T_i \bowtie T'_i \ \wedge\ l_i = l'_i \ \Rightarrow\ !\{l_i : T_i\}_{i \in I} \bowtie ?\{l'_i : T'_i\}_{i \in I}$$
$$\forall i \in I.\ T_i \bowtie T'_i \ \wedge\ l_i = l'_i \ \Rightarrow\ q\underline{?}\{l_i : T_i\}_{i \in I} \bowtie q\underline{?}\{l'_i : T'_i\}_{i \in I}$$
$$T_1 \bowtie T_2 \Rightarrow \mu \mathtt{t}.T_1 \bowtie \mu \mathtt{t}.T_2, \mathtt{t} \bowtie \mathtt{t} \quad \text{end} \bowtie \text{end}$$

Two *partial handling types* (T with T_H) are dual if their normal and failure handling activities are respectively dual. A partial handling type and an *active partial handling type* ($-$ with T_H) are dual if just their failure handling activities are dual: unless an ancestor session takes over, all participants will eventually converge on the active failure handling and labels from the normal activity will be cleaned. Two *partial spawn types* ($g.T$) are dual if they spawn the same subprotocol and their continuations are dual. Two *partial multisend receive types* are dual if both receive the same labels and their continuations are dual. Duality between send and receive is standard, as is recursion and end.

Definition 41 (Unfold)**.** *For local types L unfold* $\mathsf{unf}(L)$ *is defined as follows:*

$\mathsf{unf}(L_1)$ with $\mathsf{unf}(L_2)$	*if* $L = L_1$ with L_2	$\mathsf{unf}(L_1\{\mu t\ L_1/t\})$	*if* $L = \mu t\ L_1$
$-$ with $\mathsf{unf}(L_1)$	*if* $L = -$ with L_1	L	*otherwise*

Unfold for partial types is defined analogue to unfold for local types.

Coherence. We now define our notions of coherence based on the above elements.

Definition 42 (Intrasession Coherence)**.** *Assume a well-formed top-level $\mathcal{G}$, $(\Delta, \Sigma, \mathcal{F}, \Theta)$ and a session s in Θ. Then $(\Delta, \Sigma, \mathcal{F}, \Theta)$ satisfies* intrasession coherence *w.r.t. s if all the following hold:*

1. *For all $s[p_1] : (L_1, b_1)$, $s[p_2] : (L_2, b_2) \in \Sigma$ with p_1, p_2 unsuspected the following must hold: $\mathsf{unf}(L_1 \upharpoonright p_2 - b_1(p_2)) \bowtie \mathsf{unf}(L_2 \upharpoonright p_1 - b_2(p_1))$.*

2. *For all $s[p] : (L, b) \in \Sigma$ with p unsuspected (i.e., $p \notin \mathcal{F}$) and L not stopped ($L \neq \mathsf{end}_{L''}$): $p{:}R : \{L_i\}_{i \in I} \in \Delta$, and either $s[p] \notin \Delta \wedge \{L_i\}_{i \in I} \vdash L$ or $s[p] : L' \in \Delta \wedge \{L_i\}_{i \in I} \vdash L - L'$.*

3. *All configurations for s are present and respects well-formedness under $\mathcal{G}$.*

(1) checks for all pairs of unsuspected participants that the local types and queues are dual. (2) checks the event loop of every unsuspected participant provides coverage for its current subprotocol state L, with consideration of the active handler ($L - L'$) if any. (3) is basic bookkeeping: it records that configurations respect well-formedness and global types restrictions, e.g., normal and failure handling activities use disjoint labels and ensures that all input buffers, in particular from suspected participants, are present, a minor technicality to ensure Send reduction (cf. Figure 4.10) does not block due to a missing buffer.

A *stopped configuration* has a stopped subprotocol state (end_L). A configuration for $s[p]$ is an *ancestor* (resp. *descendent*) of a configuration for $s'[p]$ if s is an ancestor (resp. descendent) of s'.

Definition 43 (Intersession Coherence)**.** $(\Delta, \Sigma, \mathcal{F}, \Theta)$ *satisfies* intersession coherence *if both:*

1. *The parent-child subsession relation in Θ forms exactly one tree, and the participants of every child session are a subset or equal to those of its parent.*

2. *For all unsuspected participants, every stopped configuration has a non-stopped ancestor configuration with active failure handling. All descendent configurations of a configuration with active failure handling are stopped.*

Definition 44 (Coherence)**.** *Assuming a well-formed top-level $\mathcal{G}$, $(\Delta, \Sigma, \mathcal{F}, \Theta)$ satisfies* coherence *if (i) it satisfies intrasession coherence for all $s \in \Theta$ and (ii) it satisfies intersession coherence. We may simply say $(\Delta, \Sigma, \mathcal{F}, \Theta)$ is* coherent.

Well-Formedness of Configurations Lastly we defines the meaning of well-formedness for configuration types. It is a direct adaption of Definition 24 to the subprotocol states and the buffer types.[1]

Definition 45 (Configuration Well-Formedness)**.** *Assume* $(\Delta, \Sigma, \mathcal{F}, \Theta)$*, a session* s*, with* $s \in \Theta$ *and a well-formed top-level* $\mathcal{G}$*. Then the configurations of* s *respects to well-formedness under* $\mathcal{G}$ *if all the following conditions are satisfied:*

1. *The monitor must send at the beginning of the failure handling (after a potential failure detection) a label to all roles and roles sets except: the monitored role; and roles or roles sets which are already notified about the failure, i.e. have active failure handling or the label in the queue which will trigger failure handling.*
2. *The set of labels used in the default activity and the set of labels used in the failure handling activity must be distinct. This extends to queue types, i.e. a queue type can only contain a label associated with failure handling if the sender has active failure handling.*
3. *The set of protocols spawned in the default activity and the failure handling activity must be distinct.*
4. *The sender and the receiver must be distinct for all interaction.*

B.6.3. Property: subject reduction

Reduction of typing environments

Figure B.14 – Figure B.16 define the typing environment reduction rules. They are closely aligned with the network reduction rules (cf. Figure 4.10 – Figure 4.13). For brevity we omitted Γ from the reduction and implicitly assume the presence of $\mathcal{G}$ associated with the root session. Some rules contain the premise $\nexists s'[p] \notin \Delta_1$ that ensures that the participant p has no active event handling. We *overloaded* $\Delta = \Delta_1 \cdot \Delta_{\mathsf{end}}$ when it occurs in the *premise* of the type environment reduction as follows: it states that Δ is split into the environments Δ_1 and Δ_{end} where Δ_1 contains *no* end-only session.

Preservation of coherence

In this section, we show the preservation of coherence an important building block for subject reduction. We now define supporting definitions and lemmas. We write $\Sigma \setminus s$ and $\Delta \setminus s$ with the meaning of removing all $s[p]$ contained in Σ respectively Δ.

[1] Definition 49 (Extend global type well-formedness) could be used to make this definition more rigid.

[[FIRE]]

$$\frac{L_g \in \{L_i\}_{i\in I} \qquad L \asymp L_g \qquad \mathsf{fire}(L, L_g, b, \mathcal{F}) \qquad \Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \qquad \nexists s'[p] \notin \Delta_1}{\begin{array}{c}(\Delta,\ s[p] : (L, b) \cdot p{:}R : \{L_i\}_{i\in I} \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \\ (s[p] : L_g \cdot \Delta,\ s[p] : (L, b) \cdot p{:}R : \{L_i\}_{i\in I} \cdot \Sigma,\ \mathcal{F},\ \Theta)\end{array}}$$

[[SEND]]

$$\frac{l_j \in \{l_i\}_{i\in I}}{\begin{array}{c}(s[p] : r!l_j.L \cdot \Delta,\ s[p] : (\mathcal{E}[q{:}r!\{l_i : L_i\}_{i\in I}], b) \cdot s[q] : (L_q, b_q) \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \\ (s[p] : L \cdot \Delta,\ s[p] : (\mathcal{E}[L_j], b) \cdot s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_1]) \cdot \Sigma,\ \mathcal{F},\ \Theta)\end{array}}$$

[[RECV]] $$\frac{(q \mapsto l_j \cdot \bar{l}) \in b \qquad l_j \in \{l_i\}_{i\in I}}{\begin{array}{c}(s[p] : r?l_j.L \cdot \Delta,\ s[p] : (\mathcal{E}[q{:}r?\{l_i : L_i\}_{i\in I}], b) \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \\ (s[p] : L \cdot \Delta,\ s[p] : (\mathcal{E}[L_j], b[q \mapsto \bar{l}]) \cdot \Sigma,\ \mathcal{F},\ \Theta)\end{array}}$$

[[MSEND]]

$$\frac{R_{ids} \setminus \mathcal{F} = \widetilde{p} \qquad l_j \in \{l_i\}_{i\in I} \qquad \forall q \in \widetilde{p}.\ b'_q = b_q[p \mapsto b_q(p) \cdot l_j]}{\begin{array}{c}(s[p] : R!l_j.L \cdot \Delta,\ s[p] : (\mathcal{E}[R!\{l_i : L_i\}_{i\in I}], b) \cup_{q\in\widetilde{p}} \{s[q] : (L_q, b_q)\} \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \\ (s[p] : L \cdot \Delta,\ s[p] : (\mathcal{E}[L_j], b) \cup_{q\in\widetilde{q}} \{s[q] : (L_q, b'_q)\} \cdot \Sigma,\ \mathcal{F},\ \Theta)\end{array}}$$

[[UNFOLD]]

$$\frac{\Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \qquad \nexists s'[p] \notin \Delta_1}{(\Delta,\ s[p] : (\mathcal{E}[\mu\mathtt{t}.L], b) \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \qquad (\Delta,\ s[p] : (\mathcal{E}[L\{\mu\mathtt{t}.L/\mathtt{t}\}], b) \cdot \Sigma,\ \mathcal{F},\ \Theta)}$$

Figure B.14.: Reduction of typing environments.

[[SPAWN]]

$$\frac{\begin{array}{c}\widetilde{p}_1 = \bar{q} \cup (R'_{ids} \cup \overline{R}_{ids}) \setminus \mathcal{F} \qquad \widetilde{p}_2 = \bar{q} \cup p_0 \cup (\overline{R}_{ids} \setminus \mathcal{F}) \qquad \widetilde{p}_2 \cap \mathcal{F} = \emptyset \qquad p_0 \in R'_{ids} \qquad p_0 \notin \bar{q} \\ s' \text{ fresh} \qquad \Theta' = \Theta[s \mapsto (\widetilde{p}_2, \emptyset), s \mapsto (\widetilde{p}_1, \widetilde{s} \cup s) \qquad \Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \qquad \forall p \in \widetilde{p}_1.\ \nexists s[p] \notin \mathtt{dom}(\Delta_1)\end{array}}{\begin{array}{c}(\Delta,\ \{p{:}R : \{L_i\}_{i\in I_p}\}_{p\in\widetilde{p}_1} \cup_{p\in\widetilde{p}_1} s[p] : (\mathcal{E}_p[g(\overline{q{:}r}; R'; \overline{R}).L_p], b_p) \cdot \Sigma,\ \mathcal{F},\ \Theta[s \mapsto (\widetilde{p}, \widetilde{s})]) \qquad \rightarrow \\ (\Delta; \{p{:}R : \{L_i\}_{i\in I_p}\}_{p\in\widetilde{p}_1} \cup_{p\in\widetilde{p}_1} \{s[p] : (\mathcal{E}_p[L_p], b_p)\} \cup_{p\in\widetilde{p}_2} \{s'[p] : (g^{\bar{q}:p_0}\lceil p, \varepsilon)\} \cdot \Sigma, \Theta')\end{array}}$$

[[SESS-GC]]

$$\frac{\begin{array}{c}(\widetilde{p}_1, \widetilde{s}) = \Theta(s_1) \qquad (\widetilde{p}_2, \emptyset) = \Theta(s_2) \\ \widetilde{p} = \widetilde{p}_2 \setminus \mathcal{F} \qquad s_2 \in \widetilde{s} \qquad \forall p \in \widetilde{p}.\ \mathsf{done}(L_p) \qquad \Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \qquad \forall p \in \widetilde{p}.\ \nexists s[p] \notin \mathtt{dom}(\Delta_1)\end{array}}{\begin{array}{c}(\Delta,\ \{p{:}R : \{L_j\}_{j\in J}\}_{p\in\widetilde{p}} \cup_{p\in\widetilde{p}} \{s_2[p] : (L_p, b_p)\} \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \\ (\Delta,\ \{p{:}R : \{L_j\}_{j\in J}\}_{p\in\widetilde{p}} \cdot \Sigma,\ \mathcal{F},\ \Theta[s_1 \mapsto (\widetilde{p}_1, \widetilde{s} \setminus s_2)] \setminus s_2)\end{array}}$$

[[ROOT-GC]]

$$\frac{\begin{array}{c}(\widetilde{p}_1, \emptyset) = \Theta(s) \\ \widetilde{p} = \widetilde{p}_1 \setminus \mathcal{F} \qquad \forall_{p\in\widetilde{p}}\ \mathsf{done}(L_p) \qquad s \text{ is root} \qquad \Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \qquad \forall p \in \widetilde{p}.\ \nexists s[p] \notin \mathtt{dom}(\Delta_1)\end{array}}{(\Delta,\ \{p{:}R : \{L_i\}_{i\in I}\}_{p\in\widetilde{p}} \cup_{p\in\widetilde{p}} \{s[p] : (L_p, b_p)\} \cdot \Sigma,\ \mathcal{F},\ \Theta) \qquad \rightarrow \qquad (\Delta,\ \Sigma,\ \mathcal{F},\ \Theta \setminus s)}$$

Figure B.15.: Sub session related reduction of typing environments.

$$[\![\text{MON}]\!]\ \frac{p' \in \mathcal{F} \qquad s \rightsquigarrow_{\Theta}^{+} = \{s_i\}_{i\in I}}{\begin{array}{c}(s[p] : q{\downarrow}.L_g, \Delta,\ s[p] : (L' \text{ with } p'{\downarrow}.L, b) \cup_{i\in I}\ \{s_i[p] : (L_i, b_i)\} \cdot \Sigma,\ \mathcal{F},\ \Theta) \\ (s[p] : L_g, \Delta,\ s[p] : (- \text{ with } L, b) \cup_{i\in I}\ \{s_i[p] : (\mathsf{end}_{L_i}, b_i) \cdot \Sigma,\ \mathcal{F},\ \Theta)\end{array}} \rightarrow$$

$$[\![\text{RECV-HDL}]\!]\ \frac{(q \mapsto l \cdot \bar{l}) \in b \qquad s[p] \rightsquigarrow_{\Theta}^{+} = \{s_i\}_{i\in I}}{\begin{array}{c}(s[p] : r?l.L_g, \Delta,\ s[p] : (L'' \text{ with } q{:}r?l.\,L, b) \cup_{i\in I}\ \{s_i[p] : (L_i, b_i)\} \cdot \Sigma,\ \mathcal{F},\ \Theta) \\ (s[p] : L_g, \Delta,\ s[p] : (- \text{ with } L_j, b[q \mapsto \bar{l}]) \cup_{i\in I}\ \{s_i[p] : (\mathsf{end}_{L_i}, b_i)\} \cdot \Sigma,\ \mathcal{F},\ \Theta)\end{array}} \rightarrow$$

$$[\![\text{CLEAN}]\!]\ \frac{\begin{array}{c}b(q) = l_i \cdot \bar{l} \qquad (L = - \text{ with } L' \ \wedge\ l_i \notin L') \vee (L = L' \text{ with } L'' \ \wedge\ l_i \in L' \ \wedge\ \exists l' \in \bar{l}.\ l' \in L'') \\ \Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \qquad \nexists s'[p] \notin \mathtt{dom}(\Delta_1)\end{array}}{(\Delta,\ p{:}R : \{L_i\}_{i\in I} \cdot s[p] : (L, b) \cdot \Sigma,\ \mathcal{F},\ \Theta) \rightarrow (\Delta,\ p{:}R : \{L_i\}_{i\in I} \cdot s[p] : (L, b[q \mapsto \bar{l}]) \cdot \Sigma,\ \mathcal{F},\ \Theta)}$$

$$[\![\text{SUSP}]\!]\ \frac{p \notin \mathcal{F}}{(\Delta,\ \Sigma,\ \mathcal{F},\ \Theta) \quad \rightarrow \quad (\Delta,\ \Sigma,\ \mathcal{F} \cup p,\ \Theta)}$$

Figure B.16.: Failure handling related reduction of typing environments.

The next lemma states if Def. 42 (Intra. Coherence) holds and a reduction effects only one session, then after reduction Def. 42 (Intra. Coherence) still holds for all other sessions.

Lemma 9. *Given a coherent* $(\Delta, \Sigma, \mathcal{F}, \Theta)$ *typing environment and a typing environment reduction* $(\Delta, \Sigma, \mathcal{F}, \Theta) \rightarrow (\Delta', \Sigma', \mathcal{F}, \Theta)$ *with* $\Delta \setminus s = \Delta' \setminus s$ *and* $\Sigma \setminus s = \Sigma' \setminus s$ *(for some* s*) then* $\forall s' \in \Sigma'$ *with* $s \neq s'$ *Def. 42 (Intra. Coherence) holds in* $(\Delta', \Sigma', \mathcal{F}, \Theta)$.

Proof. By assumption $\forall s \in \Sigma$ in $(\Delta, \Sigma, \mathcal{F}, \Theta)$ Def. 42 (Intra. Coherence) holds. $\forall s' \in \Sigma'$ with $s' \neq s$ we have: $s' \in \Sigma$ and $\Sigma(s') = \Sigma'(s') \wedge \Delta(s') = \Delta'(s')$. Therefore, Def. 42 (Intra. Coherence) holds for s' in $(\Delta', \Sigma', \mathcal{F}, \Theta)$ □

Lemma 10 (Splitting of prefix local type minus)**.** *Given a coherent* $(\Delta, \Sigma, \mathcal{F}, \Theta)$ *typing environments, a subprotocol state* L*, a guard type* $L_1.L_2$*, and* $L - L_1.L_2$ *being defined, then:* $L - L_1.L_2 = (L - L_1) - L_2$

Proof. Proof by induction. By Definition 20 (-) $L_1.L_2$ is a simple type. Therefore, L_1 and L_2 must be simple local types. For the basic rules $(?, !)$ it is easy to see that the property holds. By the assumption $L - L_1.L_2$ being defined the *"with"* cases follow.

□

Lemma 11. *Given a coherent* $(\Delta, \Sigma, \mathcal{F}, \Theta)$ *typing environments, a partial local type* T *derived by partial projection from a protocol type and a queue* b*, then the following holds:*

- *Given* $T - b[p \mapsto \bar{l}](p)$ *and* $T - b[p \mapsto \bar{l} \cdot l'](p)$ *then* $T - b[p \mapsto \bar{l} \cdot l'](p) = (T - b[p \mapsto \bar{l}](p)) - [p \mapsto l'](p)$
- *Given* $T - b[p \mapsto l' \cdot \bar{l}](p)$ *and* $T - b[p \mapsto l'](p)$ *then* $T - b[p \mapsto l' \cdot \bar{l}](p) = (T - b[p \mapsto l'](p)) - [p \mapsto \bar{l}](p)$
- *Given* $T - b[p \mapsto \bar{l}](p)$ *then* $T - b[p \mapsto \bar{l}](p) = (T - b[p \mapsto \epsilon](p)) - [p \mapsto \bar{l}](p)$
- *Given* $T - [p \mapsto l_1](q)$ *and* $T - b[p \mapsto l_1 \cdot \bar{l}](q)$ *then* $T - b[p \mapsto l_1 \cdot \bar{l}](q) = (T - [p \mapsto l_1]) - b[p \mapsto \bar{l}](q)$

Proof. Induction on derivations. □

Lemma 12. *Given a well-formed top level global type* $\mathcal{G}$*, with* $g^{\bar{q};p_0}\restriction p_1$*,* $g^{\bar{q};p_0}\restriction p_2$*, and* $p_1 \neq p_2$ *then* $\mathsf{unf}(g^{\bar{q};p_0}\restriction p_1 \Vert p_2 - \varepsilon) \bowtie \mathsf{unf}(g^{\bar{q};p_0}\restriction p_2 \Vert p_1 - \varepsilon)$

Proof. By definition of Definitions 18, 36 and 40. □

Proof: preservation of coherence

Lemma 13 (Preservation of Coherence)**.** *Assume a well-formed top-level* $\mathcal{G}$*, coherent environments* $(\Delta, \Sigma, \mathcal{F}, \Theta)$*, and that* $(\Delta, \Sigma, \mathcal{F}, \Theta) \to (\Delta', \Sigma', \mathcal{F}', \Theta')$*. Then* $(\Delta', \Sigma', \mathcal{F}', \Theta')$ *is coherent.*

Proof. Proof **by enumerating over the cases of** $(\Delta, \Sigma, \mathcal{F}, \Theta) \to (\Delta', \Sigma', \mathcal{F}, \Theta')$. Wolg we assume p is the endpoint performing the reduction step (except in spawn where multiple endpoints perform a reduction step together).

The proof focuses on showing that Def. 42 (Coherence).1 holds after reduction. We discuss Def. 42 (Coherence).2 - Def. 42 (Coherence).3 for some interesting cases. However: Def. 42 (Coherence).2 usually follows directly from Lemma 10; and Def. 42 (Coherence).3 is an administrative property which trivially hold.

- Case [[FIRE]]:
  Assume $p \notin \mathcal{F}$ (if $p \in \mathcal{F}$ then coherence holds trivially)

  By [[FIRE]]:
  $$\Sigma = \Sigma' \text{ and } \Theta = \Theta' \text{ and } \mathcal{F} = \mathcal{F}' \tag{B.1}$$

  By [[FIRE]]:
  $$\begin{gathered} \Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \\ \Delta' = \Delta_1' \cdot \Delta_{\mathsf{end}} \\ \exists s[p] : L_s \in \Delta_1' \text{ with } \Delta_1' = \Delta_1, s[p] : L_s \text{ and } s[p] : (L, b) \in \Sigma \end{gathered} \tag{B.2}$$

Preservation of Definition 42 (Intra. Coherence) for s: We only show Def. 42 (Coherence).2 as the other parts are straightforward.

