UNIVERSITY OF KENT

DOCTORAL THESIS

Safe and Asynchronous Mixed-Choice for Timed Interactions

Author:
Jonah PEARS

Supervisor:
Dr. Laura BOCCHI

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Programming Languages and Systems Research Group
School of Computing

November 7, 2024

Declaration of Authorship

I, Jonah PEARS, declare that this thesis titled, "Safe and Asynchronous Mixed-Choice for Timed Interactions" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

(Jonah Pears)

UNIVERSITY OF KENT

Abstract

CEMS

School of Computing

Doctor of Philosophy

Safe and Asynchronous Mixed-Choice for Timed Interactions

by Jonah PEARS

Mixed-choice has long been barred from models of asynchronous communication since it compromises the decidability of key properties of communicating finite-state machines. Session types inherit this restriction, which precludes them from fully modelling timeouts and thus, greatly limits their applications to real-world systems.

This thesis aims to address this deficiency by presenting Timeout Asynchronous Session Types (TOAST for short) which model timed, asynchronous communication in the binary setting. In this thesis we explore the relationship between timeouts and mixed-choice, and show how timing constraints can be deployed to ensure that a given mixed-choice is *safe* and does *not* compromise the decidability of properties of timed, asynchronous session types. TOAST extends prior works on asynchronous timed session types with the means to describe these *safe* mixed-choice.

The formal theory of TOAST features new behavioural semantics for timed, asynchronous mixed-choice and a process calculus featuring timeout processes, process timers and time-sensitive conditional statements to facilitate the implementation of not just timeouts, but more broadly *safe* mixed-choice. The unique features of the TOAST process calculus are based upon the structures found in real programming languages, such as the `receive-after` expression in Erlang. Following the theory of TOAST, this thesis presents a proof-of-concept toolchain implemented in Erlang for generating program stubs from TOAST.

Acknowledgements

First and foremost, I would like to thank my supervisor, Laura Bocchi, who has been instrumental in helping me to grow into the researcher I am today. I feel very fortunate to have worked with Laura. I am very thankful for all the guidance, support and encouragement that I have received from you throughout this journey, and appreciate the challenges that you have pushed me to overcome. Thank you.

Secondly, I would like to thank Andy King, Maurizio Murgia and Raymond Hu, who I have each worked with closely to co-author the contributions contained within this thesis. I appreciate the impact that each of you has left on myself and the work I produce. I also thank Simon Thompson for his technical insight of Erlang. Thank you all for your patience and for taking the time to help me improve.

Next, I would like to thank my fellow PhD students, friends and researchers with whom I have shared this journey. Those who I first met in Cornwallis: Caio, Calvin, Octave, Sophie (& J.B.); those who joined a little later: Humphrey & Paulo; and, those such as Alfie who I have only got to know over the past year.

I thank Hernán