Before reduction by [[FIRE]] and assumption coherence :

$$\nexists s[p] \in \Delta_1 \tag{B.3}$$
$$p{:}R : \{L_i\}_{i \in I} \in \Delta \tag{B.4}$$
$$\{L_i\}_{i \in I} \vdash L \tag{B.5}$$
$$L_s \in \{L_i\}_I \tag{B.6}$$
$$L \asymp L_s \tag{B.7}$$

By Definition 27 and Equation (B.7): L is not a μ type in the relevant (normal/failure handling) activity and therefore $\mathsf{unf}()$ has no effect. Furthermore, by Equations (B.5) to (B.7) and Definition 33, and monotonicity we have:

$$\{L_i\}_{i \in I} \vdash L - L_s \tag{B.8}$$

Preservation of Definition 42 (Intra. Coherence) for the other sessions: Holds by Lemma 9.

Preservation of Definition 43 (Inter. Coherence): The reduction does not effect the general tree structure nor performs failure handling.

- Case [[SEND]]: By reduction:

$$\Theta = \Theta' \tag{B.9}$$
$$\mathcal{F} = \mathcal{F}' \tag{B.10}$$
$$\Delta = s[p] : r!l_1.L \cdot \Delta_c \tag{B.11}$$
$$\Sigma = s[p] : (\mathcal{E}[q{:}r!\{l_i : L_i\}_I], b) \cdot s[q] : (L_q, b_q) \cdot \Sigma_c \tag{B.12}$$
$$\Delta' = s[p] : L \cdot \Delta_c \tag{B.13}$$
$$\Sigma' = s[p] : (\mathcal{E}[L_1], b), s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_1]) \cdot \Sigma_c \tag{B.14}$$

Preservation of Def. 42 (Intra. Coherence): First we show that duality is preserved between p and a p' who is not the receiver. Then we show duality between p and q

Preservation of duality between p and a p'.
Let $p' \neq q$ and wolg p and p' unsuspected and involved in s (otherwise nothing to show). Let $s[p'] : (L_{p'}, b) \in \Sigma$. In $(\Delta, \Sigma, \mathcal{F}, \Theta)$ we have:

$$\mathsf{unf}(\mathcal{E}[q{:}r!\{l_i : L_i\}] \upharpoonright p' - b(p')) \bowtie \mathsf{unf}(L_{p'} \upharpoonright p - b_{p'}(p)) \tag{B.15}$$

By Definition 18 (projection) and Definition 36 ($\upharpoonright$) ($L_i \upharpoonright p' = L_j \upharpoonright p'$) on Equation (B.15),

$$\mathcal{E}[q{:}r!\{l_i : L_i\}] \upharpoonright p' = \mathcal{E}[L_i] \upharpoonright p' \tag{B.16}$$

Therefore, Def. 42 (Coherence).1 in $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ holds.

Now we show that duality is preserved between p and q. Case distinction over $p \notin \mathcal{F}, q \notin \mathcal{F}$ status:

Case a: $p \notin \mathcal{F}, q \notin \mathcal{F}$: Def. 42 (Coherence).2 follows from Lemma 10

By assumption in $(\Delta, \Sigma, \mathcal{F}, \Theta)$:

$$\mathsf{unf}(\mathcal{E}[q{:}r!\{l_i : L_i\}] \upharpoonright q - b(q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.17}$$

Definition 39 (-) and Definition 40 (duality)

$$\mathsf{unf}(\mathcal{E}[q{:}r!\{l_i : L_i\}] \upharpoonright q - b(q)) \bowtie \mathsf{unf}(T_{pq}) \tag{B.18}$$

Case distinction over the failure handling status:

Case 1: Case both p, q have no active handling (in partial types): By Def. 42 (Coherence).3 and Def. 42 (Coherence).3 before reduction and Definition 36 ($\upharpoonright$),

$$b(q) = \varepsilon \tag{B.19}$$

By Equation (B.19):

$$\mathsf{unf}(\mathcal{E}[q{:}r!\{l_i : L_i\}] \upharpoonright q - b[q \mapsto \epsilon](q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.20}$$

By Definition 40 (duality) and Definition 39 (-),

$$\mathsf{unf}(L_q \upharpoonright p - b_q(p)) = \mathcal{T}'[?\{l_i : T_i'\}] \tag{B.21}$$

By Definition 40 (duality)

$$\mathsf{unf}(\mathcal{T}[!\{l_i : T_i\}]) \bowtie \mathcal{T}'[?\{l_i : T_i'\}] \tag{B.22}$$

Next we show that Def. 42 (Coherence).1 holds in $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$.

$$\mathsf{unf}(\mathcal{E}[L_1]\upharpoonright q - b(q)) \bowtie \mathsf{unf}(L_q\upharpoonright p - b_q[p \mapsto b_q(p).l_1](p)) \tag{B.23}$$

By Equations (B.18) and (B.19), Lemma 11 and both non active handling (wolg $L_q\upharpoonright p - b_q(p)$ has no μ type as the prefix):

$$\mathsf{unf}(\mathcal{E}[L_1]\upharpoonright p - b[q \mapsto \epsilon](q)) \bowtie \mathsf{unf}(\mathcal{T}'[?\{l_i : T_i'\}] - [p \mapsto l_1](p)) \tag{B.24}$$

Apply Definition 36 ($\upharpoonright$) on Equation (B.24), Definition 39 (-)

$$\mathsf{unf}(\mathcal{T}[T_1]) \bowtie \mathsf{unf}(\mathcal{T}'[T_1']) \tag{B.25}$$

By Equation (B.22) and Definition 40 (duality) Equation (B.25) holds and therefore Equation (B.23).

Case 2: Case p not active handling, q active handling: By Def. 42 (Intra. Coherence) in $(\Delta, \Sigma, \mathcal{F}, \Theta)$:

$$\mathsf{unf}(q{:}r!\{l_i : L_i\} \text{ with } L_p'\upharpoonright q - b(q)) \bowtie \mathsf{unf}(- \text{ with } L_q'\upharpoonright p - b_q(p)) \tag{B.26}$$

Two cases (i) failure label in b or (ii) otherwise

Case i: By Definition 36 ($\upharpoonright$) and Definition 39 (-)

$$q!\{l_i : L_i\} \text{ with } L_p'\upharpoonright q - b(q) = \{- \text{with } T_{pq}\} \tag{B.27}$$

By Definition 40 (duality) and Definition 41 ($\mathsf{unf}()$) on Equation (B.27)

$$\mathsf{unf}(\{- \text{with } T_{pq}\}) \bowtie \mathsf{unf}(- \text{ with } T_{qp}) \tag{B.28}$$

We show that Def. 42 (Coherence).1 holds in $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ ($b_q' = b_q[p \mapsto b_q(p).l_1]$):

$$\mathsf{unf}(L_i \text{ with } L_p'\upharpoonright q - b(q)) \bowtie \mathsf{unf}(- \text{ with } L_q'\upharpoonright p - b_q'(p)) \tag{B.29}$$

By case, Definition 36 ($\upharpoonright$), and Definition 39 (-) on Equation (B.29)

$$\mathsf{unf}(\{- \text{with } T_{pq}\}) \bowtie \mathsf{unf}(- \text{ with } L_q'\upharpoonright p - b_q'(p)) \tag{B.30}$$

By Lemma 11 on Equation (B.30)

$$\mathsf{unf}(\{- \text{with } T_{pq}\}) \bowtie \mathsf{unf}(\{- \text{with } T_{qp}\} - [p \mapsto l_1]) \tag{B.31}$$

By Definition 39 (-) (clean) and Def. 42 (Coherence).3 on Equation (B.31)

$$\mathsf{unf}(\{- \text{with } T_{pq}\}) \bowtie \mathsf{unf}(\{- \text{with } T_{qp}\}) \tag{B.32}$$

Done.

Case ii: In $(\Delta, \Sigma, \mathcal{F}, \Theta)$

$$\mathsf{unf}(q{:}r!\{l_i : L_i\} \text{ with } L'_p \Updownarrow q - b(q)) \bowtie \mathsf{unf}(- \text{ with } L'_q \Updownarrow p - b_q(p)) \tag{B.33}$$

By [[SEND]] and Definition 39 (-) on Equation (B.33)

$$\begin{aligned}
&\mathsf{unf}(q{:}r!\{l_i : L_i\} \text{ with } L'_p \Updownarrow q - b(q)) \bowtie \mathsf{unf}(\{-\text{with } T_{qp}\}) \\
&\mathsf{unf}(\{!\{l_i : T_i\} \text{ with } T'_p\} - b(q)) \bowtie \mathsf{unf}(\{-\text{with } T_{qp}\}) \\
&\mathsf{unf}(\{!\{l_i : T'_i\} \text{ with } T'_p\}) \bowtie \mathsf{unf}(\{-\text{with } T_{qp}\})
\end{aligned} \tag{B.34}$$

We show that Def. 42 (Coherence).1 holds in $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$\mathsf{unf}(L_1 \text{ with } L'_p \Updownarrow q - b(q)) \bowtie \mathsf{unf}(- \text{ with } L'_q \Updownarrow p - b_q[p \mapsto b_q(p).l_1](p)) \tag{B.35}$$

$$\mathsf{unf}(\{T_1 \text{ with } T'_p\} - b(q)) \bowtie \mathsf{unf}(- \text{ with } L'_q \Updownarrow p - b_q[p \mapsto b_q(p).l_1](p)) \quad \text{By Equation (B.34)} \tag{B.36}$$

$$\mathsf{unf}(\{T'_1 \text{ with } T'_p\}) \bowtie \mathsf{unf}(\{-\text{with } T_{qp}\} - l_i) \quad \text{(Equation (B.34) and Lemma 11)} \tag{B.37}$$

$$\mathsf{unf}(\{T'_1 \text{ with } T'_p\}) \bowtie \mathsf{unf}(\{-\text{with } T_{qp}\}) \quad \begin{array}{c}\text{(Definition 39 (-)(clean)} \\ \text{and Def. 42 (Coherence).3)}\end{array} \tag{B.38}$$

Done.

Case 3: Case p active handling, q non active handling (under consideration of b_q)

$$\mathsf{unf}(- \text{ with } q{:}r!\{l_i : L_i\} \Updownarrow q - b(q)) \bowtie \mathsf{unf}(L_q \Updownarrow p - b_q(p)) \tag{B.39}$$

Rewriting left side (by case there is no mulitsend label associated with failure handling in the queue):

$$\mathsf{unf}(- \text{ with } q{:}r!\{l_i : L_i\} \Updownarrow q - b(q)) = \mathsf{unf}(\{-\text{with } !\{l_i : L_i \Updownarrow q\}\}) \tag{B.40}$$

by (duality) and wolg:

$$\mathsf{unf}(L_q \Updownarrow p - b_q(p)) = \mathsf{unf}(\{T_{qp} \text{ with } ?\{l_i : T'_i\}\}) \tag{B.41}$$

We need to show in $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$\mathsf{unf}(- \text{ with } L_1 \Updownarrow q - b(q)) \bowtie \mathsf{unf}(L_q \Updownarrow p - b_q[p \mapsto b_q(p).l_1](p)) \tag{B.42}$$

$$\mathsf{unf}(\{-\text{with } L_1 \Updownarrow q\}) \bowtie \mathsf{unf}(\{-\text{with } ?\{l_i : T'_i\}\} - l_1) \tag{B.43}$$

$$\mathsf{unf}(\{-\text{with } L_1 \Updownarrow q\}) \bowtie \mathsf{unf}(\{-\text{with } T'_1\}) \tag{B.44}$$

Done.

We now show that duality between q and some non suspected participants, say p_1 is preserved ($s[p_1] : (L_{p'1} \text{ with } L_{p'2}, b_{p_1}) \in \Sigma$. This case is interesting if the label send (by p) is the label that activate failure handling in q's partial types. Therefore, we assume l_1 is the label that activates failure handling in q's partial type (otherwise duality follows easily for p_1). We further assume p_1 has no active failure handling (in its partial type). The case where p_1 has active handling is similar.

$$\mathsf{unf}(L_q \text{ with } L'_q \Uparrow p_1 - b_q(p_1)) \bowtie \mathsf{unf}(L_{p'1} \text{ with } L_{p'2} \Uparrow q - b_{p_1}(q)) \tag{B.45}$$

By Definition 36 ($\Uparrow$) on Equation (B.45)

$$\mathsf{unf}(\{T_q \text{ with } T'_q\} - b_q(p_1)) \bowtie \mathsf{unf}(\{T_{p11} \text{ with } T_{p12}\} - b_{p_1}(q)) \tag{B.46}$$

By assumption both default activity (in their partial types):

$$\mathsf{unf}(\{T''_q \text{ with } T'_q\}) \bowtie \mathsf{unf}(\{T'''_{p11} \text{ with } T_{p12}\}) \tag{B.47}$$

After reduction we have to show for $b'_q = b_q[p \mapsto b_q(p) \cdot l_1]$:

$$\mathsf{unf}(L_q \text{ with } L'_q \Uparrow p_1 - b'_q(p_1)) \bowtie \mathsf{unf}(L_{p'1} \text{ with } L_{p'2} \Uparrow q - b_{p_1}(q)) \tag{B.48}$$

By Definition 36 ($\Uparrow$) on Equation (B.48)

$$\mathsf{unf}(\{-\text{with } T'_q\} - b'_q(p_1)) \bowtie \mathsf{unf}(L_{p'1} \text{ with } L_{p'2} \Uparrow q - b_{p_1}(q)) \tag{B.49}$$

By l_1 not a mulitsent label

$$\mathsf{unf}(\{-\text{with } T'_q\} - b'_q(p_1)) = \mathsf{unf}(\{-\text{with } T'_q\} - b_q(p_1)) \tag{B.50}$$

By Equation (B.50), clean on Equation (B.49)

$$\mathsf{unf}(\{-\text{with } T'_q\}) \bowtie \mathsf{unf}(\{T'''_{p11} \text{ with } T_{p12}\}) \tag{B.51}$$

Therefore duality holds by Equation (B.47).

Case 4: Case p, q active handling: similar to the case both p, q non active handling

Case b: $p \notin \mathcal{F} \ \wedge \ q \in \mathcal{F}$. Since $q \in \mathcal{F}$ we do not require duality.

Case c: $p \in \mathcal{F} \ \wedge \ q \notin \mathcal{F}$. Since $p \in \mathcal{F}$ we do not require duality. See [[RECV]] and [[RECV-HDL]] for the treatment of a message from a suspected participant.

Preservation of Definition 43 (Inter. Coherence): The reduction does not effect the general tree structure nor performs failure handling.

- Case [[MSEND]]: By reduction:

$$\Theta = \Theta' \tag{B.52}$$
$$\mathcal{F} = \mathcal{F}' \tag{B.53}$$
$$\Delta = s[p] : R!l_1.L \cdot \Delta_c \tag{B.54}$$
$$\Sigma = s[p] : (\mathcal{E}[R!\{l_i : L_i\}_I], b) \cdot \cup_{q \in \widetilde{p}} \{s[q] : (L_q, b_q)\} \cdot \Sigma_c \tag{B.55}$$
$$\Delta' = s[p] : L \cdot \Delta_c \tag{B.56}$$
$$\Sigma' = s[p] : (\mathcal{E}[L_1], b), \cup_{q \in \widetilde{p}} \{s[q] : (L_q, b'_q)\} \cdot \Sigma_c \tag{B.57}$$
$$R_{ids} \setminus \mathcal{F} = \widetilde{p} \tag{B.58}$$
$$l_1 \in \{l_i\}_{i \in I} \tag{B.59}$$
$$\forall q \in \widetilde{p}.\ b'_q = b_q[p \mapsto b_q(p) \cdot l_1] \tag{B.60}$$

Preservation of Def. 42 (Coherence).1: We will show that duality is preserved between two unsuspected participants p_1 and p_2 with $p_1, p_2 \in \widetilde{p}$. The other cases are similar to the [[SEND]] case.

Wolg the multisend happens in the default activity and p_1, p_2, and p are in the default activity (in their partial types). (The case is similar for a multisend in the failure handling activity.)

In $(\Delta, \Sigma, \mathcal{F}, \Theta)$ we have:

$$\mathsf{unf}(L_{p_1} \upharpoonright p_2 - b_{p_1}(p_2)) \bowtie \mathsf{unf}(L_{p_2} \upharpoonright p_1 - b_{p_2}(p_1)) \tag{B.61}$$

and

$$\mathsf{unf}(L_{p_j} \upharpoonright p - b_{p_j}(p)) \bowtie \mathsf{unf}(\mathcal{E}[R!\{l_i : L_i\}_I] - b(p_j)) \qquad j \in \{1, 2\} \tag{B.62}$$

By Equation (B.62), Definition 39 (-) and Definition 36 ($\upharpoonright$):

$$\mathsf{unf}(L_{p_j} \upharpoonright p - b_{p_j}(p)) = \mathcal{T}_j[?\{l_i : T_{ji}\}_I] \tag{B.63}$$

By Equation (B.62), Equation (B.63), Definition 39 (-) and Definition 36 ($\upharpoonright$):

$$\mathsf{unf}(L_{p_j} \upharpoonright p_k - b_{p_j}(p_k)) = T_j \quad T_j = \mathcal{T}'_j[T'_j] \quad p\underline{?}\{l_i : T_{ji}\}_I \in T'_j \qquad \{j, k\} = \{1, 2\} \tag{B.64}$$

By Definition 36 ($\Vert$), Definition 39 (-) on Equation (B.61)

$$\mathsf{unf}(T_1) \bowtie \mathsf{unf}(T_2) \tag{B.65}$$

After reduction we have to show:

$$\mathsf{unf}(L_{p_1} \Vert p_2 - b_{p_1}[p \mapsto b_{p_1}(p) \cdot l_1](p_2)) \bowtie \mathsf{unf}(L_{p_2} \Vert p_1 - b_{p_2}[p \mapsto b_{p_2}(p) \cdot l_1](p_1)) \tag{B.66}$$

By $\mathsf{unf}(L_{p_j} \Vert p - b_{p_j}(p_k)) = T_j$ being defined ($j \in \{1, 2\}$):

$$\mathsf{unf}(T_1 - [p \mapsto l_1](p_2)) \bowtie \mathsf{unf}(T_1 - [p \mapsto l_1](p_1)) \tag{B.67}$$

By Equation (B.64)

$$\mathsf{unf}(T_1') \bowtie \mathsf{unf}(T_2') \tag{B.68}$$

Removing "the" partial multisend receive type present in two dual partial types preserves duality. Therefore, duality holds in Equation (B.68).

- Case [[RECV]]: By reduction of [[RECV]]

$$\Theta = \Theta' \tag{B.69}$$
$$\Delta = s[p] : r?l_1.L \cdot \Delta_c \tag{B.70}$$
$$\Sigma = s[p] : (\mathcal{E}[q{:}r?\{l_i : L_i\}_{i \in I}], b_p) \cdot \Sigma_c \tag{B.71}$$
$$(q \mapsto l_1 \cdot \bar{l}) \in b \tag{B.72}$$
$$\Delta' = s[p] : L \cdot \Delta_c \tag{B.73}$$
$$\Sigma' = s[p] : (\mathcal{E}[L_1], b_p[q \mapsto \bar{l}]) \cdot \Sigma_c \tag{B.74}$$

Preservation of Def. 42 (Intra. Coherence): Def. 42 (Coherence).2 hold after reduction by Lemma 10.

Next we show that Def. 42 (Coherence).1 holds after reduction by a case distinction over p, q. And separated case if the receive belongs to a multisend for two other participants associated with that multisend.

Case a: Case p and q unsuspected:

By case assumption and Definition 42 (Intra. Coherence):

$$s[q] : (L_q, b_q) \in \Sigma_c \tag{B.75}$$

By Equation (B.75), Definition 42 (Intra. Coherence) and [[RECV]]:

$$\mathsf{unf}(\mathcal{E}[q{:}r?\{l_i : L_i\}] \Vert q - b_p(q)) \bowtie \mathsf{unf}(L_q \Vert p - b_q(p)) \tag{B.76}$$

Case distinction based on active failure handling:

Case 1: Case p, q have no active failure handling (in partial types):

$$\mathsf{unf}(\mathcal{E}[q?\{l_i : L_i\}] \Uparrow q - b_p(q)) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \quad \text{(Equation (B.72) and Equation (B.76))} \tag{B.77}$$

$$\mathsf{unf}(\mathcal{T}[?\{l_i : T_i\}] - [q \mapsto l_1 \cdot \bar{l}]) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \quad \text{(Definition 36 (}\Uparrow\text{))} \tag{B.78}$$

$$\mathsf{unf}(\mathcal{T}[T_1] - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \quad \text{(Lemma 11 and Definition 39 (-))} \tag{B.79}$$

For $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ we show:

$$\mathsf{unf}(\mathcal{E}[L_1] \Uparrow q - b_p[q \mapsto \bar{l}](q)) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \tag{B.80}$$

$$\mathsf{unf}(\mathcal{T}[T_1] - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \quad \text{(Definition 36 (}\Uparrow\text{))} \tag{B.81}$$

Done (Equation (B.81) same as Equation (B.76)). Other $p' \notin \mathcal{F}$ follow trivially.

Case 2: Case p not active handling, q active handling: By case assumption:

$$L_q = - \text{ with } L'_q \tag{B.82}$$

$$\mathsf{unf}(q{:}r?\{l_i : L_i\} \text{ with } L'_p \Uparrow q - b_p(q)) \bowtie \mathsf{unf}(- \text{ with } L'_q \Uparrow p - b_q(p)) \tag{B.83}$$

Case distinction on if label from failure handling present in b_p.

Case a: No label from failure handling and Definition 36 ($\Uparrow$) on Equation (B.83):

$$\mathsf{unf}(\{?\{l_i : T_i\} \text{ with } T_{pq}\} - [q \mapsto l_1 \cdot \bar{l}]) \bowtie \mathsf{unf}(\{- \text{with } T_{qp}\} - b_q(p)) \tag{B.84}$$

By (Lemma 11)

$$\mathsf{unf}(\{T_1 \text{ with } T_{pq}\} - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(\{- \text{with } T_{qp}\} - b_q(p)) \tag{B.85}$$

After reduction $((\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta'))$ we have.

$$\mathsf{unf}(L_1 \text{ with } L'_p \Uparrow q - b_p[q \mapsto \bar{l}](q)) \tag{B.86}$$

$$\mathsf{unf}(\{T_1 \text{ with } T_{pq}\} - [q \mapsto \bar{l}]) \quad \text{Definition 36 (}\Uparrow\text{) \& Definition 39 (-) (step 1)} \tag{B.87}$$

Equation (B.87) is same as Equation (B.85) (left side), therefore Def. 42 (Coherence).1 holds. Other $p' \in \Theta$ in both cases follow trivially

Case b: Label for failure handling present:
Definition 36 ($\Downarrow$) on Equation (B.83):

$$\mathsf{unf}(\{-\mathsf{with}\ T_{pq}\} - b_p(q)) \bowtie \mathsf{unf}(\{-\mathsf{with}\ T_{qp}\} - b_q(p)) \tag{B.88}$$

By Lemma 11, Definition 39 (-)(first part) and clean in Definition 39 (-) (second part), i.e. removing the l_1 label, on Equation (B.88)

$$\mathsf{unf}(\{-\mathsf{with}\ T'_{pq}\} - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(\{-\mathsf{with}\ T_{qp}\} - b_q(p)) \tag{B.89}$$

By case assumption

$$\mathsf{unf}(\{-\mathsf{with}\ T''_{pq}\}) \bowtie \mathsf{unf}(\{-\mathsf{with}\ T_{qp}\} - b_q(p)) \tag{B.90}$$

After reduction $((\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta'))$ we have:

$$\mathsf{unf}(\{-\mathsf{with}\ T_{pq}\} - b_p[q \mapsto \bar{l}](q)) \bowtie \mathsf{unf}(\{-\mathsf{with}\ T_{qp}\} - b_q(p)) \tag{B.91}$$

By Definition 39 (-) (first part removal multisend labels)

$$\mathsf{unf}(\{-\mathsf{with}\ T'_{pq}\} - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(\{-\mathsf{with}\ T_{qp}\} - b_q(p)) \tag{B.92}$$

By Definition 39 (-) (second part)

$$\mathsf{unf}(\{-\mathsf{with}\ T''_{pq}\}) \bowtie \mathsf{unf}(\{-\mathsf{with}\ T_{qp}\} - b_q(p)) \tag{B.93}$$

Case 3: Case p active handling, q non active handling. We show that this case *can not* happen.

Assume:

$$\mathsf{unf}(-\ \mathsf{with}\ q{:}r?\{l_i : L_i\} \Downarrow q - b_p(q)) \bowtie \mathsf{unf}(L_q\ \mathsf{with}\ L'_q \Downarrow p - b_q(p)) \tag{B.94}$$

$$\mathsf{unf}(\{-\mathsf{with}\ T_i\} - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(L_q\ \mathsf{with}\ L'_q \Downarrow p - b_q(p)) \quad ([\![\text{RECV}]\!]\ (\text{Equation (B.72)})) \tag{B.95}$$

By Def. 42 (Coherence).3 (which holds before reduction) we have distinct set of labels for the normal activity and the failure handling activity. Therefore q needs to be in active failure handling and we have a contradiction.

Case 4: p, q active handling: Similar to the case both p, q non active handling

Case b: p unsuspected and q suspected. Wolg p' unsuspected and $s[p'] : (L_{p'}, b_{p'}) \in \Sigma_c$. By Definition 44 (Coherence) before reduction:

$$\mathsf{unf}(\mathcal{E}[q{:}r?\{l_i : L_i\}] \upharpoonright p' - b_p(p')) \bowtie \mathsf{unf}(L_{p'} \upharpoonright p - b_{p'}(p)) \tag{B.96}$$

By Definition 18 (projection) for any i:

$$\mathcal{E}[q{:}r?\{l_i : L_i\}] \upharpoonright p' = \mathcal{E}[L_i] \upharpoonright p' \tag{B.97}$$

After reduction we have (in $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$) by Equation (B.97):

$$\mathsf{unf}(\mathcal{E}[L_1] \upharpoonright p' - b_p[q \mapsto l](p')) \bowtie \mathsf{unf}(L'_p \upharpoonright p - b_{p'}(p)) \tag{B.98}$$

Case multisend. Let $p, p' \in R_{ids}$, $p, p' \notin \mathcal{F}$ and $q{:}r?\{l_i : L_i\}_{i \in I}$ be a receive that was projected from the multisend $r \to R\{l_i : G_i\}$. With $s[p'] : (L', b_{p'}) \in \Sigma$

By coherence before reduction:

$$\mathsf{unf}(\mathcal{E}[q{:}r?\{l_i : L_i\}] \upharpoonright p' - b_p(p')) \bowtie \mathsf{unf}(L' \upharpoonright p - b_{p'}(p)) \tag{B.99}$$

By Definition 36 ($\upharpoonright$) and wolg Definition 36 ($\upharpoonright$) does not activate failure handling

$$\mathsf{unf}(\mathcal{E}[q{:}r?\{l_i : L_i\}] \upharpoonright p' - b_p(p')) = \mathsf{unf}(\mathcal{T}[q\underline{?}\{l_i : T_i\}] - b_p(p')) \tag{B.100}$$

By Definition 39 (-) (first part, one step)

$$\mathsf{unf}(\mathcal{T}[q\underline{?}\{l_i : T_i\}] - b_p(p')) = \mathsf{unf}(\mathcal{T}[T_1] - b_p[q \mapsto \bar{l}](p')) \tag{B.101}$$

After reduction we have to show:

$$\mathsf{unf}(\mathcal{E}[L_1] \upharpoonright p' - b_p[q \mapsto \bar{l}](p')) \bowtie \mathsf{unf}(L'_p \upharpoonright p - b_{p'}(p)) \tag{B.102}$$

By Definition 36 ($\upharpoonright$)

$$\mathsf{unf}(\mathcal{T}[T_1] - b_p[q \mapsto \bar{l}](p')) \bowtie \mathsf{unf}(L'_p \upharpoonright p - b_{p'}(p)) \tag{B.103}$$

By Equation (B.103) being the same as Equation (B.101), duality holds also after reduction.

Preservation of Definition 43 (Inter. Coherence): The reduction does not effect the general tree structure nor performs failure handling.

- Case [[CLEAN]]: By reduction:

$$\Theta = \Theta' \quad \Delta = \Delta' \quad \mathcal{F} = \mathcal{F}' \tag{B.104}$$

Preservation of Definition 42 (Intra. Coherence) for s: Assume $p \notin \mathcal{F}$ (If $p \in \mathcal{F}$ coherence holds trivially).

Wolg, before and after [[CLEAN]] reduction:

$$\Sigma = s[p] : (L, b), \Sigma'' \tag{B.105}$$
$$\Sigma' = s[p] : (L', b'), \Sigma'' \tag{B.106}$$
$$L = L' \quad b = b'[q \mapsto l_1 \cdot b'(q)] \tag{B.107}$$

Note [[CLEAN]] only removes label associated with default activities.

By Equation (B.104) we have Def. 42 (Coherence).2 after reduction. By Equations (B.104) to (B.106) we have Def. 42 (Coherence).3 after reduction.

We now show Def. 42 (Coherence).1 for p, q, wolg assume $q \notin \mathcal{F}$ (otherwise we have nothing to show). Note that for $s[p_i] \in \Sigma$ with $s[p_i] \neq s[q] \ \wedge \ s[p_i] \neq s[p]$ Def. 42 (Coherence).1 holds trivially by Equations (B.105) to (B.107).

By Def. 42 (Coherence).1 in $(\Delta, \Sigma, \mathcal{F}, \Theta)$ for $s[q] : (L_q, b_q) \in \Sigma$ we have

$$\mathsf{unf}(L \upharpoonright q - b(q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.108}$$

In $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ we need to show:

$$\mathsf{unf}(L \upharpoonright q - b'(q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.109}$$

By definition of [[CLEAN]], Definition 36 ($\upharpoonright$) and Definition 39 (-) (If a label is cleaned then we have active handling in partial types. Therefore clean applies there as well.)

$$L \upharpoonright q - b(q) = L \upharpoonright q - b'(q) \tag{B.110}$$

Therefore, Def. 42 (Coherence).1 holds after reduction.

Preservation of Definition 42 (Intra. Coherence) for the other sessions: Holds by Lemma 9.

Preservation of Definition 43 (Inter. Coherence): The reduction does not effect the general tree structure nor performs failure handling.

- Case [[SESS-GC]]: Wolg, s is the parent session and s' was removed.

$$(\widetilde{p}_s, \widetilde{s}) = \Theta(s) \quad (\widetilde{p}_1, \emptyset) = \Theta(s') \tag{B.111}$$

Preservation of Def. 42 (Intra. Coherence): By hypothesis that we have Definition 43 (Inter. Coherence) before reduction:

$$\widetilde{p}_1 \subseteq \widetilde{p}_s \tag{B.112}$$

By [[SESS-GC]]

$$\Delta = \Delta' \ \wedge \ \Theta[s \mapsto (\widetilde{q}, \widetilde{s} \setminus s')] \setminus s' = \Theta' \ \wedge \ \mathcal{F} = \mathcal{F}' \ \wedge \ \widetilde{p} = \widetilde{p}_1 \setminus \mathcal{F} \tag{B.113}$$

By [[SESS-GC]]

$$\Sigma = \Sigma_c \cup \{s'[p] : (L'_p, b'_p)\}_{p \in \widetilde{p}} \tag{B.114}$$

$$\Sigma' = \Sigma_c, \tag{B.115}$$

Def. 42 (Intra. Coherence) holds after reduction since: By $\Delta = \Delta'$ Def. 42 (Coherence).2 holds and by for all configuration in Σ' there is no change to the protocol states and queues Def. 42 (Coherence).1 holds. By Equations (B.113) and (B.115) Def. 42 (Coherence).3 holds.

Preservation of Definition 43 (Inter. Coherence): A leaf of the session tree was removed and the rest of the session tree remains the same. Therefore, Definition 43 (Inter. Coherence) holds after reduction.

- Case [[SUSP]]:

  Wolg, p was removed, by reduction:

$$\Delta = \Delta' \quad \Sigma = \Sigma' \quad \Theta = \Theta' \quad \mathcal{F} \cup q = \mathcal{F}' \tag{B.116}$$

Preservation of Def. 42 (Intra. Coherence): Holds trivially by Equation (B.116) as Def. 42 (Intra. Coherence) requires less if q is set to suspected, which is the only change.

Preservation of Definition 43 (Inter. Coherence): The reduction does not effect the general tree structure nor performs failure handling.

- Case [[MON]] (p performed the step and $p \notin \mathcal{F}$): By reduction:

$$\bar{c} \rightsquigarrow_{\Theta}^{+} = \{s_i\}_{i \in I} \tag{B.117}$$
$$\Theta = \Theta' \tag{B.118}$$
$$\mathcal{F} = \mathcal{F}' \tag{B.119}$$
$$p' \in \mathcal{F} \tag{B.120}$$
$$\Delta = s[p] : r'{\downarrow}.L_h \cdot \Delta_c \tag{B.121}$$
$$\Delta' = s[p] : L_h \cdot \Delta_c \tag{B.122}$$
$$\Sigma = s[p] : (L' \text{ with } p'{:}r'{\downarrow}.L, b) \cup_{i \in I} \ \{s_i[p] : (L_i, b_i)\} \cdot \Sigma_c \tag{B.123}$$
$$\Sigma' = s[p] : (- \text{ with } L, b) \cup_{i \in I} \ \{s_i[p] : (\mathsf{end}_{L_i'}, b_i)\} \cdot \Sigma_c \tag{B.124}$$
$$\text{where: } \mathsf{end}_{L_i'} = L_i \text{ or } L_i' = L_i$$

Note to Equation (B.124): a subsession, say s_i, of s is potentially already stopped and then $s_i[p]$ already has a stopped local type. If that is the case $\{s_i\}_{i \in I}$ contains an active handling before the reduction.

Preservation of Def. 42 (Intra. Coherence) for s: For non related c, c' with $c, c' \notin \{s, s_1, ...s_n\}$ trivial satisfied.

We show for (all) $s[q] : (L_q, b_q) \in \Sigma$ with $p \neq q$, $q \neq p'$ and $q \notin \mathcal{F}$ that point wise duality holds.

By hypothesis in $(\Delta, \Sigma, \mathcal{F}, \Theta)$:

$$\mathsf{unf}(L' \text{ with } p'{:}r'{\downarrow}.L \upharpoonright q - b(q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.125}$$

Definition 36 ($\upharpoonright$), Definition 39 (-) and Definition 41 ($\mathsf{unf}()$) on Equation (B.125) (left side)

$$\mathsf{unf}(L' \text{ with } p'{:}r'{\downarrow}.L \upharpoonright q - b(q)) = \{T_p \text{ with } T_p'\} \tag{B.126}$$

Definition 40 (duality) and Equation (B.126) gives for Equation (B.125) (right side)

$$\mathsf{unf}(L_q \upharpoonright p - b_q(p)) = \{T_q \text{ with } T_q'\} \text{ or } \{- \text{with } T_q'\} \tag{B.127}$$

$\{- \text{with } T_q'\}$ not possible by Def. 42 (Coherence).3. By Definition 40 (duality):

$$\{T_q \text{ with } T_q'\} \quad T_p \bowtie T_q \quad T_p' \bowtie T_q' \tag{B.128}$$

By Equation (B.128) and Def. 42 (Coherence).3

$$\forall l \in b \quad l \notin T'_p \tag{B.129}$$

In $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ we have to show:

$$\mathsf{unf}(-\ \mathsf{with}\ L\!\upharpoonright\!q - b(q)) \bowtie \mathsf{unf}(L_q\!\upharpoonright\!p - b_q(p)) \tag{B.130}$$

using Equation (B.129), and Definition 39 (-) we get

$$\mathsf{unf}(-\ \mathsf{with}\ L\!\upharpoonright\!q) \bowtie \{T_q\ \mathsf{with}\ T'_q\} \tag{B.131}$$

By $p'\!\downarrow\!.L\!\upharpoonright\!q = L\!\upharpoonright\!q$

$$\{-\mathsf{with}\ T'_p\} \bowtie \{T_q\ \mathsf{with}\ T'_q\} \tag{B.132}$$

Duality holds.

We now show Def. 42 (Coherence).3: Let $\{L_i\}$ be the set of protocol states in $(\Delta, \Sigma, \mathcal{F}, \Theta)$ for s, then $(L_p = L'\ \mathsf{with}\ p'\!\downarrow\!.L) \in \{L_i\}$ and for any $L_q \in \{L_i\}$ with $L_q \neq L_p$:

$$L_p = L'\ \mathsf{with}\ p'\!\downarrow\!.L \quad L_q = L'_q\ \mathsf{with}\ p?l.L''_q \tag{B.133}$$

In $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$L_p = -\ \mathsf{with}\ L \quad L_q = L'_q\ \mathsf{with}\ p?l.L''_q \tag{B.134}$$

which confirms to Def. 42 (Coherence).3.

Preservation of Definition 42 (Intra. Coherence) the other sessions: For s_i in $\{s_1, .., s_n\}$ the change is setting the local types, say L to end_L. Let p be the participant who activated failure handling in s and let s_i be an arbitrary sub session of s which involves p. Further let q be a different unsuspected participant in s_i.

By $(\Delta, \Sigma, \mathcal{F}, \Theta)$ being coherent have:

$$s_i[p] : (L_p, b_p) \in \Sigma \quad s_i[q] : (L_q, b_q) \in \Sigma \tag{B.135}$$

and

$$\mathsf{unf}(L_p\!\upharpoonright\!q - b_p(q)) \bowtie \mathsf{unf}(L_q\!\upharpoonright\!p - b_q(p)) \tag{B.136}$$

By Definition 36 ($\upharpoonright$) on Equation (B.136) (left side):

$$\mathsf{unf}(T_p - b_p(q)) \bowtie \mathsf{unf}(L_q\!\upharpoonright\!p - b_q(p)) \tag{B.137}$$

After reduction, by using Equation (B.124), we have to show :

$$\mathsf{unf}(\mathsf{end}_{L_p} \upharpoonright q - b_p(q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.138}$$

Definition 36 ($\upharpoonright$) ($\mathsf{end}_{L_p} \upharpoonright q = L_p \upharpoonright q$) on Equation (B.138) (left side only first partial projection step)

$$\mathsf{unf}(L_p \upharpoonright q - b_p(q)) \bowtie \mathsf{unf}(L_q \upharpoonright p - b_q(p)) \tag{B.139}$$

By Equation (B.139) being the same as Equation (B.136) duality holds.

Preservation of Definition 43 (Inter. Coherence): For $p \notin \mathcal{F}$ the tree structure was not changed from Σ to Σ'.

We now show Def. 43 (2) for a $s_i[p]$. For $s'[p] \notin s[p] \cup \{s_i[p]\}_{i \in I}$ or any $s''[q]$ ($q \neq p$) Def. 43 (2) trivially holds after reduction.

Let

$$s_i[p] : (L_p, \_) \in \Sigma \quad s_i[p] : (L'_p, \_) \in \Sigma' \tag{B.140}$$

By [[MON]] either $L'_p = \mathsf{end}_{L_p}$ or $L_p = \mathsf{end}_{L''_p} = L'_p$

By Equation (B.123) ($s[p]$'s configuration is not stopped) and Def. 43 (2) if $L_p = \mathsf{end}_{L''_p} = L'_p$ then the active failure handling which covers $s_i[p]$ is in $\{s_i\}_{i \in I}$. Therefore, both cases are treated simultaneously.

By $s_i \in \{s_i\}_{i \in I}$ and Equation (B.124), $s[p]$ configuration is active, is in failure handling and covers s_i after reduction.

- Case [[SPAWN]]:

By reduction with $\widetilde{p}$ performing the reduction

$$\Theta' = \Theta[s_2 \mapsto (\widetilde{p}_2, \emptyset), s \mapsto (\widetilde{p}_1, \widetilde{s} \cup s_2)] \tag{B.141}$$
$$\mathcal{F} = \mathcal{F}' \tag{B.142}$$
$$\Delta = \Delta' \tag{B.143}$$
$$\Theta(s) = (\widetilde{p}_1, \widetilde{s}) \tag{B.144}$$
$$\widetilde{p} = \bar{q} \cup (R'_{ids} \cup \overline{R}_{ids}) \setminus \mathcal{F} \tag{B.145}$$
$$\widetilde{p}_2 \cap \mathcal{F} = \emptyset \tag{B.146}$$
$$\widetilde{p}_2 = \bar{q} \cup p_0 \cup (\overline{R}_{ids} \setminus \mathcal{F}) \tag{B.147}$$
$$p_0 \notin \bar{q} \tag{B.148}$$
$$p_0 \in R'_{ids} \tag{B.149}$$
$$\Sigma = \{p : \{L_i\}_{i \in I_p}\}_{p \in \widetilde{p}} \cup \{s[p] : (\mathcal{E}_p[g(\overline{q{:}r}; R'; \overline{R}).L_p], b_p)\}_{p \in \widetilde{p}} \cup \Sigma_c \tag{B.150}$$
$$\Sigma' = \{p : \{L_i\}_{i \in I_p}\}_{p \in \widetilde{p}} \cup \{s[p] : (\mathcal{E}_p[L_p], b_p)\}_{p \in \widetilde{p}} \cup \{s'[p] : (g^{\bar{q};p_0} \upharpoonright p, \varepsilon)\}_{\widetilde{p}_2} \cup \Sigma_c \tag{B.151}$$
$$\Delta = \Delta_1 \cdot \Delta_\emptyset \tag{B.152}$$
$$\forall p \in \widetilde{p}.\ \nexists s'[p] \notin \mathtt{dom}(\Delta_1) \tag{B.153}$$

Preservation of Def. 42 (Intra. Coherence): By Def. 42 (Coherence).2 before reduction and [[SPAWN]] (all $p \in \widetilde{p}$)

$$\{L_i\}_{i \in I} \vdash \mathcal{E}_p[g(\overline{q{:}r}; R; \overline{R}).L''_p] \tag{B.154}$$

By Definition 33, Equation (B.154) and Definition 20 (definition requires spawn handler with not continuation):

$$\{L_i\}_{i \in I} \vdash \mathcal{E}_p[L''_p] \tag{B.155}$$

i.e. Def. 42 (Coherence).2 holds after reduction.

Preservation of Def. 42 (Coherence).1: Rule [[SPAWN]] together with Def. 42 (Coherence).3 ensures that either all participants are in the default activity or the failure handling activity.

Case distinction: **Case a** all participants are in the default activity (we further assume no failure label in the queue; such a label implies we only need duality in the failure handling activity) or **Case b** all participant are in the failure handling activity.

Def. 42 (Coherence).1 for $p \in \widetilde{p}$ and $q \notin \widetilde{p}$ follows directly after reduction, since Definition 36 ($\Uparrow$) "removes" spawn in that case from the partial types.

By assumption before reduction ($p, q \in \widetilde{p}$):

$$\mathsf{unf}(\mathcal{E}_p[g(\overline{p{:}R}, R', \overline{R}).L''_p] \Uparrow p - b_p(q)) \bowtie \mathsf{unf}(\mathcal{E}_q[g(\overline{p{:}R}, R', \overline{R}).L''_q] \Uparrow q - b_q(p)) \quad \text{(B.156)}$$

Case a: Equation (B.156), case assumption and Definition 39 (-) (queue is empty). Wolg no multisend labels in the queue:

$$\mathsf{unf}(\mathcal{E}_p[g(\overline{p{:}r}, R', \overline{R}).L''_p] \Uparrow p) \bowtie \mathsf{unf}(\mathcal{E}_q[g(\overline{p{:}r}, R', \overline{R}).L''_q] \Uparrow q) \quad \text{(B.157)}$$

$$\mathsf{unf}(\mathcal{T}_p[g(\overline{p{:}r}, R', \overline{R}).(L''_p \Uparrow q)]) \bowtie \mathsf{unf}(\mathcal{T}_q[g(\overline{p{:}r}, R', \overline{R}).(L''_q \Uparrow p)]) \quad \text{(Definition 36 } (\Uparrow)) \quad \text{(B.158)}$$

$$(L''_p \Uparrow q) \bowtie (L''_q \Uparrow p) \quad \text{(Definition 40 (duality), Definition 41 } (\mathsf{unf}())) \quad \text{(B.159)}$$

after reduction for p:

$$\mathsf{unf}(\mathcal{E}_p[L''_p] \Uparrow p - b_p(q)) \quad \text{(B.160)}$$

$$\mathsf{unf}(\mathcal{E}_p[L''_p] \Uparrow p) \quad b_p(q) = \varepsilon \quad \text{(B.161)}$$

$$\mathsf{unf}(\mathcal{T}_p[L''_p \Uparrow q]) \quad \text{(B.162)}$$

after reduction for q:

$$\mathsf{unf}(\mathcal{E}_q[L''_q] \Uparrow q - b_q(p)) \quad \text{(B.163)}$$

$$\mathsf{unf}(\mathcal{E}_q[L''_q] \Uparrow p) \quad b_q(p) = \varepsilon \quad \text{(B.164)}$$

$$\mathsf{unf}(\mathcal{T}_q[L''_q \Uparrow p]) \quad \text{(B.165)}$$

By Equation (B.158) and Equation (B.159) we have for Equation (B.162) and Equation (B.165)

$$\mathsf{unf}(\mathcal{T}_p[L''_p \Uparrow q]) \bowtie \mathsf{unf}(\mathcal{T}_q[L''_q \Uparrow p]) \quad \text{(B.166)}$$

Case b: Similar to case Case a (the queues can potentially contain labels which can be cleaned).

Duality for s_2: Follows directly from projection of G.

Preservation of Definition 43 (Inter. Coherence): The reduction adds a new leaf node s_2 to session s which involves $\widetilde{p}_2$. We have $\widetilde{p}_2 \subseteq \widetilde{p} \subseteq \widetilde{p}_1$. Therefore, Definition 43 (Inter. Coherence) is preserved.

- Case [[RECV-HDL]]: Wolg by reduction:

$$\Theta = \Theta \tag{B.167}$$
$$\mathcal{F} = \mathcal{F}' \tag{B.168}$$
$$\Delta = s[p] : r_q?l_1.L, \Delta_c \tag{B.169}$$
$$\Delta' = s[p] : L, \Delta_c \tag{B.170}$$
$$\Sigma = s[p] : (L'' \text{ with } q{:}r_q?\{l_i : L_i\}_{i\in I}, b_p) \cup_{i\in I} \{s_i[p] : (L_i, b_i) \cup \Sigma_c \tag{B.171}$$
$$\Sigma' = s[p] : (- \text{ with } L_1, b_p[q \mapsto \bar{l}])_{i\in I} \{s_i[p] : (\mathsf{end}_{L_i}, b_i)\} \cup \Sigma_c \tag{B.172}$$
$$(q \mapsto l \cdot \bar{l}) \in b \tag{B.173}$$
$$s \rightsquigarrow_{\Theta}^{+} = \{s_i\}_I \tag{B.174}$$
$$b'_p = b_p[q \mapsto \bar{l}] \tag{B.175}$$

Preservation of Def. 42 (Intra. Coherence): The Def. 42 (Coherence).2 follows from Lemma 10.

Case distinction on p, q for Def. 42 (Coherence).1.

Case a: p, q both unsuspected. By well-formedness q is the monitor in session s.

By assumption before reduction

$$\mathsf{unf}(\{L'' \text{ with } q{:}r_q?\{l_i : L_i\}\} \Uparrow q - b_p(q)) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \tag{B.176}$$

Definition 36 ($\Uparrow$) on Equation (B.176) ($b_p(q) = l_1 \cdot \bar{l}$):

$$\mathsf{unf}(\{- \text{with } ?\{l_i : T_i\}\} - b_p(q)) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \tag{B.177}$$

Definition 39 (-) (first part) on Equation (B.177):

$$\mathsf{unf}(\{- \text{with } ?\{l_i : T'_i\}\} - [q \mapsto l_1 \cdot \bar{l}]) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \tag{B.178}$$

Definition 39 (-), Lemma 11 (second part) on Equation (B.178):

$$\mathsf{unf}(\{- \text{with } T'_1\} - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(L_q \Uparrow p - b_q(p)) \tag{B.179}$$

By [[RECV-HDL]], Def. 42 (Coherence).3 (a.o., queue labels adhere to well-formedness), p is in normal activity (before the reduction), where as q is in the failure handling activity, together with Definition 36 ($\Uparrow$) and Definition 39 (-) (clean) on Equation (B.179)

$$\mathsf{unf}(\{- \text{with } T'_1\} - [q \mapsto \bar{l}]) \bowtie \mathsf{unf}(\{- \text{with } T_q\}) \tag{B.180}$$

After reduction for p, q:

$$\mathsf{unf}(-\text{ with } L_1 \Cap q - b_p[q \mapsto \bar{l}](q)) \qquad \mathsf{unf}(L_q \Cap p - b_q(p)) \tag{B.181}$$

Definition 36 ($\Cap$) and Definition 39 (-) (right side) on Equation (B.181):

$$\mathsf{unf}(-\text{ with } T_1 - b_p[q \mapsto \bar{l}](q)) \qquad \mathsf{unf}(\{-\text{with } T_q\}) \tag{B.182}$$

Definition 36 ($\Cap$) and Definition 39 (-) (left side) on Equation (B.182):

$$\mathsf{unf}(-\text{ with } T_1 - b_p[q \mapsto \bar{l}](q)) = \mathsf{unf}(-\text{ with } T_1' - [q \mapsto \bar{l}](q)) \tag{B.183}$$

By Equation (B.180) for Equations (B.182) and (B.183):

$$\mathsf{unf}(-\text{ with } T_1 - b_p[q \mapsto \bar{l}](q)) \bowtie \mathsf{unf}(\{-\text{with } T_q\}) \tag{B.184}$$

Def. 42 (Coherence).1 for a p' with p' unsuspected, $(\widetilde{p}, \_) = \Theta(s)$ and $p' \in \widetilde{p}$:
Before reduction:

$$\mathsf{unf}(L'' \text{ with } q{:}r_q?\{l_i : L_i\}_{i \in I} \Cap p' - b_p(p')) \bowtie \mathsf{unf}(L_{p'} \Cap p - b_{p'}(p)) \tag{B.185}$$

Definition 36 ($\Cap$) on Equation (B.185) (left side)

$$\mathsf{unf}(-\text{ with } T_{(pp')_2} - b_p(p')) \bowtie \mathsf{unf}(L_{p'} \Cap p - b_{p'}(p)) \tag{B.186}$$

Case distinction on if p' is in the default activity or the failure handling activity.

Case i: Case p' in default activity (in partial types), i.e. we can ignore the multisend part of Definition 39 (-) since we have no label associated with failure handling in the queue:

Definition 42 (Intra. Coherence) and using Definition 39 (-) and Definition 36 ($\Cap$) on Equation (B.186)

$$\mathsf{unf}(-\text{ with } T_{(pp')_2}) \bowtie \mathsf{unf}(\{T_{(p'p)} \text{ with } T'_{(p'p)}\}) \tag{B.187}$$

By Equation (B.187), Definition 41 ($\mathsf{unf}()$) and Definition 40 (duality)

$$\mathsf{unf}(T_{(pp')_2}) \bowtie \mathsf{unf}(T'_{(p'p)}) \tag{B.188}$$

After the reduction for $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$\mathsf{unf}(-\text{ with } L_1 \Cap p' - b'_p(p')) \qquad \mathsf{unf}(L_{p'} \Cap p - b_{p'}(p)) \tag{B.189}$$

Definition 18 (projection) and Definition 36 ($\Uparrow$) ($L_i \Uparrow p' = L_j \Uparrow p'$) on Equation (B.189):

$$\mathsf{unf}(\{-\mathsf{with}\ T_{(pp')_2}\} - b'_p(p')) \qquad \mathsf{unf}(\{T_{(p'p)}\ \mathsf{with}\ T'_{(p'p)}\}) \tag{B.190}$$

Using Equation (B.187), assumption p' is in the normal activity and - (clean) on Equation (B.190):

$$\mathsf{unf}(\{-\mathsf{with}\ T_{(pp')_2}\}) \qquad \mathsf{unf}(\{T_{(p'p)}\ \mathsf{with}\ T'_{(p'p)}\}) \tag{B.191}$$

By Equation (B.188) Def. 42 (Coherence).1 holds after reduction.

Case ii: Case p' in failure activity (in partial types):

Case distinction on if we have $p, p' \in R_{ids}$ and the receive belong to a multisend to R; or we have a normal receive.

(Normal receive): Definition 39 (-) and Definition 36 ($\Uparrow$) on Equation (B.186)

$$\{-\mathsf{with}\ T_{(pp')_6}\} \bowtie \{-\mathsf{with}\ T_{(p'p)}\} \tag{B.192}$$

By Equation (B.192) and Definition 40 (duality) for :

$$T_{(pp')_6} \bowtie T_{(p'p)} \tag{B.193}$$

After the reduction for $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$\mathsf{unf}(-\ \mathsf{with}\ L_1 \Uparrow p' - b'_p(p')) \qquad \mathsf{unf}(L_{p'} \Uparrow p - b_{p'}(p)) \tag{B.194}$$

Definition 18 (projection) and Definition 36 ($\Uparrow$) ($L_i \Uparrow p' = L_j \Uparrow p'$) on Equation (B.199):

$$\mathsf{unf}(\{-\mathsf{with}\ T_{(pp')_2}\} - b'_p(p')) \qquad \{-\mathsf{with}\ T_{(p'p)}\} \tag{B.195}$$

Using Equations (B.186), (B.192) and (B.195), assumption p' in failure handling.

$$\{-\mathsf{with}\ T_{(pp')_6}\} \qquad \{-\mathsf{with}\ T_{(p'p)}\} \tag{B.196}$$

By Equation (B.193) Def. 42 (Coherence).1 holds after reduction.

(Multisend receive): By coherence and case before reduction:

$$\mathsf{unf}(-\ \mathsf{with}\ q\underline{?}\{l_i : T_{(pp')_i}\} - b_p(p')) \bowtie \mathsf{unf}(-\ \mathsf{with}\ T_{p'p} - b_{p'}(p)) \tag{B.197}$$

One step Definition 39 (-) (first part) on Equation (B.197) (left side):

$$\mathsf{unf}(-\ \mathsf{with}\ q\underline{?}\{l_i : T_{(pp')_i}\} - b_p(p')) = \mathsf{unf}(-\ \mathsf{with}\ T_{(pp')_1} - b_p[q \mapsto \bar{l}](p')) \tag{B.198}$$

After the reduction for $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$\mathsf{unf}(-\text{ with } L_1 \Uparrow p' - b'_p(p')) \qquad \mathsf{unf}(L_{p'} \Uparrow p - b_{p'}(p)) \tag{B.199}$$

By Definition 36 ($\Uparrow$) ($b'_p = b_p[q \mapsto \bar{l}]$):

$$\mathsf{unf}(-\text{ with } T_{(pp')_1} - b'_p(p')) \qquad \mathsf{unf}(-\text{ with } T_{p'p} - b_{p'}(p)) \tag{B.200}$$

Equation (B.200) is dual by Equation (B.198).

Case b: $p \in \mathcal{F} \ \wedge\ q \notin \mathcal{F}$. Straight forward since there is no change at q's "side".

Case c: p, q both suspected, trivially as duality does not need to hold if one side is suspected.

Case d: $p \notin \mathcal{F} \ \wedge\ q \in \mathcal{F}$.[2] Wolg $p' \notin \mathcal{F}$ with

$$s[p'] : (L_{p'}, b_{p'}) \in \Sigma \tag{B.201}$$

In $(\Delta, \Sigma, \mathcal{F}, \Theta)$ by Def. 42 (Coherence).1:

$$\mathsf{unf}(L'' \text{ with } q{:}r_q?\{l_i : L_i\}_{i\in I} \Uparrow p' - b_p(p')) \bowtie \mathsf{unf}(L_{p'} \Uparrow p - b_{p'}(p)) \tag{B.202}$$

Case distinction based on p' failure handling:

Case 1: Case p' in normal activity (partial types) and case on Equation (B.202):

$$\mathsf{unf}(L'' \text{ with } q{:}r_q?\{l_i : L_i\}_{i\in I} \Uparrow p' - b_p(p')) \bowtie \mathsf{unf}(L'_{p'} \text{ with } L''_{p'} \Uparrow p - b_{p'}(p)) \tag{B.203}$$

Definition 36 ($\Uparrow$) on Equation (B.202)

$$\mathsf{unf}(-\text{ with } T_1 - b_p(p')) \bowtie \mathsf{unf}(T'_{p'} \text{ with } T''_{p'} - b_{p'}(p)) \tag{B.204}$$

Definition 42 (Intra. Coherence), Definition 39 (-) and case assumption (e.g. no failure handling multisend labels in queues) on Equation (B.204):

$$\mathsf{unf}(-\text{ with } T_1) \bowtie \mathsf{unf}(T'''_{p'} \text{ with } T''_{p'}) \tag{B.205}$$

After reduction for $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$:

$$\mathsf{unf}(\{-\text{with } L_1\} \Uparrow p' - b[q \mapsto \bar{l}](p')) \qquad \mathsf{unf}(L'_p \Uparrow p - b_{p'}(p)) \tag{B.206}$$

[2] Since q is suspected coherence does not define duality between p and q, however we need to ensure that the message from q does not disturb duality between p and other participants in that session.

Definition 18 (projection) and Definition 36 ($\Vert$) ($L_i \Vert p' = L_j \Vert p'$) and Equation (B.206)

$$\mathsf{unf}(- \text{ with } T_1 - b[q \mapsto \bar{l}](p')) \qquad \mathsf{unf}(T'_{p'} \text{ with } T''_{p'} - b_{p'}(p)) \tag{B.207}$$

Definition 39 (-) (c.f. clean), Equation (B.206) on Equation (B.207)

$$\mathsf{unf}(- \text{ with } T_1) \qquad \mathsf{unf}(T'''_{p'} \text{ with } T''_{p'}) \tag{B.208}$$

Definition 40 (duality) and Equation (B.205):

$$\mathsf{unf}(- \text{ with } T_1) \bowtie \mathsf{unf}(T'''_{p'} \text{ with } T''_{p'}) \tag{B.209}$$

Def. 42 (Coherence).1 holds after reduction.

Case 2: Case p' in active failure handling. See above and Case ii.

Preservation of Definition 43 (Inter. Coherence): Similar to argumentation in case [[MON]].

□

Proof: Type Preservation

Lemma 14 (Inversion lemma processes)**.** *Inversion lemma for processes.*

1. *If* $\Gamma, \Sigma \vdash (H_1 \cdot .. \cdot H_n, p) \triangleright \Delta$ *then* $s : \mathcal{G} = \Gamma$, $p{:}R : \{L_i\}_{i \in I} = \Sigma$, Δ *is end only,* $p \in R_{ids}$, $\mathcal{L} = \mathcal{G} \upharpoonright R$, $I = \{1, .., n\}$, $\forall i \in I.\ H_i = [L_i]\lambda x_i.\ P_i \ \wedge\ s : \mathcal{G} \vdash H_i \triangleright \emptyset$ *and* $\forall g \in \mathcal{L}.\ \{L_i\}_{i \in I} \vdash g$.
2. *If* $\Gamma, \Sigma \vdash c[r]!l.P \triangleright \Delta$ *then* $\Delta = c : r!l.L, \Delta_{\mathsf{end}}$ *and* $\Gamma, \Sigma \vdash P \triangleright c : L, \Delta_{\mathsf{end}}$.
3. *If* $\Gamma, \Sigma \vdash c[R]!l.P \triangleright \Delta$ *then* $\Delta = c : R!l.L, \Delta_{\mathsf{end}}$ *and* $\Gamma, \Sigma \vdash P \triangleright c : L, \Delta_{\mathsf{end}}$.
4. *If* $\Gamma, \Sigma \vdash c[r]?l.P \triangleright \Delta$ *then* $\Delta = c : r?l.L, \Delta_{\mathsf{end}}$ *and* $\Gamma, \Sigma \vdash P \triangleright c : L, \Delta_{\mathsf{end}}$.
5. *If* $\Gamma, \Sigma \vdash c[r]{\downarrow}.P \triangleright \Delta$ *then* $\Delta = c : r{\downarrow}.L, \Delta_{\mathsf{end}}$ *and* $\Gamma, \Sigma \vdash P \triangleright c : L, \Delta_{\mathsf{end}}$.
6. *If* $\Gamma, \Sigma \vdash c[\overline{p{:}r}; R; \overline{R}](g) \triangleright \Delta$ *then* $\Delta = c : g(\bar{r}; R, \overline{R})$, $\mathcal{G} \in \Gamma$ *and* $g(\bar{r}; \underline{r}; \overline{R}) = ... \in \mathcal{G}$.
7. *If* $\Gamma \vdash [L]\lambda x.\ P \triangleright \emptyset$ *then* $\Gamma \vdash P \triangleright x : L$ *and* $\mathsf{wf}(L)$
8. *If* $\Gamma, \Sigma \vdash \mathsf{loop} \triangleright \Delta$ *then* $\Sigma = \emptyset$ *and* Δ *is end only.*

Lemma 15 (Inversion lemma network)**.** *Inversion lemma for the event handling typing.*

1. *If* $\Gamma, \Sigma \vdash s[p] : (L, b) \triangleright \Delta$ *then* $\Sigma = s[p] : (L, b)$*,* Δ *is* end*-only.*
2. *If* $\Gamma, \Sigma \vdash N_1 \mid\mid N_2 \triangleright \Delta$ *then* $\Gamma, \Sigma_1 \vdash N_1 \triangleright \Delta_1$*,* $\Gamma, \Sigma_2 \vdash N_2 \triangleright \Delta_2$*,* $\Sigma = \Sigma_1 \cdot \Sigma_2$ *and* $\Delta = \Delta_1 \cdot \Delta_2$*.*
3. *If* $\Gamma, \emptyset \vdash 0 \triangleright \Delta$ *then* Δ *is end only*

Lemma 16 (Inversion lemma environments)**.** *Inversion lemma for the a network with the environments* $\mathcal{F}$ *and* Θ*.*

1. *If* $\vdash (\Theta, \mathcal{F}, (\nu(s : \mathcal{G}))\, N)$ *then* $s : \mathcal{G}, \Sigma \vdash N \triangleright \Delta$*,* $\mathsf{wf}(\mathcal{G})$ *and* $(\Delta, \Sigma, \mathcal{F}, \Theta)$ *coherent*

Lemma 17 (Substitution Lemma)**.**

1. *If* $\Gamma, \Sigma \vdash P \triangleright x : L, \Delta$ *then* $\Gamma, \Sigma \vdash P\{s[p]/x\} \triangleright s[p] : L, \Delta$
2. *If* $\Gamma, \Sigma \vdash (H_1 \cdot \ldots \cdot H_n, p) \triangleright \Delta_{\mathsf{end}}$*, and* $H_i = [L_i]\lambda x_i.\, P_i$ *then* $\Gamma, \Sigma \vdash P_i\{s[p]/x_i\}\{(H_1 \cdot \ldots \cdot H_n, p)/\mathtt{loop}\} \triangleright s[p] : L_i \cdot \Delta_{\mathsf{end}}$*,*

Proof. Induction over type derivation. □

Lemma 18 (Type Preservation under Equivalence)**.** *Given,* $\Gamma, \Sigma \vdash N \triangleright \Delta$ *and* $N \equiv N'$*, then* $\Gamma, \Sigma \vdash N' \triangleright \Delta$

Proof. Induction over $\equiv$. □

Lemma 19 (Type Preservation)**.** *Assume* $\Gamma, \Sigma \vdash N \triangleright \Delta$*, runtime environments* Θ *and* $\mathcal{F}$*, and* $(\Theta, \mathcal{F}, N) \to (\Theta', \mathcal{F}', N')$*, then* $(\Delta, \Sigma, \mathcal{F}, \Theta) \to (\Delta', \Sigma', \mathcal{F}', \Theta')$ *and* $\Gamma, \Sigma' \vdash N' \triangleright \Delta'$*.*

Proof. The proof is done by induction **over the derivation of** $\to$, i.e., over the operational semantic for networks.

For readability, we sometimes drop Θ and or $\mathcal{F}$ from the reduction $(\Theta, \mathcal{F}, N) \to (\Theta', \mathcal{F}', N')$ if Θ and/or $\mathcal{F}$ does not change in the reduction step.

Case Send:

The reduction is (wolg we assume l_1):

$$\begin{aligned} & s[p][r]!l_1 P \mid\mid s[p] : (\mathcal{E}[q{:}r!\{l_i : L_i\}_{i \in I}], b) \mid\mid s[q] : (L_q, b_q) && \to \\ & P \mid\mid s[p] : (\mathcal{E}[L_1], b_p) \mid\mid s[q] : (L, b_q[p \mapsto b_q(p) \cdot l_1]) && \end{aligned} \tag{B.210}$$

By assumption:

$$\Gamma, \Sigma \vdash s[p][r]!l_1 P \mid\mid s[p] : (\mathcal{E}[q{:}r!\{l_i : L_i\}_{i\in I}], b_p) \mid\mid s[q] : (L_q, b_q) \triangleright \Delta \tag{B.211}$$

By twice Lemma 15.2 (Inversion) we get:

$$\Gamma, \Sigma_1 \vdash s[p][r]!l_1 P \triangleright \Delta_1 \tag{B.212}$$
$$\Gamma, \Sigma_2 \vdash s[p] : (\mathcal{E}[q{:}r!\{l_i : L_i\}_{i\in I}], b_p) \triangleright \Delta_2 \tag{B.213}$$
$$\Gamma, \Sigma_3 \vdash s[q] : (L_q, b_q) \triangleright \Delta_3 \tag{B.214}$$
$$\Sigma = \Sigma_1, \Sigma_2, \Sigma_3 \tag{B.215}$$
$$\Delta = \Delta_1, \Delta_2, \Delta_3 \tag{B.216}$$

By Lemma 14.2 (Inversion) on Equation (B.212):

$$\Delta_1 = s[p] : r!\{l_1 : L_1\} \quad \Gamma, \Sigma_1 \vdash P \triangleright s[p] : L_1 \cdot \Delta_{\mathsf{end}} \tag{B.217}$$

By Lemma 15.1 (Inversion) on Equation (B.213)

$$\Sigma_2 = s[p] : (\mathcal{E}[q!\{l_i : L_i\}_{i\in I}], b_p) \quad \Delta_2 \text{ end only} \tag{B.218}$$

By Lemma 14.1 (Inversion) on Equation (B.214)

$$\Sigma_3 = s[q] : (L_q, b_q) \quad \Delta_3 \text{ end only} \tag{B.219}$$

We now show $(\Delta, \Sigma, \mathcal{F}, \Theta) \to (\Delta', \Sigma', \mathcal{F}, \Theta')$ via the typing environments reduction rule [[SEND]] as p. After that we will show that $(\Delta', \Sigma', \mathcal{F}, \Theta')$ types Equation (B.210)

By [[SEND]] (see Figure B.14) for p on $(\Delta,\ \Sigma,\ \mathcal{F},\ \Theta)$ we get:

$$\Sigma' = \Sigma_1, \begin{array}{l} s[p] : (\mathcal{E}[L_1], b_p), \\ s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_1]) \end{array}$$ By [[SEND]] on Equation (B.215) with Equation (B.215), Equation (B.218), Equation (B.219)

$$\Delta' = s[p] : L, \underbrace{\Delta_2, \Delta_3}_{\text{end only}} \cdot \Delta_{\mathsf{end}}$$ By [[SEND]] on Equation (B.216) with Equation (B.217)

$$\Theta' = \Theta$$ By [[SEND]]

$$\mathcal{F}' = \mathcal{F}$$ By [[SEND]]

(B.220)

We now show that we can type $P \mid\mid s[p] : (\mathcal{E}[L_1], b_p) \mid\mid s[q] : (L, b_q[p \mapsto b_q(p) \cdot l_1])$ with Equation (B.220) (we use: $P_c = s[p] : (\mathcal{E}[L_j], b_p) \mid\mid s[q] : (L, b_q[p \mapsto b_q(p) \cdot l_1])$). Typing

for the configuration of p:

$$\frac{\textsc{TCfg}}{\Gamma, \underbrace{s[p] : (\mathcal{E}[L_1], b_p)}_{\Sigma_2'} \vdash s[p] : (\mathcal{E}[L_j], b_p) \ \triangleright \ \Delta_{\mathsf{end}}} \tag{B.221}$$

Typing of p's event loop with active handler:

$$\frac{\begin{array}{c}\textsc{By Equation} \text{ (B.217)} \\ \dots\end{array}}{\Gamma, \Sigma_1 \vdash P \ \triangleright \ s[p] : L \cdot \Delta_{\mathsf{end}}} \tag{B.222}$$

Typing of q's configuration:

$$\frac{\textsc{TCfg}}{\Gamma, \underbrace{s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_1])}_{\Sigma_3'} \vdash s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_1]) \ \triangleright \ \Delta_{\mathsf{end}}} \tag{B.223}$$

$$\textsc{TPar}\ \frac{\textsc{TPar}\ \frac{\textit{Equation} \text{ (B.222)} \qquad \frac{\textit{Equation} \text{ (B.221)} \qquad \textit{Equation} \text{ (B.223)}}{\Gamma, \Sigma_2', \Sigma_3' \vdash P_c \ \triangleright \ \Delta_{\mathsf{end}}}\ \textsc{TPar}}{\Gamma, \Sigma' \vdash P \mid\mid s[p] : (\mathcal{E}[L_1], b_p) \mid\mid s[q] : (L, b_q[p \mapsto b_q(p) \cdot l_1]) \ \triangleright \ s[p] : L \cdot \Delta_{\mathsf{end}}} \tag{B.224}$$

Case Fire: From the case we have:

$$(\Theta, \mathcal{F}, (\overline{H}, p) \mid\mid s'[p] : (L, b_p)) \tag{B.225}$$

$$\rightarrow$$

$$(\Theta, \mathcal{F}, \underbrace{P\{s'[p]/x\}\{(\overline{H}, p)/\mathsf{loop}\}}_{N'} \mid\mid s'[p] : (L, b_p)) \tag{B.226}$$

with $[L_g]\lambda x.\, P \in \overline{H}$.

By assumption:

$$\Gamma, \Sigma \vdash (\overline{H}, p) \mid\mid s'[p] : (L, b_p) \ \triangleright \ \Delta \tag{B.227}$$

By Lemma 15.2 (Inversion) on Equation (B.227):

$$\Gamma, \Sigma_1 \vdash (\overline{H}, p) \ \triangleright \ \Delta_1 \tag{B.228}$$
$$\Gamma, \Sigma_2 \vdash s'[p] : (L, b_p) \ \triangleright \ \Delta_2 \tag{B.229}$$
$$\Delta = \Delta_1 \cdot \Delta_2 \tag{B.230}$$
$$\Sigma = \Sigma_1 \cdot \Sigma_2 \tag{B.231}$$

By Lemma 15.1 (Inversion) on Equation (B.229):

$$\Delta_2 = \Delta_{\mathsf{end}} \tag{B.232}$$

By Lemma 14.1 (Inversion) on Equation (B.228):

$$\begin{gathered} s : \mathcal{G} = \Gamma \quad p{:}R : \{L_i\}_{i \in I} = \Sigma_1 \quad \Delta_1 \text{ is end only} \quad \mathcal{L} = \mathcal{G} \upharpoonright R \\ \forall i \in I.\ H_i = [L_i]\lambda x_i.\ P_i \ \wedge \ s : \mathcal{G} \vdash H_i \ \triangleright \ \emptyset, \\ \forall g \in \mathcal{L}.\ \{L_i\}_{i \in I} \vdash g \quad p \in R_{ids} \end{gathered} \tag{B.233}$$

By Equation (B.233) and Fire we have:

$$s : \mathcal{G} \vdash [L_g]\lambda x.\ P \ \triangleright \ \emptyset \tag{B.234}$$

By Lemma 14 (Inversion) on Equation (B.234) we have:

$$s : \mathcal{G} \vdash P \ \triangleright \ x : L_g \text{ and } \mathsf{wf}(L_g) \tag{B.235}$$

By Lemma 17 (Substitution) we have:

$$\Gamma, \underbrace{\Sigma_1}_{p:R:\{L_i\}_{i \in I}} \ \vdash \ P\{s'[p]/x\}\{(\overline{H}, s[p])/\mathsf{loop}\} \ \triangleright \ s'[p] : L_g \cdot \underbrace{\Delta_1}_{\text{end only}} \tag{B.236}$$

We show $(\Delta,\ \Sigma,\ \mathcal{F},\ \Theta) \to (\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ by applying the rule [[FIRE]] and then showing that $(\Delta', \Sigma', \mathcal{F}, \Theta')$ types Equation (B.226).

Reduction rule [[FIRE]] applies to $(\Delta, \Sigma, \mathcal{F}, \Theta)$ with p activating it in session s' for handler L_g since: (a) by Equation (B.233) we have $L_g \in \{L_i\}_{i \in I}$, (b) by case reduction Fire we have $L \asymp L_g$ and s', (c) by case reduction Fire we have $\mathsf{fire}(L, L_g, b, \mathcal{F})$, (d) by Equation (B.233) and Equation (B.230):

$$\Delta \text{ is end only} \tag{B.237}$$

and (e) by Equation (B.237) there is no active endpoint for $s'[p]$ in Δ

After [[FIRE]] reduction:

$$\Sigma' = \Sigma \tag{B.238}$$
$$\Delta' = \Delta, s'[p] : L_g \tag{B.239}$$
$$\Theta' = \Theta \tag{B.240}$$
$$\mathcal{F}' = \mathcal{F} \tag{B.241}$$

We now show that we can type Equation (B.226) with the environments $(\Sigma,\ \Delta',\ \mathcal{F},\ \Theta)$

$$\frac{\text{By Equation (B.236)}}{\Gamma, p{:}R : \{L_i\}_{i\in I} \vdash N' \ \triangleright\ s'[p] : L_g, \Delta_\emptyset} \tag{B.242}$$

$$\frac{\text{By Equation (B.229)}}{\Gamma, \Sigma_2 \vdash s'[p] : (L, b_p) \ \triangleright\ \Delta_2}\ \textsc{TCfg} \tag{B.243}$$

$$\textsc{TPar}\ \frac{\textit{Equation}\ \text{(B.242)} \qquad \textit{Equation}\ \text{(B.243)}}{\Gamma, \underbrace{\Sigma_1 \cdot \Sigma_2}_{\Sigma} \vdash P' \mid\mid s'[p] : (L, b_p) \ \triangleright\ s'[p] : L_g \cdot \Delta_\emptyset} \tag{B.244}$$

Case Spawn: We show a part of this case the rest is relative straightforward.

From the case we have:

$$\begin{aligned}&(\Theta[s \mapsto (\widetilde{p}, \widetilde{s})], \mathcal{F}, \underbrace{\mid\mid_{p\in\widetilde{p}_1} \big((\overline{H}_p, p) \mid\mid s[p] : (\mathcal{E}_p[g(\overline{q{:}r}; R'; \overline{R}).L_p], b_p)\big)}_{N}) \to \\ &(\Theta', \mathcal{F}, \underbrace{\mid\mid_{p\in\widetilde{p}_1}\big((\overline{H}_p, p) \mid\mid s[p] : (L_p, b_p)\big) \mid\mid_{p\in\widetilde{p}_2} s'[p] : (g^{\bar{q};p_0}\lceil p, b_p)}_{N'})\end{aligned} \tag{B.245}$$

By reduction Spawn:

$$\Theta' = \Theta[s \mapsto (\widetilde{p}, \widetilde{s} \cup s'), s' \mapsto (\widetilde{p}_2, \emptyset)] \tag{B.246}$$
$$\widetilde{p}_1 = \bar{q} \cup (R'_{ids} \cup \overline{R}_{ids}) \setminus \mathcal{F} \tag{B.247}$$

By assumption:

$$\Gamma, \Sigma \vdash N \ \triangleright\ \Delta \tag{B.248}$$

By Equation (B.245) and deconstruction via Lemma 14 and Lemma 15:

$$\begin{aligned}\Sigma &= \{p{:}R_p : \{L_i\}_{i\in I_p}\}_{p\in\widetilde{p}_1} \cup \{s[p] : (L_p, b_p)\}_{p\in\widetilde{p}_1} \\ \Delta &= \Delta_{\mathsf{end}}\end{aligned} \tag{B.249}$$

By Equation (B.249) and that the preconditions on Spawn and [[SPAWN]] are roughly equal, [[SPAWN]] applies to $(\Delta, \Sigma, \mathcal{F}, \Theta)$. The extra condition $\Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \quad \forall p \in \widetilde{p}_1.\ \nexists s[p] \notin \mathtt{dom}(\Delta_1)$ of [[SPAWN]] is trivially satisfied as $\Delta = \Delta_{\mathsf{end}}$. With selecting the same s' and extended projection being deterministic it is easy to see that the result of [[SPAWN]] can be used to type P'.

Case Sess-GC: By case we have the reduction Sess-GC by that we know [[SESS-GC]] is applicable. The result of [[SESS-GC]] can be used to type N'

Case Root-GC: Similar to case Sess-GC.

Case Clean: Straightforward type deconstruction via Inversion Lemma followed by [[CLEAN]] which applies by case and easy to see that result of [[CLEAN]] can be used to type result of Clean.

Case Susp: No typing rule requires $\mathcal{F}$.

Case Mon: From the case the reduction is:

$$(\Theta, \mathcal{F}, \underbrace{s[p][r]{\downarrow}.P_c \mid\mid s[p] : (L' \text{ with } p{:}r{\downarrow}.L, \_) \mid\mid_{i \in I} s_i[p] : (L_i, \_)}_{N}) \tag{B.250}$$

$$\rightarrow$$

$$(\Theta, \mathcal{F}, P_c \mid\mid s[p] : (- \text{ with } L, \_) \mid\mid_{i \in I} s_i[p] : (\mathsf{end}_{L_i}, \_)) \tag{B.251}$$

By assumption

$$\Gamma, \Sigma \vdash N \ \triangleright\ \Delta \tag{B.252}$$

By reduction:

$$p' \in \mathcal{F} \qquad s \rightsquigarrow_{\Theta}^{+} = \{s_i\}_{i \in I} \tag{B.253}$$

By Equation (B.253), reduction Mon and Equation (B.252) we can apply [[MON]]. Furthermore, as [[MON]] mimics Mon we have the typing after reduction.

Case MSnd: By the case the reduction is:

$$(\Theta, \mathcal{F}, \underbrace{\begin{array}{l} \mid\mid s[p][R]!l_j.P_c \\ \mid\mid s[p] : (\mathcal{E}[R!\{l_i : L_i\}_{i \in I}], \_) \\ \mid\mid_{q \in \widetilde{p}} s[q] : (\_, b_q) \end{array}}_{N}) \rightarrow (\Theta, \mathcal{F}, \begin{array}{l} \mid\mid P_c \\ \mid\mid s[p] : (\mathcal{E}[L_j], \_) \\ \mid\mid_{p \in \widetilde{p}} s[q] : (\_, b'_q) \end{array}) \tag{B.254}$$

with

$$\widetilde{p} = R_{ids} \setminus \mathcal{F} \quad l_j \in \{l_i\}_{i \in I} \quad \forall q \in \widetilde{p}.\ b'_q = b_q[p \mapsto b_q(p) \cdot l_j] \tag{B.255}$$

By assumption,

$$\Gamma, \Sigma \vdash N \ \triangleright \ \Delta \tag{B.256}$$

deconstruction via Lemma 14 (Inversion) and Lemma 15 (Inversion) similar to case Send:

$$\begin{aligned} \Delta &= s[p] : R!l_j.L_c, \Delta_{\mathsf{end}} \\ \Sigma &= s[p] : (\mathcal{E}[R!\{l_i : L_i\}_{i \in I}], b) \cup_{i \in I} \ \{s[p_i] : (L_i, b_i)\} \cdot \Sigma'' \\ \Gamma, \Sigma'' &\vdash s[p][R]!l_j.P_c \ \triangleright \ s[p] : R!l_j.L_c, \Delta_{\mathsf{end}} \end{aligned} \tag{B.257}$$

With Equation (B.255) and Equation (B.257) we can apply [[MSend]] and have:

$$\begin{aligned} \Delta' &= s[p] : L_c \cdot \Delta_{\mathsf{end}} \\ \Sigma' &= s[p] : (\mathcal{E}[L_j], b) \cup_{i \in I} \ \{s[p_i] : (L_i, b_i')\} \cdot \Sigma'' \end{aligned} \tag{B.258}$$

We have:

$$\Gamma, \Sigma' \vdash N' \ \triangleright \ \Delta' \tag{B.259}$$

Unfold By the reduction via Unfold we have:

$$\underbrace{s[p] : (\mathcal{E}[\mu t.L], \_)}_{N} \rightarrow \underbrace{s[p] : (\mathcal{E}[L\{\mu t.L/t\}], \_)}_{N'} \tag{B.260}$$

By assumption,

$$\Gamma, \Sigma \vdash N \ \triangleright \ \Delta \tag{B.261}$$

By inversion lemma we have:

$$\begin{aligned} \Delta &= \emptyset \\ \Sigma &= s[p] : (\mathcal{E}[\mu \mathtt{t}.L], b) \\ \Theta &= \Theta \end{aligned} \tag{B.262}$$

With Equation (B.262) we can apply [[UNFOLD]] and have:

$$\begin{aligned} \Delta' &= \emptyset \\ \Sigma' &= s[p] : (\mathcal{E}[\{\mu \mathtt{t}.L/\mathtt{t}\}L], b) \end{aligned} \tag{B.263}$$

We have:

$$\Gamma, \Sigma' \vdash \ N' \ \triangleright \ \emptyset \tag{B.264}$$

PAR (induction case) By the case we have the reduction:

$$(\Theta_1, \mathcal{F}_1, \underbrace{N_1 \mid\mid N_2}_{N}) \to (\Theta_1, \mathcal{F}_1, N_1' \mid\mid N_2) \tag{B.265}$$

and

$$(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta_1', \mathcal{F}_1', N_1') \tag{B.266}$$

By assumption,

$$\Gamma, \Sigma \vdash N \ \triangleright \ \Delta \tag{B.267}$$

By Lemma 14 (Inversion), it exists $\Delta_1 \cdot \Delta_2 = \Delta$ and $\Sigma_1 \cdot \Sigma_2 = \Sigma$ with

$$\text{TPAR}\ \frac{\Gamma, \Sigma_1 \vdash N_1 \ \triangleright \ \Delta_1 \qquad \Gamma, \Sigma_2 \vdash N_2 \ \triangleright \ \Delta_2}{\Gamma, \Sigma \vdash N_1 \mid\mid N_2 \ \triangleright \ \Delta} \tag{B.268}$$

By induction hypnosis it exists $(\Delta_1',\ \Sigma_1',\ \mathcal{F}_1',\ \Theta_1')$ with

$$(\Delta_1,\ \Sigma_1,\ \mathcal{F}_1,\ \Theta) \to (\Delta_1',\ \Sigma_1',\ \mathcal{F}_1',\ \Theta_1') \tag{B.269}$$

and

$$\Gamma, \Sigma_1' \vdash N_1' \ \triangleright \ \Delta_1' \tag{B.270}$$

We show: $\Delta_1' \cdot \Delta_2$: By assumption on the structure of N (c.f. 4.4.3):

$$s[p] \in \mathtt{dom}(\Delta_i) \text{ and } s[p] \text{ is not end implies } p{:}R : \{L_i\}_{i \in I} \in \Sigma_i \tag{B.271}$$

Note [[FIRE]] is the only reduction with adds new elements to Δ.

By Equation (B.271) and no type environment reduction adds a new event loop type:

$$\forall s[p] \in \mathtt{dom}(\Delta_1' \setminus \Delta_{\mathsf{end}}).\ p{:}R : \{L_i\}_{i \in I} \in \Sigma_1 \ \wedge \ s[p] \notin \mathtt{dom}(\Delta_2 \setminus \Delta_{\mathsf{end}}) \tag{B.272}$$

By Equation (B.272),

$$\Delta_1' \cdot \Delta_2 \tag{B.273}$$

We show $\Sigma_1' \cdot \Sigma_2$: [[SPAWN]] is the only typeing environment reduction rule which adds new elements to Σ_i and [[SPAWN]] assumes a fresh s. Therefore:

$$\Sigma_1' \cdot \Sigma_2 \tag{B.274}$$

By Equation (B.273) and Equation (B.274):

$$\Gamma, \Theta_1', \Sigma_1' \cdot \Sigma_2 \vdash N_1' \mid\mid N_2' \ \triangleright \ \Delta_1' \cdot \Delta_2 \tag{B.275}$$

We now show that the following reduction is possible $(\Delta_1 \cdot \Delta_2, \Sigma_1 \cdot \Sigma_2, \mathcal{F}, \Theta) \rightarrow (\Delta_1' \cdot \Delta_2, \Sigma_1' \cdot \Sigma_2, \mathcal{F}, \Theta_1')$. The reductions [[SEND]],[[RECV]],[[MSEND]],[[UNFOLD]],[[MON]], [[RECV-HDL]] and [[SUSP]] are all not effected by larger session environments and or configuration environments.

The preconditions of the kind: "$\Delta = \Delta_1 \cdot \Delta_{\mathsf{end}} \quad s'[p] \notin \Delta_1$" are still valid in the composed environment, see argumentation above (e.g. Equation (B.271)). Therefore reductions [[FIRE]], [[SPAWN]], [[SESS-GC]],[[ROOT-GC]] and [[CLEAN]] also still apply.

□

Theorem 3 (Subject Reduction)**.** *Let* $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G}) N_1)$ *such that* $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G}) N_1) \rightarrow (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$. *Then* $\vdash (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$.

Proof. By reduction from assumption with RES,

$$(\Theta_1, \mathcal{F}_1, N_1) \rightarrow (\Theta_2, \mathcal{F}_2, N_2) \tag{B.276}$$

By Lemma 16 (Inversion),

$$s : \mathcal{G}, \Sigma_1 \vdash N_1 \ \triangleright \ \Delta_1, \mathsf{wf}(\mathcal{G}) \text{ and } (\Delta_1,\ \Sigma_1,\ \mathcal{F}_1,\ \Theta_1) \text{ coherent} \tag{B.277}$$

By Equation (B.276) and Equation (B.277): Lemma 19 (Type Preservation) applies to reduction Equation (B.276):

$$\begin{gathered}(\Delta_1,\ \Sigma_1,\ \mathcal{F}_1,\ \Theta_1) \rightarrow (\Delta_2,\ \Sigma_2,\ \mathcal{F}_2,\ \Theta_2) \\ \text{with} \\ \Gamma, \Sigma_2 \vdash N_2 \ \triangleright \ \Delta_2\end{gathered} \tag{B.278}$$

Lemma 13 (Preservation of Coherence) applies by Equation (B.277) to Equation (B.278):

$$(\Delta_2,\ \Sigma_2,\ \mathcal{F}_2,\ \Theta_2) \text{ is coherent} \tag{B.279}$$

By TSYSTEM applies to Equation (B.276) by Equation (B.278) and Equation (B.279):

$$\vdash (\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\ N_2) \tag{B.280}$$

□

Safety Corollary

The safety corollaries follow directly from Theorem 3 (Subject Reduction).

Definition 46 (Reception Error)**.** *Given an unsuspected participant* $(\overline{H}, p)$*, a session* s *with configuration* $s[p] : (L, b)$*. Then* p *has a reception error in* s *if one of the following cases is true:*

- $L = q{:}r?\{l_i : L_i\}_{i \in I}$ with L'_p, $b(q) = l_j \cdot \bar{l}$, $q \notin \mathcal{F}$, $l_j \notin \{l_i\}_{i \in I}$ *and* $\nexists l' \in b.\ l' \in L'_p$
- $L = -$ with $q{:}r?\{l_i : L_i\}_{i \in I}$, $b(q) = l_j \cdot \bar{l}$, $q \notin \mathcal{F}$, $l_j \notin \{l_i\}_{i \in I}$ *and* $l_j \in \{L_i\}_{i \in I}$

Reception error only covers the case where both sender and receiver are unsuspected. Failure cases around suspicion will be defined shortly. The first case defines a reception error in the default activity, which occurs if the first label in the input queue does not match the receive, can not be cleaned, and is also not consumed in the failure handling; because of asynchrony, the send can be in failure handling, and p will consume the message once it is in failure handling. The second case is for failure handling, where we have a reception error if the first label can not be consumed and cleaning does not apply.

We now define when a participant is stuck, i.e., cannot progress (take a reduction step), e.g., because of a failure or a blocking action like spawn.

Definition 47 (Stuck)**.** *Given* $(\Theta, \mathcal{F}, (\nu(s' : \mathcal{G}))\ N)$*, an unsuspected participant* $(\overline{H}, p)$ *and the configuration* $s[p] : (L, b)$*. Then* p *is stuck if one of the following is true:*

1. $L = g(\overline{p{:}r}; R'; \overline{R}).L_p$ with L'_p, $\exists q \in (\bar{p} \cup R'_{ids} \cup_{R \in \overline{R}} R_{ids}) \setminus \mathcal{F}$ *with either*
   a) $s[q] : (p{:}r?\{l_i : L_i\}_{i \in I}$ with $L'_q, b_q)$ *and* $b_q(p) = \epsilon$ *or*
   b) $s[q] : (g'(\overline{p'{:}r'}; R''; \overline{R'})$ with $L'_q, b_q)$, $g' \neq g$ *and* $p \in\ (\bar{p} \cup R''_{ids} \cup_{R \in \overline{R'}} R_{ids}) \setminus \mathcal{F}$
2. $L = -$ with $g(\overline{p{:}r}; R; \overline{R}).L_p$, $\exists q \in (\bar{p} \cup R'_{ids} \cup_{R \in \overline{R}} R_{ids}) \setminus \mathcal{F}$ *with either*
   a) $s[q] : (-$ with $p{:}r?\{l_i : L_i\}_{i \in I}, b_q)$ *and* $b_q(p) = \epsilon$ *or*
   b) $s[q] : (-$ with $g'(\overline{p'{:}r'}; R''; \overline{R'}), b_q)$, $g \neq g'$ *and* $p \in\ (\overline{p'} \cup R''_{ids} \cup_{R \in \overline{R'}} R_{ids}) \setminus \mathcal{F}$
3. $L = q{:}r_q?\{l_i : L_i\}_{i \in I}$ with L'_p *with* $b(q) = \epsilon$ *and* $\nexists l \in b.\ l \in L'_p$ *if* q *is unsuspected,* $s[q] : (p{:}r?\{l_i : L_i\}_{i \in I}$ with $L'_q, b_q)$, $\nexists l \in b_q.\ l \in L'_q$, *and* $b_q(p) = \epsilon$
4. $L = -$ with $q{:}r_q?\{l_i : L_i\}_{i \in I}$ *with* $b(q) = \epsilon$ *if* q *is unsuspected and* $s[q] : (-$ with $p{:}r?\{l_i : L_i\}_{i \in I}, b_q)$ *and* $b_q(p) = \epsilon$

$$
\begin{array}{lll}
G ::= \{[\widetilde{p}]G \text{ with}[\widetilde{p}]_{p:r@p:r}\ G\} & & p\ ::= 1 \mid 2 \mid .. \\
\quad p{:}r \rightarrow z\{l_i : G_i\}_{i\in I} \mid g(\overline{p{:}r}; R; \overline{R}).G \mid \mu \mathtt{t}.G \mid \mathtt{t} \mid \mathsf{end} \mid & & R ::= W \mid M \mid ... \\
\quad p{:}r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i\in I} & & z\ ::= p{:}r \mid R
\end{array}
$$

Figure B.17.: Extended global type.

(1) - (4) state that an unsuspected participant p is stuck if p wants to perform a (blocking) spawn/receive action and another unsuspected participant that is part of that action wants to perform a blocking action which involves p.

We now define faulty intersession relations.

Definition 48 (Failure not covered)**.** *Given* $(\Theta, \mathcal{F}, (\nu(s'' : \mathcal{G}))N)$*, an unsuspected participant* $(\overline{H}, p)$ *and the configuration* $s[p] : (L, b)$*. Then* p *has a not covered failure in* s *if one of the following is true:*

1. $L = \mathcal{E}[q{:}r_q?\{l_i : L_i\}_{i\in I}]$*,* $b(q) = \varepsilon$*,* q *is suspected and there is no session* s' *in* N *in which* q *is monitored and* $s = s' \vee s \in (s' \rightsquigarrow^{+}_{\Theta})$*.*
2. $L = \mathcal{E}[g(\overline{p{:}r}; R; \overline{R})]$*,* $q \in \bar{p}$*,* q *is suspected and there is no session* s' *in* N *in which* q *is monitored and* $s = s' \vee s \in s' \rightsquigarrow^{+}_{\Theta}$*.*

An unsuspected participant has a failure not covered if it is waiting on a failed participant and there is no (transitively) parent session which will take over because of that failure.

Corollary 3.1 (Communication Safety)**.** *Let* $\vdash (\Theta, \mathcal{F}, (\nu s : \mathcal{G})\ N))$*. For every session* s *in* Θ *and unsuspected* p *in* s*,* p *has the following properties: (i)* p *does not have a* reception error*; (ii)* p *is not* stuck*; and (iii)* p *does not have a* non-covered failure*.*

Proof. Follows directly from coherence. □

B.6.4. Property: fidelity

Extended Global Type

Figure B.17 shows the extended global type G, i.e., we reuse the symbol of the "normal" global type (cf. Figure 4.7). The main differences compared to the global type are that it uses participant ids and role names instead of just role names and contains new and modified constructs that occur while reducing an extended global type. The differences are highlighted in gray. The two new or modified constructs are $\{[\widetilde{p}_1]G_1 \text{ with}[\widetilde{p}_2]_{p_f:r_f@p_m:r_m}\ G_2\}$

and $p{:}r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i\in I}$. $\{[\widetilde{p}_1]G_1 \text{ with}[\widetilde{p}_2]_{p_f:r_f@p_m:r_m} G_2\}$ describes the *with* type, which is a representation of the body of a subprotocol. It records the default activity G_1 and the failure handling activity G_2. The sets $\widetilde{p}_1$ and $\widetilde{p}_2$ contain the participant ids of participants in the normal activity, respectively in failure handling. Further, it records the participant p_f, whose failure is monitored, and the participant p_m, who monitors that failure. $p{:}r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i\in I}$ indicates that p has sent the label l_j to z. Further, all participants in $\widetilde{p}$ have already received the label.[3]

Extended Well-Formedness

Definition 49 (Extended Well-Formedness)**.** *A given extended global type G is well-formed if all the following conditions are true.*

1. *The structure of G conforms to the structure imposed by the protocol definition and the global type reduction semantic. I.e., there is a G' which is derived from a protocol or an element of a top-level global type, and either $G' = G$ or $\mathcal{F}, G \rightarrow^* \mathcal{F}, G'$.*
2. *If $G = \{[\widetilde{p}]G' \text{ with}[\widetilde{p}']_{p:r@p_m:r_m} G'\}$ then the monitor p_m, must immediately send a label, in G', to all participants in $\widetilde{p}$ except p, after the potential failure detection.*
3. *The set of labels used in the default activity and the set of labels used in the failure handling activity must be distinct.*
4. *The set of protocols spawned in the default activity and the failure handling activity are distinct.*
5. *The sender and the receiver must be distinct for all interaction.*

The extended well-formedness is closely aligned with Def. 24 (Well-Formedness). However, the global type syntax is a bit more restrictive than the extended global type syntax, and condition (1) ensures that a well-formed extended global follows the more restrictive syntax. Further condition (2) is a modification of Def. 24 (Well-Formedness)((1)) which works for global type reduction. In particular, it caters to the case that some participants switched from the normal activity to failure handling.

Extended Global Type Reduction

Figure B.18 contains the extended global type labeled reduction relation ($\mathcal{F}, G_1 \xrightarrow{\alpha} \mathcal{F}, G_1'$). Its labels ranging over: $!\underline{p}l_j \mid ?\underline{p}l_j \mid p@\underline{p'} \mid \tau \mid g(\underline{\bar{p}; R; \overline{R}})$.

[3]Coppo et al. [CDYP16] proposed to model a fired interaction in the global type and the global type reduction. An interesting difference is that we keep the branching options and record the branching choice (l_j).

$$\textsc{WithDef}\ \frac{\mathcal{F}, G_1 \xrightarrow{\alpha} \mathcal{F}, G_1' \qquad \mathsf{sub}(\alpha) \subseteq \widetilde{p}_1}{\mathcal{F}, \{[\widetilde{p}_1]G_1 \ \mathsf{with}\ \_\} \xrightarrow{\alpha} \mathcal{F}, \{[\widetilde{p}_1]G_1' \ \mathsf{with}\ \_\}} \qquad \textsc{Spawn}\ \mathcal{F}, g(\overline{p{:}r}; R; \overline{R}).G \xrightarrow{g(\overline{p}; R; \overline{R})} \mathcal{F}, G$$

$$\textsc{WithHdl}\ \frac{\mathcal{F}, G_1 \xrightarrow{\alpha} \mathcal{F}, G_1' \qquad \mathsf{sub}(\alpha) \subseteq \widetilde{p}_1}{\mathcal{F}, \{\_\ \mathsf{with}_{p@p'}[\widetilde{p}_1]\ G_1\} \xrightarrow{\alpha} \mathcal{F}, \{\_\ \mathsf{with}_{p@p'}[\widetilde{p}_1]\ G_1'\}} \qquad \textsc{Mut}\ \mathcal{F}, \mu\mathtt{t}.G \xrightarrow{\tau} \mathcal{F}, G\{\mu\mathtt{t}.G/\mathtt{t}\}$$

$$\textsc{WithActHdl}\ \frac{\mathcal{F}, G_2 \xrightarrow{\alpha} \mathcal{F}, G_2' \qquad \mathsf{sub}(\alpha) = p'' \qquad p'' \notin \widetilde{p}_2 \qquad p'' \neq p}{\mathcal{F}, \{[\widetilde{p}_1]G_1 \ \mathsf{with}[\widetilde{p}_2]_{p@p'}\ G_2\} \xrightarrow{\alpha} \mathcal{F}, \{[\widetilde{p}_1 \setminus p'']G_1 \ \mathsf{with}[\widetilde{p}_2 \cup p'']_{p@p'}\ G_2'\}}$$

$$\textsc{WithActFHdl}\ \frac{p \in \mathcal{F}}{\mathcal{F}, \{[\widetilde{p}_1]G_1 \ \mathsf{with}[\emptyset]_{p@p'}\ G_2\} \xrightarrow{p@p'} \mathcal{F}, \{[\widetilde{p}_1 \setminus p']G_1 \ \mathsf{with}[p']_{p@p'}\ G_2\}}$$

$$\textsc{Snd}\ \frac{l_j \in \{l_i\}_{i \in I}}{\mathcal{F}, p{:}r \to z\,\{l_i : G_i\}_{i \in I} \xrightarrow{p!l_j z} \mathcal{F}, p{:}r \dashrightarrow^{l_j} z[\emptyset]\{l_i : G_i\}_{i \in I}}$$

$$\textsc{Rcv}\ \frac{p' \in \mathsf{pid}(z) \qquad p' \notin \widetilde{p}_1}{\mathcal{F}, p{:}r \dashrightarrow^{l_j} z[\widetilde{p}_1]\{l_i : G_i\}_{i \in I} \xrightarrow{p?l_j p'} \mathcal{F}, p{:}r \dashrightarrow^{l_j} z[\widetilde{p}_1 \cup p']\{l_i : G_i\}_{i \in I}}$$

$$\textsc{AsyncSpawn}\ \frac{\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G' \qquad \widetilde{p}_r = \{\bar{p}, R_{ids}, \overline{R}_{ids}\} \qquad \mathsf{sub}(\alpha) \cap \widetilde{p}_r = \emptyset}{\mathcal{F}, g(\overline{p{:}r}; R; \overline{R}).G \xrightarrow{\alpha} \mathcal{F}, g(\overline{p{:}r}; R; \overline{R}).G'}$$

$$\textsc{AsyncSnd}\ \frac{\forall i \in I.\ \mathcal{F}, G_i \xrightarrow{\alpha} \mathcal{F}, G_i' \qquad \widetilde{p}_r = \{p, \mathsf{pid}(z)\} \qquad \mathsf{sub}(\alpha) \cap \widetilde{p}_r = \emptyset}{\mathcal{F}, p{:}r \to z\,\{l_i : G_i\}_{i \in I} \xrightarrow{\alpha} \mathcal{F}, p{:}r \to z\,\{l_i : G_i'\}_{i \in I}}$$

$$\textsc{AsyncRcv}\ \frac{\widetilde{p}_r = \mathsf{pid}(z) \setminus \widetilde{p} \qquad \mathcal{F}, G_j \xrightarrow{\alpha} \mathcal{F}, G_j' \qquad \mathsf{sub}(\alpha) \cap \widetilde{p}_r = \emptyset}{\mathcal{F}, p{:}r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i \in I} \xrightarrow{\alpha} \mathcal{F}, p{:}r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i \in I \setminus j} \cup \{l_j : G_j'\}}$$

Figure B.18.: Global type reduction.

We now define the subject of a reduction, i.e. the participant(s) who performed the global type labeled reduction.

Definition 50 (sub$(\alpha) = \widetilde{p}$)**.** *The subject(s) associated with a label are defined as:*

$$sub(!\underline{p}l_j) = p \quad sub(?\underline{p}l_j) = p \quad sub(p@\underline{p'}) = p \quad sub(\tau) = \emptyset$$
$$sub(g(\underline{\bar{p}; R; \overline{R}})) = \bar{p} \cup R_{ids} \cup_{R \in \overline{R}} R_{ids}$$

Lemma 20 ($\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G'$ is deterministic)**.** *Given a well-formed* $\{[\widetilde{p}_1]G_1 \text{ with}[\widetilde{p}_2]_{p@p'} G_2\}$*,* $\mathcal{F}, G_i \xrightarrow{\alpha} \mathcal{F}, G'_i$*,* $\mathcal{F}, G_i \xrightarrow{\alpha} \mathcal{F}, G''_i$*,* $\alpha \neq \tau$ *and* $i \in \{1, 2\}$ *then* $G' = G''$

Proof. Structural induction on $\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G'$. □

Extended Projection

Definition 51 ($b \cdot b'$)**.** $b \cdot b'$ *is the concatenation of all queues in* b *and* b'

Definition 52 (Extended Projection)**.**
The projection of an extended global type G *to an unsuspected participant* p *is defined in Figure B.19.*

The extended projection mimics the behavior of "normal" projection. The key differences are: (a) it uses participant ids and roles instead of only roles, (b) it projects not only to a local type but also to a queue type, (c) it caters for the extended with type ($\{[\widetilde{p}_1]G_1 \text{ with}[\widetilde{p}_2]_{p_f@p_m} G_2\}$) , and (d) it considers partially completed interactions, i.e., interactions where a message(s) was sent but is not yet received. The first point is a minor technicality, and we will now explain (b) - (d) in more detail.

(b): As the original MPST works [HYC08, HYC16], our system applies an asynchronous reduction semantic, and therefore duality needs to consider messages in the queue. E.g., end should be dual to $?l_1$.end if we have l_1 in the queue. The classical MPSTs works "merges" the queue type and the endpoint types before duality (cf. [Conc] in Fig 8 page 30 in [HYC16]). Our work draws inspiration from session remainders [MY15], and instead of "merging" the endpoint type and the queue type, we calculate the effect of the queue type on the local type (see Definition 39 (-)). Therefore, we require the queue types for duality, and as a consequence, extended projection extracts both the local type and the queue type for a participant.

(c): The asynchronous reduction enables cases where some participants perform failure handling, and others are in the normal activity. Therefore, a participant in the normal activity needs to consider potential messages from the failure handling. If a participant has active failure handling, then projection will not produce labels for the default activity,

$$
\{[\widetilde{p}_1]G_1 \text{ with} [\widetilde{p}_2]_{p_f@p_m}\ G_2\} \upharpoonright p ::=
\begin{cases}
(L \text{ with } p_f{:}r_f{\downarrow}.L', b) & \text{if } p \in \widetilde{p}_1 \ \wedge\ p = p_m \ \wedge\ p \notin \widetilde{p}_2 \\
& \quad (L, b) = G_1 \upharpoonright p \quad (L', \epsilon) = G_2 \upharpoonright p \\
(L \text{ with end}, b) & \text{else if } p \in \widetilde{p}_1 \ \wedge\ p = p_f \ \wedge\ p \notin \widetilde{p}_2 \\
& \quad (L, b) = G_1 \upharpoonright p \\
(L \text{ with } L', b \cdot b') & \text{else if } p \in \widetilde{p}_1 \ \wedge\ p \notin \widetilde{p}_2 \\
& \quad (L, b) = G_1 \upharpoonright p \quad (L', b') = G_2 \upharpoonright p \\
(- \text{ with } L', b') & \text{else if } p \in \widetilde{p}_2 \ \wedge\ p \notin \widetilde{p}_1 \\
& \quad (L', b) = G_2 \upharpoonright p
\end{cases}
$$

$$
p_s{:}r_s \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i \in I} \upharpoonright p ::=
\begin{cases}
G_j \upharpoonright p & \text{if } p_s = p \\
(p_s{:}r_s?\{l_i : L_i\}_{i \in I}, [p_s \mapsto l_j] \cdot b_j) & \text{else if } p \in \mathsf{pid}(z) \ \wedge\ p \notin \widetilde{p} \\
& \quad \forall i \in I.(L_i, b_i) = G_i \upharpoonright p \\
G_j \upharpoonright p & \text{else}
\end{cases}
$$

$$
p{:}r \to z\,\{l_i : G_i\}_{i \in I} \upharpoonright p' ::=
\begin{cases}
(z!\{l_i : L_i\}_{i \in I}, b) & \text{if } p = p' \quad \wedge \quad \forall i.(L_i, b) = G_i \upharpoonright p' \\
(p{:}r?\{l_i : L_i\}_{i \in I}, b) & \text{else if } p \in \mathsf{pid}(z) \ \wedge\ \forall j \in I.(L_j, b) = G_j \upharpoonright p \\
G_1 \upharpoonright p & \text{else if } \forall i, j \in I.\ G_i \upharpoonright p' = G_j \upharpoonright p'
\end{cases}
$$

$$
g(\overline{p{:}r}; R; \overline{R}).G \upharpoonright p' ::=
\begin{cases}
(g(\overline{p{:}r}; R; \overline{R}).L, b) & \text{if } (L, b) = G \upharpoonright p' \ \wedge\ (p' \in \overline{p{:}r} \vee p' \in \{R_{ids}, \overline{R}_{ids}\}) \\
G \upharpoonright p' & \text{else}
\end{cases}
$$

$$
\mu\mathtt{t}.G \upharpoonright p' ::= \left\{\ (\mu\mathtt{t}.L, b) \ \text{ if } G \upharpoonright r = (L, b) \ \wedge\ L \neq \mathtt{t}, \quad (\mathsf{end}, \varepsilon) \ \text{ otherwise}\ \right\}
$$

$$
\mathtt{t} \upharpoonright p' ::= (\mathtt{t}, \varepsilon) \qquad \mathsf{end} \upharpoonright p' ::= (\mathsf{end}, \varepsilon)
$$

Figure B.19.: Extended projection.

as clean would remove them without advancing the protocol state (cf. Definition 57 (mirroring), will be defined shortly).

(d): In the projection of a fired interaction, we need to differentiate if a message is in transit or is received. If a message is received, the endpoint type moves forward, e.g., $\mathcal{E}[q{:}r?\{l_i : L_i\}_{i\in I}]$ goes to $\mathcal{E}[L_j]$ (cf. Recv in Figure 4.10).

Definition 53. *The projection of an extended global type to all unsuspected participant, $[[G]]_{\mathcal{F}}$ is defined as:*

$$\{p \mapsto (L, b) \mid G \upharpoonright p = (L, b) \ \wedge \ p \notin \mathcal{F} \ \wedge \ (p \in G \vee (p \in R_{ids} \ \wedge \ R \in G))\}$$

Lemma 21 (Preservation of projection)**.** *Given a well-formed extended global type G with $[[G]]_{\mathcal{F}}$ and $\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G'$ then $[[G']]_{\mathcal{F}}$.*

Proof. Proof by structural induction over $\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G'$. □

Lemma 22 (Preservation of Well-Formedness)**.** *Given a well-formed extended global type G and $\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G'$ then G' is well formed.*

Proof. Proof by structural induction over $\mathcal{F}, G \xrightarrow{\alpha} \mathcal{F}, G'$ □

Fidelity

Before defining fidelity, we define some supporting definitions. Furthermore, we state and prove the main supporting lemma for fidelity (cf. Lemma 23).

Definition 54. *$g{\downarrow}^{\overline{p{:}r}}$ transforms the subprotocol g, with $g_i(\overline{r}; r; \overline{R}) = G$ with $r@r.G \in \mathcal{G}$, into an extended global type using $\overline{p{:}r}$ for the association of role to participant ids.*

Definition 55 (Reachable)**.** *An extended global type G' is reachable in G if and only if after replacing all $r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i\in I}$ with $r \dashrightarrow^{l_j} z[\widetilde{p}]l_j : G_j$ the following is still true: $G' \in G$*

Definition 56 (Outer failure)**.** *A session s has an outer failure if it contains a participant p who is associated with a named role, p is suspected and p is not monitored in s.*

Definition 57 (Mirrors)**.** *Given a well formed extended global type G, typing environments $(\Delta,\ \Sigma,\ \mathcal{F},\ \Theta)$ and a session s with $(\widetilde{p}, \_) \in \Theta(s)$ and no outer failure in s, then G mirrors s is defined as:*

- *if the monitored participant in s is not suspected*

$$\forall p \in \widetilde{p}.\ p \notin \mathcal{F} \implies s[p] : (L, b) \in \Sigma \wedge \\ G \upharpoonright p = (L', b') \ \wedge\ b = b' \ \wedge\ \mathsf{DAct}(L) \equiv_{unfold} \mathsf{DAct}(L') \wedge \\ \mathsf{FAct}(L) \equiv_{unfold} \mathsf{FAct}(L')$$

- *if the monitored participant in s is suspected*

$$\forall p \in \widetilde{p}.\ p \notin \mathcal{F} \implies s[p] : (L, b) \in \Sigma \wedge \\ G \upharpoonright p = (L', b') \ \wedge\ b \equiv_{G\ cln} b' \ \wedge\ \mathsf{FAct}(L) \equiv_{unfold} \mathsf{FAct}(L') \wedge \\ \mathsf{isFHdl}(L) = \mathsf{isFHdl}(L')$$

$L \equiv_{\text{unfold}} L'$ denotes that two local types are equivalent up to unfoldings. Where L and end_L are considered to be identical under $\equiv_{\text{unfold}}$. For example $p_1{:}r?l.\mu\mathtt{t}L \equiv_{\text{unfold}} p_1{:}r?l.L\{\mu\mathtt{t}L/\mathtt{t}\}$. We use equivalent up to unfolding since asynchrony allows endpoints to proceed independently. Therefore, some endpoints have potentially proceeded further into a recursion than others. $\mathsf{DAct}(L)$ returns the type of the normal activity, and $\mathsf{FAct}(L)$ returns the type of the failure handling activity. $\mathsf{isFHdl}(L)$ returns true if L has active failure handling and false otherwise.

Definition 58 ($b_1 \equiv_{G\ \text{cln}} b_2$)**.** *Let $G' = \mathsf{FAct}(G)$. $b_1 \equiv_{G\ cln} b_2$ is defined as:*

$$\forall p.\ b_1(p) = \bar{l}_1 \cdot \bar{l}'_1 \ \wedge\ b_2(p) = \bar{l}_2 \cdot \bar{l}'_2 \ \wedge\ \bar{l}'_1 = \bar{l}'_2 \ \wedge\ \forall l \in \bar{l}_1 \cdot \bar{l}_2.\ l \notin G'$$

Definition 59 (Partial mirroring)**.** *Given a well formed extended global type G, the typing environments $(\Delta,\ \Sigma,\ \mathcal{F},\ \Theta)$ and a session s with $(\widetilde{p}, \_) \in \Theta(s)$, no outer failure in s. Then G partially mirrors s is defined as follows:*

- *if the monitored participant in s is not suspected*

$$\forall p \in \widetilde{p}.\ p \notin \mathcal{F} \ \wedge\ s[p] : (L, b) \in \Sigma \implies \\ G \upharpoonright p = (L', b') \ \wedge\ b = b' \wedge \\ \mathsf{DAct}(L) \equiv_{unfold} \mathsf{DAct}(L') \ \wedge\ \mathsf{FAct}(L) \equiv_{unfold} \mathsf{FAct}(L')$$

- *if the monitored participant in s is suspected*

$$\forall p \in \widetilde{p}.\ p \notin \mathcal{F} \ \wedge\ s[p] : (L, b) \in \Sigma \implies \\ G \upharpoonright p = (L', b') \ \wedge\ b \equiv_{G\ cln} b' \ \wedge\ \mathsf{FAct}(L) \equiv_{unfold} \mathsf{FAct}(L')$$

Lemma 23 (Fidelity main lemma)**.** *Given $(\mathcal{F}, \Theta, N)$, a session s with no outer failure, a well-formed extended global type G and $(\mathcal{F}, \Theta, N) \to (\mathcal{F}', \Theta', N')$ in s, such that $\Gamma, \Theta, \Sigma \vdash N \triangleright \Delta$, G partially mirrors s, $(\Delta, \Sigma, \mathcal{F}, \Theta) \to (\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ and $\Gamma', \Theta', \Sigma' \vdash N' \ \triangleright\ \Delta'$, then:*

- *If an unsuspected participant performs one of the following base reductions: Send, Recv, MSnd, Spawn, Mon or RcvFN and either: the monitored participant is not suspected; or the monitored participant is suspected, and the reduction happens in the failure handling activity, then after the reduction s is partially mirrored by G'' where $\mathcal{F}, G \xrightarrow{\tau}^{*} \mathcal{F}, G'$ and $\mathcal{F}, G' \xrightarrow{\alpha} \mathcal{F}, G''$. Further, α and sub(α) mirrors the reduction in $(\mathcal{F}, \Theta, N) \rightarrow (\mathcal{F}', \Theta', N')$.*
- *If the monitored participant is suspected and the reduction happens in the default activity, then G partially mirrors s after the reduction.*
- *If a suspected participant not associated with a named role performs the reduction, then G partially mirrors s after the reduction.*
- *If the base reduction is Fire, Unfold, or Clean, then G partially mirrors s after the reduction.*
- *If the reduction is Sess-GC with a session s' ($s' \neq s$) being garbage collected, then G partially mirrors s after the reduction.*
- *If the reduction is Susp and the session s has no outer failure after reduction, then G partially mirrors s after the reduction.*

Proof. Induction over $(\mathcal{F}, \Theta, N) \rightarrow (\mathcal{F}', \Theta', N')$.

- Case Fire: No change to any configuration in N, i.e, no change to any protocol state. Therefore, G partially mirrors after reduction.
- Case Send:

  By reduction (before reduction):

$$N = \tag{B.281}$$
$$s[p][r]!l_j.P' \mid\mid \tag{B.282}$$
$$s[p] : (L_p = \mathcal{E}[q{:}r!\{l_i : L_i\}_{i \in I}], b_p) \mid\mid \tag{B.283}$$
$$s[q] : (L_q, b_q) \tag{B.284}$$

  and after reduction:

$$N' = \tag{B.285}$$
$$P' \mid\mid \tag{B.286}$$
$$s[p] : (\mathcal{E}[L_j], b_p) \mid\mid \tag{B.287}$$
$$s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_j]) \tag{B.288}$$

By typing:

$$s[p] : (L_p = \mathcal{E}[q{:}r!\{l_i : L_i\}_{i\in I}], b_p) \in \Sigma \ \wedge \ s[q] : (L_q, b_p) \in \Sigma \tag{B.289}$$

and by assumption (typing) after reduction:

$$s[p] : (\mathcal{E}[L_j], b_p) \in \Sigma' \ \wedge \ s[q] : (L_q, b_q[p \mapsto b_q(p) \cdot l_j]) \in \Sigma' \tag{B.290}$$

Case 1. p or q suspected. By assumption: no outer failure. Therefore, the interaction is in the normal activity. Therefore mirroring before reduction is only on the failure handling activities and only needs to be ensured for failure handling activities after reduction.

Assume p suspected (q suspected similar and simpler)

$$G \upharpoonright q = (L'_q, b'_q) \tag{B.291}$$

$$b'_q \equiv_{G\ \mathrm{cln}} b_q \tag{B.292}$$

$$\mathsf{FAct}(L_q) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_q) \tag{B.293}$$

$$\mathsf{isFHdl}(L) = \mathsf{isFHdl}(L') \tag{B.294}$$

By well-formedness of G and p being the monitored participant (no outer failure)

$$\nexists l \in b'_q(p).\ l \in \mathsf{FAct}(L'_q) \tag{B.295}$$

By Equation (B.292) and Equation (B.295)

$$\nexists l \in b_q(p).\ l \in \mathsf{FAct}(L_q) \tag{B.296}$$

By Equation (B.296), the newly added label can be removed in $\equiv_{G\ \mathrm{cln}}$ after reduction. No mirroring is needed in the normal activity since we have a failure. Therefore, G mirrors $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$.

Case 2. p and q unsuspected, active failure handling, interaction is in default activity: By case p not in failure handling.

$$L_p = q{:}r_q!\{l_i : L_i\}_{i\in I} \text{ with } L''_p \tag{B.297}$$

By mirroring:

$$G \upharpoonright p = (L'_p, b'_p) \tag{B.298}$$
$$b'_p \equiv_{G\ \mathrm{cln}} b_p \tag{B.299}$$
$$\mathsf{FAct}(L_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_p) \tag{B.300}$$
$$\mathsf{isFHdl}(L_p) = \mathsf{isFHdl}(L'_p) \tag{B.301}$$
$$G \upharpoonright q = (L'_q, b'_q) \tag{B.302}$$
$$b'_q \equiv_{G\ \mathrm{cln}} b_q \tag{B.303}$$
$$\mathsf{FAct}(L_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_p) \tag{B.304}$$
$$\mathsf{isFHdl}(L_q) = \mathsf{isFHdl}(L'_q) \tag{B.305}$$

To show: By case mirroring with G is required after reduction. After reduction:

$$s[p] : (\mathcal{E}[L_j], b_p) \in \Sigma' \qquad \mathcal{E}[L_j] = L_j \text{ with } L''_p \tag{B.306}$$

By Equation (B.297), and Equation (B.300):

$$\mathsf{FAct}(L_j \text{ with } L''_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_p) \tag{B.307}$$

By Equation (B.301), Equation (B.303), Definition 49 and projection:

$$\nexists l \in b'_q(p).\ l \in \mathsf{FAct}(L_q) \tag{B.308}$$

By Equation (B.308): $b'_q(p)$ contains no label from the failure activity, further $l_j \notin L''_p$ therefore $\equiv_{G\ \mathrm{cln}}$ discards every message from p before mirroring the queue content.

Case 3. p and q unsuspected and mirroring coves the interaction. Assume no active failure handling, active failure handling is similar. By Assumption

$$G \upharpoonright p = (L'_p, b'_p) \tag{B.309}$$
$$b_p = b'_p \tag{B.310}$$
$$\mathsf{DAct}(L_p) \equiv_{\mathrm{unfold}} \mathsf{DAct}(L'_p) \tag{B.311}$$
$$\mathsf{FAct}(L_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_p) \tag{B.312}$$
$$G \upharpoonright q = (L'_q, b'_q) \tag{B.313}$$
$$b_q = b'_q \tag{B.314}$$
$$\mathsf{DAct}(L_q) \equiv_{\mathrm{unfold}} \mathsf{DAct}(L'_q) \tag{B.315}$$
$$\mathsf{FAct}(L_q) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_q) \tag{B.316}$$

By projection the extended global type contains a reachable type of the form:

$$G_s = p{:}r_p \to q{:}r\{l_i : G_i\}_{i \in I} \tag{B.317}$$

If G_s is the prefix in the default activity, then WithDef followed by Snd applies to G. Extended projection ensures $\mathrm{sub}(l) \subseteq \widetilde{p}_1$ in WithDef. When G_s is not a prefix: We show that G_s can do a step (after potential unfolds, $\mathcal{F}, G \xrightarrow{\tau}{}^{*} \mathcal{F}, G'$) via WithDef followed by an arbitrary number of Async* followed by an Snd by contradiction. G_s cannot do a step if G_b is a prefix of G_s and G_b is blocking. Case distinction over G_b:

- G_b is an interaction and AsyncSnd does not apply:

  By projection, p is neither the sender nor the receive of that interaction. Furthermore, the receiver cannot be the role set of p. As p is not the sender or receiver, by projection, the behavior of p must be the same in all branches. Therefore, AsyncSnd applies. Contradiction.
- G_b is a blocking fired interaction, i.e. AsyncRcv does not apply: By projection either: p is not the receiver (neither named nor as a role set); or if p is the receiver and p already received the message (p is contained in the received set). Therefore, AsyncRcv applies. Contradiction.
- G_b is a blocking spawn, i.e. AsyncSpawn does not apply: By projection p cannot be part of that spawn. Contradiction.

Let G' be the extended global after unfolding as shown above the reduction $\mathcal{F}, G' \xrightarrow{!\underline{p}l_j} \mathcal{F}, G''$ is applicable, with the effect of "moving" G_s to $G'_s = p{:}r \dashrightarrow^{l_j} q{:}r_q[\emptyset]\{l_i : G_i\}_{i \in I}$. The change of G_s to G'_s mirrors the changes in the types of p and q, i.e., advancing the send type of p to the continuation L_j and adding the label l_j to the queue of q.

- Case Recv.

  By case before reduction:

$$N = s[p][r_q]?l_j.P' \mid\mid s[p] : (\mathcal{E}[q{:}r_q?\{l_i : L_i\}_{i \in I}], b_p) \tag{B.318}$$

$$b_p(q) = l_j \cdot \bar{l} \tag{B.319}$$

  and after reduction:

$$N' = P' \mid\mid s[p] : (\mathcal{E}[L_j], b'_p) \tag{B.320}$$

$$b'_p = b_p[q \mapsto \bar{l}] \tag{B.321}$$

By assumption of typing (both before and after reduction):

$$s[p] : (\mathcal{E}[q{:}r_q?\{l_i : L_i\}_{i\in I}], b_p) \in \Sigma \tag{B.322}$$
$$s[p] : (\mathcal{E}[L_j], b'_p) \in \Sigma' \tag{B.323}$$

Case distinct over failure of p

Case 1. p is suspected: Case distinction on if p plays a named role or not

Case 1.1. p plays a named role: By Case *Case 1.* and Case *Case 1.1.*: p is in the default activity. Further mirroring is only required for unsuspected participants and the reduction Recv does not effect the type of other participants. Therefore, $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ is still mirrored by G.

Case 1.2. p is only part of a role set: Assume failure handling is active and p is in the failure handling activity[4]. The other cases (failure handling is not active or failure handling is active and p is in the default activity) are similar.

Mirroring is only required by unsuspected participant and the reduction Recv does not effect the type of other participants. Therefore, $(\Delta',\ \Sigma',\ \mathcal{F}',\ \Theta')$ is still mirrored by G.

Case 2. p is unsuspected:

Case 2.1. Monitored participant not suspected: By Assumption

$$G \upharpoonright p = (L'_p, b'_p) \tag{B.324}$$
$$\mathsf{DAct}(L'_p) \equiv_{\text{unfold}} \mathsf{DAct}(L_p)(= q{:}r_q?\{l_i : L_i\}_{i\in I}) \tag{B.325}$$
$$\mathsf{FAct}(L'_p) \equiv_{\text{unfold}} \mathsf{FAct}(L_p) \tag{B.326}$$
$$b'_p(q) = l_j \cdot \bar{l} \tag{B.327}$$

By projection the extended global type contains a reachable type of the form:

$$G_s = q{:}r_q \dashrightarrow^{l_j} p{:}r\{l_i : G_i\}_{i\in I} \text{ or } G_s = q{:}r_q \dashrightarrow^{l_j} R\{l_i : G_i\}_{i\in I} \tag{B.328}$$

the rest of the proof is similar to the Send case (cf. *Case 3.*).

Case 2.2. Monitored participant suspected: If the receive is covered by partial mirroring, i.e., occurs in the failure handling case is similar to *Case 2.1.*. If on the other hand the receive is not covered, i.e. occurs in the normal activity then:

[4]In this case p is suspected but does not play a named role (only named role can be monitored) and p's failure is not an outer failure.

$$G \upharpoonright p = (L'_p, b'_p) \tag{B.329}$$
$$\mathsf{FAct}(L'_p) \equiv_{\text{unfold}} \mathsf{FAct}(L_p) \tag{B.330}$$
$$\mathsf{isFHdl}(L_p) = \mathsf{isFHdl}(L'_p) \tag{B.331}$$
$$b'_p \equiv_{G\ \text{cln}} b_p \tag{B.332}$$

By Equation (B.332):

$$b_p(q) = \bar{l}_1 \cdot \bar{l}'_1 \ \wedge\ b'_p(q) = \bar{l}_2 \cdot \bar{l}'_2 \ \wedge\ \bar{l}'_1 = \bar{l}'_2 \ \wedge\ \forall l \in \bar{l}_1 \cdot \bar{l}_2.\ l \notin G' \tag{B.333}$$

By case:

$$l_j \in \bar{l}_1 \tag{B.334}$$

Therefore, N' is partially mirrored by G after reduction.

- Case MSnd. Argumentation is similar to Send.
- Case Unfold. Mirroring is up to unfolds.
- Case Spawn. By assumption and projection, spawn is present in the global type for all participants involved. Async* applies by the same argument as in the Send case (the argument needs to be repeated for every participant involved in the spawn). The rule Spawn has no precondition, i.e., step follows trivially.
- Case Sess-GC.

  The rule Sess-GC has no effect on the configuration types in s.
- Case Mon.

  By case:

$$N = s[p][r]{\downarrow}.P' \mid\mid \tag{B.335}$$
$$s[p] : (L_p = L' \text{ with } q{:}r{\downarrow}.L, b_p) \tag{B.336}$$
$$\mid\mid_{s' \in \widetilde{s}}\ s'[p] : (L_{s'}, b_{s'}) \tag{B.337}$$

and after reduction:

$$N' = P \mid\mid \tag{B.338}$$
$$s[p] : (- \text{ with } L, b_p) \tag{B.339}$$
$$\mid\mid_{s' \in \widetilde{s}}\ s'[p] : (\mathsf{end}_{L_{s'}}, b_{s'}) \tag{B.340}$$

By typing, the configurations types are present in Σ and Σ'.

By case, q suspected and we have:

$$G \upharpoonright p = (L'_p, b'_p) \tag{B.341}$$
$$L'_p = L''_p \text{ with } q{:}r{\downarrow}.L'''_p \tag{B.342}$$
$$b'_p \equiv_{G\ \mathrm{cln}} b_p \tag{B.343}$$
$$\mathsf{FAct}(L_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L'_p) \tag{B.344}$$
$$\mathsf{isFHdl}(L) = \mathsf{isFHdl}(L') \tag{B.345}$$

By projection and well-formedness, G must have the form:

$$\{[\widetilde{p}_1]G_1 \text{ with}[\emptyset]_{q:r@p:r_p}\ G_2\} \tag{B.346}$$

By Equation (B.346) and case, WithActFHdl applies:

$$\{[\widetilde{p}_1 \setminus p]G_1 \text{ with}[p]_{q:r@p:r_p}\ G_2\} \tag{B.347}$$

Projection on to Equation (B.347)

$$- \text{ with } L'''_p \tag{B.348}$$

By Equation (B.342):

$$\mathsf{FAct}(- \text{ with } L'''_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(- \text{ with } L) \tag{B.349}$$

For all participants (in s), excluding p, projection of Equation (B.346) and Equation (B.347) result in the same local types and same queue types.

- Case RcvFN. Similar to the cases Mon and Recv.
- Case Susp. W.o.l.g p was removed. Case distinction over p's involvement in s.

  Case 1. p not in s. Nothing to show.

  Case 2. p does not play a named role. After the reduction partial mirroring needs to hold for one less participant in the same way as before the reduction.

  Case 3. p plays a named role other then the monitored role. Nothing to show since fidelity does not apply to s after the reduction.

  Case 4. p plays the monitored role. After the reduction the second case, instead of the first case, of partial mirroring must apply. $b \equiv_{G\ \mathrm{cln}} b'$ required less then $b = b'$ and $\mathsf{DAct}(L) \equiv_{\mathrm{unfold}} \mathsf{DAct}(L') \wedge \mathsf{FAct}(L) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L')$ implies $\mathsf{isFHdl}(L) = \mathsf{isFHdl}(L')$. Therefore, G mirrors s after the reduction.

- Case Clean.

  By assumption:

$$N = (\overline{H}, p) \mid\mid s[p] : (L, b) \tag{B.350}$$
$$b(q) = l_j \cdot \bar{l} \tag{B.351}$$
$$(L = - \text{ with } L' \ \wedge \ l \notin L') \tag{B.352}$$
$$\vee$$
$$(L = L' \text{ with } L'' \ \wedge \ l_j \in L' \ \wedge \ \exists l' \in \bar{l}.\ l' \in L'')$$

  and after reduction:

$$(\overline{H}, p) \mid\mid s[p] : (L, b') \text{ where } b' = b[q \mapsto \bar{l}] \tag{B.353}$$

  By typing

$$s[p] : (L, b) \in \Sigma \quad s[p] : (L, b') \in \Sigma' \tag{B.354}$$

  By projection and mirroring before reduction: the monitored participant has failed and failure handling is active (both at the processes level and in the global type). Therefore, by partial mirroring:

$$G \upharpoonright q = (L'_q, b'_q) \tag{B.355}$$
$$b'_q \equiv_{G \text{ cln}} b_q \tag{B.356}$$
$$\mathsf{FAct}(L_q) \equiv_{\text{unfold}} \mathsf{FAct}(L'_q) \tag{B.357}$$
$$\mathsf{isFHdl}(L) = \mathsf{isFHdl}(L') \tag{B.358}$$

  By Equation (B.352) and Equation (B.357) and Equation (B.354): G partially mirrors N'

- Case Par: By reduction

$$(\Theta_1, \mathcal{F}_1, N_1 \mid\mid N_2) \to (\Theta'_1, \mathcal{F}'_1, N'_1 \mid\mid N_2) \tag{B.359}$$

  with

$$(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta'_1, \mathcal{F}'_1, N'_1) \tag{B.360}$$

  By induction hypothesis, Lemma 23 holds for Equation (B.360). If we have one of the cases where G also mirrors N'_1, i.e., no reduction of G is needed, then we have nothing to show.

  Assume G'' mirrors N'_1 with $\mathcal{F}, G \xrightarrow{\tau}{}^{*} \mathcal{F}, G'$ and $\mathcal{F}, G' \xrightarrow{\alpha} \mathcal{F}, G''$. Case distinction over α. The cases are similar. Therefore, we only show a sub-set of the cases.

Case 1. $\alpha = !\underline{p}l'$**:** By case and induction hypothesis: the base reduction in Equation (B.359) was SEND. By typing, the configurations of the sender and receiver are present in N_1/N_1'. By extended global type reduction (cf. SND) and projection: the projection of G and G'' results in the same local types (up to unfolds) and queue types for participants that are neither the sender nor receiver.

Case 2. $\alpha = g(\underline{\bar{p}; R; \overline{R}})$**:** By case and induction hypothesis: the base reduction in Equation (B.359) was SPAWN. Therefore, at least the participants involved in the spawn are in N_1 and N_1'. There can be suspected participants associated with a role set involved in the spawn in N_2. However they are ignored in partial mirroring. Rest is similar to the previous case.

- Cases RES and STRR: similar to PAR.

□

Theorem 7 (Fidelity)**.** *Assume* $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1)$*, with* $\{s_i\}_{i \in I}$ *as the sessions in* Θ_1*, and well-formed extended global types* $\{G_i\}_{i \in I}$ *such that* $\forall i \in I$ *either* G_i *mirrors* s_i *or* s_i *has an* outer failure*. Let* $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1) \to (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$*, with* $\{s_j'\}_{j \in J}$ *as the sessions in* Θ_2*. Then there exist well-formed* $\{G_j'\}_{j \in J}$ *such that* $\forall j \in J$ *either (a)* s_j' *has an* outer failure *or (b)* G_j' *mirrors* s_j'*, and if* $s_j' = s_i$*,* $i \in I$*, then either (i)* $G_j' = G_i$*, or (ii)* $\mathcal{F}_1, G_i \to^{\tau^*} \to^{\alpha} \mathcal{F}_2, G_j'$*. (*$\to^{\tau^*}$*means recursive unfoldings.)*

Proof. By Theorem 3 (Subject Reduction) we have:

$$\vdash (\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\ N_2) \tag{B.361}$$

By Lemma 15 (Inversion)

$$s : \mathcal{G}, \Sigma_2 \vdash N_2 \ \triangleright\ \Delta_2 \text{ and } (\Delta_2,\ \Sigma_2,\ \mathcal{F}_2,\ \Theta_2) \text{ coherent} \tag{B.362}$$

Let s_i be the session that "performed" the reduction. (We cover SUSP and rules that effect multiple sessions in more detail later):

Case 1. s_i **has no outer failure**: Therefore, by assumption s_i is mirrored by G_i. By Equation (B.362) Lemma 23 applies to the the reduction:

$$(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta_2, \mathcal{F}_2, N_2) \tag{B.363}$$

$N_{1/2}$ are typed in a coherent environment, i.e.,the endpoint configuration exists for all participant that are part of a session. I.e., we get (full) mirroring from partial mirroring which we get from Lemma 23.

Case 2. s_i **has an outer failure**: No mirroring after reduction required.

Let s_j be a session in Θ that did not "perform" the reduction. s_j is only effected by a reduction in s_i if the reduction used one of the following rules: Sess-GC, Mon, or RcvFN. If the reduction used any other rule in s_i the endpoint configuration in s_j do not change in $(\Theta_1, \mathcal{F}_1, N_1) \to (\Theta_2, \mathcal{F}_2, N_2)$ and therefore s_j is still mirrored or still contains an outer failure.

Case distinction over the "special" reduction in s_i. If (a) s_j was removed by Sess-GC then $s_j \notin \Theta_2$, i.e. nothing to show. If (b) a local type of a configuration in s_j was set to stopped by either Mon or RcvFN then mirroring is not effect, since L and end_L are consider to be identical in $\equiv_{\text{unfold}}$.

Let s' be a session in Θ_2 but not in Θ_1. That implies the Spawn reduction was used. By definition of Spawn s' is fresh and only contains unsuspected participant. By typing a well formed global type is used for spawn and therefore the extended version of that global type mirrors s'.

Lastly we cover the reduction Susp. The Susp reduction is not associated with any session. Let p be the newly suspected participant. Case distinction over p's involvement in a session: (a) p is not part of a session then setting p to suspected does not influence mirroring in that session; (b) p is part of a session but not as a named role, i.e., only in a role set. Then there is one less participant for which mirroring is required but no other change; (c) p plays a named role but is not monitored, then we have an outer failure and no mirroring is required; (d) p plays a named role and is monitored. Then mirroring change to the case of monitored participant is suspected. $b \equiv_{G\ \text{cln}} b'$ is less restrictive then $b = b'$ and by well-formedness of G all participant are in the default activity therefore $\mathsf{isFHdl}(L) = \mathsf{isFHdl}(L')$ is given.

□

B.6.5. Property: progress

Lemma 24 (Progress active handler)**.** *Given* $\vdash (\Theta_1, \mathcal{F}_1, (\nu(s : \mathcal{G}))\ N_1)$ *then all active event handlers will reduce to event handling loops.*

Proof. By global assumption initially no handling is active. By typing (in particular THandler) all handler are well-formed, i.e., the handler contains at most one blocking action (spawn, receiver or $\notin$) and if a blocking action exists it is the first action. Furthermore, a handler contains no recursion and no with statement. In addition, the reduction rule Fire does not activate any handler which contains a spawn.

If FIRE actives a handler which contains a receive or a $\notin$ then $\mathsf{fire}(L, L', b, \mathcal{F})$ ensures that a matching message respectively a matching suspicion is present at time of activation. The only reduction beside RECV that can remove messages is CLEAN. CLEAN only applies to event loops with no active handler, i.e., it is not applicable since we have an active handler. There is no rule that sets a suspected participant to unsuspected. Therefore, there cannot be a blocking based on the input.

Furthermore, there cannot be a blocking because of a non-matching between the local type in the configuration and the action performed in the handler by definition of $L \asymp L'$. The rule MON and RCVFN, which can change the local type across session boundaries cannot interfere as typing ensure that an endpoint can have at most one active handling.

Lastly SPAWN together with Definition 44 (Coherence) ensures that the required endpoint configuration for the reduction of the handler exists, e.g. the configuration of the receiver in a send reduction. □

Definition 60 (Stopped Participant)**.** *A session s contains a stopped participant if the protocol type in configuration of a non-failed participant has the shape:* end_L.

Proposition 2 (Event loops provide full coverage)**.** *Given $\vdash (\Theta, \mathcal{F}, (\nu(s : \mathcal{G}))\ N)$, an unsuspected p, a configuration $s[p] : (L, b) \in N$ which is not stopped and $(\overline{H}, p) \in N$ with no active event handler, then it exist $[L']\lambda x.\, P \in \overline{H}$ with $L \asymp L'$*

Definition 61 (Session is end / Session is terminated)**.** *A session s is end/is terminated if all configuration of unsuspected participants in s either (1) reached end in the normal activity and the monitored participant is unsuspected or (2) reached end in the failure handling activity.*

Theorem 8 (Subsession Progress)**.** *Assume an initial system $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1)$ and $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1) \rightarrow^* (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$. Let s' in Θ_2 such that s' has* no outer failure*,* no stopped participant*, and is* not terminated*. Then $(\Theta_2, \mathcal{F}_2, N_2') \rightarrow (\Theta_3, \mathcal{F}_3, N_3)$ via a reduction in session s', with either $N_2' = N_2$ or $(\Theta_2, \mathcal{F}_2, N_2) \rightarrow^* (\Theta_2, \mathcal{F}_2, N_2')$.*

Proof. By assumption Θ_1 contains only one session s which is mirrored by the main protocol in $\mathcal{G}$. By Theorem 7 (Fidelity) on each step from $(\Theta_1, \mathcal{F}_1, (\nu(s : \mathcal{G}))\ N_1)$ to $(\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\, N_2)$ there is an extended G for s' (once s' exists). Select the prefix action (after potential unfoldings) in the default activity respectively the prefix in the failure handling activity if failure handling is active. In the prefix selection we potential skip over completed interaction, i.e., $p{:}r \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i \in I}$ with $\widetilde{p} \supseteq \mathsf{pid}(z) \setminus \mathcal{F}$. A prefix exist by assumption *session not end*. In the following we show that the prefix action in G enables

progress in $(\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\, N_2)$, in particular the subject in the prefix is able to perform a step.

Let p be the subject of the prefix in G with the event loop $(\overline{H}, p)$. Let either $N_2' = N_2$ if p has no active handler in N_2, or alternative we get by Lemma 24:

$$(\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\, N_2) \rightarrow^* (\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\, N_2')$$

and p has no active handler in N_2'. If the reduced active handler was associated with s' we are done. By Theorem 3 (Subject Reduction)

$$\vdash (\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\, N_2') \tag{B.364}$$

with $\Gamma_3, \Sigma_3 \vdash N_2' \triangleright \Delta_3$

Case distinction over the prefix of G with p unsuspected and a separate case for p is suspected. Note: Spawn behavior is a bit different compared to other constructs as it has more then one subject

- *Case prefix:* $p{:}r \rightarrow z\{l_i : G_i\}_{i \in I}$ The argumentation is similar for default and failure handling activity.

  By fidelity with extended projection

  $$s'[p] : (L = \mathcal{E}[q{:}r!\{l_i : ...\}_{i \in I}], \_) \in \Sigma \tag{B.365}$$

  or

  $$s'[p] : (L = \mathcal{E}[R!\{l_i : ...\}_{i \in I}], \_) \in \Sigma \tag{B.366}$$

  By assumption p has no active event handler.

  By the coverage check in TELoop it exist

  $$[L']\lambda x.\, P_h \in \overline{H} \text{ with } L \asymp L' \tag{B.367}$$

  Definition 28 (Fire) is true for Equation (B.367) and Fire can be followed up by Send. Done.

- *Case p playing a named role and p is suspected, i.e. s' contains a handled failure but failure handling is not yet active.*

  By assumption (no outer failure) this session handles the failure of p, i.e. we have:

  $$\{[\widetilde{p}]G \text{ with} [\emptyset]_{p:r@q:r_q}\, G\} \tag{B.368}$$

Wolg q has no active event handler (Lemma 24 ensures that an active handler of q could reduce)

$$(\overline{H}_q, q) \tag{B.369}$$

By Theorem 7 (Fidelity) we have

$$s'[p] : (L_q = L_q \text{ with } p{\downarrow}.L'_q, \_) \in N'_2 \tag{B.370}$$

By typing (coverage check)

$$\exists [L'] \lambda x.\, P_h \in \overline{H} \text{ with } L \asymp L' \quad \text{where } L' = r{\downarrow}.L'' \tag{B.371}$$

By case assumption

$$p \in \mathcal{F} \tag{B.372}$$

By Equation (B.371), Equation (B.372) and Definition 28 (Fire) we can activate a failure handler. Further Lemma 24 (Progress Handler) ensures progress for that handler.

- *Case prefix:* $q{:}r_q \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i \in I}$ with $p \in \mathsf{pid}(z) \setminus \widetilde{p}$. There are 3 cases of interest: (1) prefix is in the default activity, (2) prefix in failure handling activity, and (3) prefix in failure handing activity and clean applies for p . We cover the last case (a clean is required before receive) as the other two cases are simpler.

  By case and selection (see above) [ignoring potential $p' \dashrightarrow^{l_k} z'[\widetilde{p}]\{l_j : G_j\}_{j \in J}$ with $\widetilde{p} \supseteq \mathsf{pid}(z')$]:

$$G = \{[\widetilde{p}_1]G' \text{ with} [\widetilde{p}_2]_{p':r_{p'}@p'':r_{p''}}\ q{:}r_q \dashrightarrow^{l_j} z[\widetilde{p}]\{l_i : G_i\}_{i \in I}\} \tag{B.373}$$

  We have

$$p \in \mathsf{pid}(z) \setminus \widetilde{p} \tag{B.374}$$

  Case distinct if $p \in \widetilde{p}_1$ or if $p \in \widetilde{p}_2$ (i.e. if p is in the default activity respectively in the failure handling activity)

  Case $p \in \widetilde{p}_1$: The configuration of p and its type is:

$$s'[p] : (L_p, b_p) \in N'_2 \quad s'[p] : (L_p, b_p) \in \Sigma'_2 \tag{B.375}$$

By Theorem 7 (Fidelity) and projection for Equation (B.375)

$$G \upharpoonright p = (L', b') \tag{B.376}$$
$$b_p \equiv_{G\ \mathrm{cln}} b' \tag{B.377}$$
$$\mathsf{FAct}(L_p) \equiv_{\mathrm{unfold}} \mathsf{FAct}(L') \tag{B.378}$$
$$\mathsf{isFHdl}(L_p) = \mathsf{isFHdl}(L') \tag{B.379}$$

By Equation (B.378) and Equation (B.379)

$$L_p = L' \text{ with } q{:}r_q?\{l_i : L_i\}_{i \in I} \tag{B.380}$$

By case "clean"

$$b_p(q) = l' \cdot \bar{l} \quad l' \notin q{:}r_q?\{l_i : L_i\}_{i \in I} \tag{B.381}$$

By Equation (B.373) and Equation (B.377)

$$\exists l'' \in b_p(q).\ l'' = l_j \tag{B.382}$$

By Equation (B.382) and Equation (B.381) Clean applies. After Clean reduction:

$$s'[p] : (L' \text{ with } q{:}r_q?\{l_i : L_i\}_{i \in I}, b'_p) \in P \tag{B.383}$$
$$b'_p(q) = \bar{l} \tag{B.384}$$
$$\bar{l} = l_1 \cdot \bar{l}' \tag{B.385}$$

Wolg $l_1 \notin L'$ (if $l_1 \notin L'$ then Clean applies with reasoning above). By Theorem 7 (Fidelity) and projection

$$l_1 = l_j \tag{B.386}$$

By Theorem 3 (Subject Reduction) and well typed (coverage check)

$$[L']\lambda x.\ P_h \in \overline{H} \text{ with } L \asymp L' \tag{B.387}$$

can activate handler for failure handling (note the coverage ensure handler for both default and failure handling)

By Equation (B.386) the Definition 28 (Fire) is true together with Equation (B.387) Fire applies.

Case $p \in \widetilde{p}_2$:

By Theorem 7 (Fidelity) and projection

$$s'[p] : (L = - \text{ with } q{:}r_q?\{l_i : L_i\}_{i \in I}, b) \in N_2' \tag{B.388}$$

$$b(q) = l' \cdot \bar{l} \tag{B.389}$$

By case "clean"

$$l' \notin q{:}r_q?\{l_i : L_i\}_{i \in I} \tag{B.390}$$

By Equation (B.388), Equation (B.390) Clean applies. After Clean reduction:

$$s'[p] : (L = - \text{ with } q{:}r_q?\{l_i : L_i\}_{i \in I}, b') \in N_2' \tag{B.391}$$

$$b'(q) = \bar{l} = l_1 \cdot \bar{l}' \tag{B.392}$$

The remaining argumentation follows the case of $p \in \widetilde{p}_1$.

- *Case prefix:* $g(\bar{p}; R; \overline{R}).G$. By case no participant in $\bar{p}$ is suspected (see case above for suspicion in $\bar{p}$). Let $\widetilde{p}_1 = \mathsf{pid}(g(\bar{p}; R; \overline{R})) \setminus \mathcal{F}$. Wolg all q in $\widetilde{p}_1$ have no active event handling (Lemma 24 (Progress Handler) ensure progress for any active handler). By typing, Theorem 7 (Fidelity), and projection

$$\forall q \in \widetilde{p}_1.\ s'[q] : (L_q = \mathcal{E}_q[g(\overline{p{:}R}; R; \overline{R})], b_q) \in N_2' \tag{B.393}$$

By all participant have no active handler, Equation (B.393), assumption there *are enough participant to pick* and typing (ensures that g is projectable) Spawn applies.

□

Theorem 5 (Global Progress)**.** *Assume an initial system* $\vdash (\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1)$ *and a reduction* $(\Theta_1, \mathcal{F}_1, (\nu s : \mathcal{G})\ N_1) \rightarrow^* (\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2)$*. Then either* Θ_2 *is empty, or without using* Susp *we have* $(\Theta_2, \mathcal{F}_2, (\nu s : \mathcal{G})\ N_2) \rightarrow (\Theta_3, \mathcal{F}_3, (\nu s : \mathcal{G})\ N_3)$*.*

Proof. Proof by contradiction, we assume no progress for $(\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\ N_2)$.

Setup: N_2 contains no event loop with an active handler (otherwise Lemma 24 (Progress Handler) applies and we have a contradiction). All session in $(\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\ N_2)$ have (a) an outer failures and/or (b) a stopped participant and/or (c) the session is end (otherwise Theorem 8 (Session Progress) provides progress; by assumption we have enough participant to pick)

First we show that a leaf session which has (a) an outer failure is also (b) stopped (or we could do a step, i.e. contradiction). Secondly we show a contradiction for a stopped leaf session without progress. If a leaf *session reached end* then by Definition 44 (Coherence) $((\Theta_2, \mathcal{F}_2, (\nu(s : \mathcal{G}))\ N_2)$ is well type) immediately either Sess-GC or Root-GC applies.

We say: a session a *covers* a session b if b is spawned in the *default activity* of a. The *top most session* is either the root session or the session closes to the root session.

Leaf session has an outer failure Let s be a leaf session which has an *an outer failure*. Let s' be (a) the top most session which has an active or activatable failure handling which (b) (transitively) covers s. By well-formedness and Spawn all participant in s also occur in s'.

s' has *no outer failure* (would contradiction (a) or the root participant failed which by assumption does not fail). Further s' is *not stopped* (would contradiction (a) only failure activation triggers stop). By assumption no progress and therefore s' *is end*. Furthermore by (a) and no progress in s' is end in failure handling and therefore s must be stopped (failure handling sets all transitively child session to stop and a stop configuration cannot spawn new sessions).

Leaf session contains stopped participant Let s be a leaf session with a *stopped participant*. Let s' be (a) the top most session which has an active failure handling which (b) (transitively) covers s. By well-formedness and Spawn all participant in s also occur in s'. s' must be an *end session* in failure handing or it would have progress. Therefore all participant in s *are stopped*. Sess-GC applies and therefore we have a contradiction.

□

B.7. Global Type for the Full Version of Session-CM

Figure B.20 provides the full global type the full Session-CM. We provide a high-level explanation here. Section 4.2 explains the scheduling process of Spark applications and a simplified global type in more detail.

The protocol contains the subprotocols `Main`, `PDriver`, `PExSchedule`, and `PExecutor`. The `Main` subprotocol is the entry point into the Session-CM protocol and describes the scheduling process of multiple Spark applications. It spawns `PDriver` protocols. The `PDriver` subprotocol is responsible for scheduling *one* Spark application. It assigns a worker to run the driver of that Spark application and spawns `PExSchedule`. The `PExSchedule` subprotocol is responsible to assign executors to a Spark application. For each executor, it

```
Main (roles zk : ZK; assign m : M; rSets M, W) = {
  mu t.
   m → zk : {
     NewDriver(appID: Int) :
       m → W : PrepSpawn().
       spawn PDriver(m; W; W). t,
     DriverDone(appID : Int) : m → W : BMsg(). t,
     EndCM() : m → W : Terminate(). 0}
 with m@zk. zk → M : FailMtoM(). zk → W : FailMtoW(). 0
}
PDriver(roles m : M; assign w : W; rSets W) = {
  m → w : LaunchDriver(appID : Int driver : SPARK.LaunchDriver).
  w → m : AckNStatus(appID : Int).
  spawn PExSchedule(m,w;W ;W).
  w → m : DriverStateChange(status : SPARK.DriverState).0
 with w@m. m → W : {
  FailDriverSpawn(appId : Int newAppID : Int) :
   spawn PDriver(m; W; W).0,
  FailDriverEnd(appId : Int ) : 0}
}
PExSchedule(roles m : M, w : W; assign tw : W; rSets W) = {
  mu t.
   m → W : {
     StartExCase(appID : Int) : spawn PExecutor(m, w; W; W).t,
     End() : 0}
 with tw@m. m → W : {
  FailExScheduleSpawn(appId : Int):
   spawn PExSchedule(m,w; W; W).0,
  FailExScheduleEnd(appId : Int) : 0}
}
PExecutor(roles m : M, w : W; assign wEx : W; rSets W) = {
  m → wEx : StartEx( appId : Int, exId : Int,
                     launchEx : SPARK.LaunchExecutor).
  wEx → m : ExStarted(appId : Int, exId : Int).
  m → w : ExRunning(appId : Int, exId : Int).
  wEx → m : ExDone(appId : Int, exId : Int).
  m → w : ExFinishStatus(appId : Int, exId : Int).0
 with wEx@m. m → W : {
  ExFailSpawn(appId : Int, exId : Int) :
   spawn PExecutor(m,w;W;W).0,
  ExFailEnd(appId : Int, exId : Int) : 0}
};
```

Figure B.20.: Full global type of the Session-CM

spawns `PExecutor`. The `PExecutor` subprotocol handless the lifecycle of an executor for that it assigns a worker to run an executor and monitors that worker.

The four subprotocols will approximately lead to the following subsession tree during execution (ignoring failures): the root session follows `Main` and handles the scheduling of multiple Spark applications. For every Spark application that is currently scheduled, the root session has a child session that follows the `PDriver` subprotocol, i.e., the child session handles the scheduling for *one* Spark application. Each session following `PDriver` has one subsession following `PExSchedule`, which is responsible for assigning executors to the driver. The session following `PExSchedule` has a subsession `PExecutor` for each executor assigned to the Spark applications.

We will now explain the four subprotocols in more detail.

The `Main` subprotocol is the entry point into the Session-CM protocol and root subprotocol. It defines that a master role `m` from the master role set `M` is assigned, and the Zookeeper role `zk` monitors the newly assigned role `m`. In the default activity, it defines a recursive behavior. In it the role `m` has a choice between three options on how the protocol continues. In the first option, `m` selects a Spark application for scheduling and informs the Zookeeper role `zk`, via `NewDriver(appID: Int)`, and all workers, via `PrepSpawn()`. After that, all participants spawn `PDriver`, and recurse back to the beginning. In the second option, `m` informs the participants of the completion of a Spark application, by sending `DriverDone(appID : Int)` to `zk` and `BMsg()` to `W`. Then all participants recurse back. In the third option, `m` signals the termination of the protocol by sending `EndCM()` and `Terminate()` to `zk`, respectively `W`. In the failure handling activity, the Zookeeper role `zk` informs all workers `W` and all standby masters `M` about the failure of `m` and all participants will end the protocol.

The `PDriver` protocol is responsible for scheduling *one* Spark application. It defines that a worker is assigned to play `w` that worker `w` will launch the driver of the Spark application, which gets scheduled. In the default activity, the master role `m` sends to the newly assigned worker role `w` the required information to launch a Spark driver (`LaunchDriver(appID : Int driver : SPARK.LaunchDriver)`). We use `SPARK` as a shorthand for a fully qualified package name. The worker will launch the driver and acknowledge the launch via `AckNStatus(appID : Int)`. After that, all participants spawn the `PExSchedule` subprotocol, which is responsible for assigning executors to the driver. Lastly, the worker `w` informs the master `m` once the Spark application has finished, via `DriverStateChange(status : SPARK.DriverState)`. In the failure handling, activity the master role `m` has two options, either (a) it restarts this protocol or (b) it safely terminates the protocol. In the first option (a), the master informs all workers, via `FailDriverSpawn`, that `m` intends to restart the protocol. After that, the master and all workers spawn `PDriver`, i.e., restart the failed protocol. In the second option (b), the master informs all workers to

stop the protocol via `FailDriverEnd`.

The `PExSchedule` protocol is responsible for assigning executors to a Spark application. It assignes a worker to play `tw`, `tw` performs no specific behavior – it is a dummy assignment–, i.e., the assignment is performed since every spawn involves a assignment. The default activity defines a recursive behavior. In it, the master `m` has two options. Their first option is to signal all workers `W`, via `StartExCase(appID : Int)`, that they will together spawn `PExecutor`. After that, they recurse back, i.e., the first option assigns an additional executor to a Spark application. Their second option is to signal, via `End()`, the end of this protocol. The second option, is taken once sufficient executors are assigned to the Spark application. The failure handling, is similar to the failure handling of `PDriver`. The master `m` informing all workers about the failure and is in control to either restart the subprotocol or to terminate the subprotocol.

The `PExecutor` protocol is responsible for the lifecycle of an executor. It assignes a worker to play `wEx`, who will launch an executor. In the default block, the master role `m` sends the required information to launch the executor to `wEx`, via `StartEx(...)`. Then `wEx` launches the executor and informs the master, via `ExStarted(appId : Int, exId : Int)`. The master `m` informs the worker `w`, which runs the driver associated with the spawn executor, about the executor launch. Lastly, `wEx` informs the master `m` once the executor has finished, and `m` forwards this information to `w`. The failure handling is similar to that of the last two protocols. The master `m` informs all workers about the failure and is in control to either restart the subprotocol or to terminate the subprotocol.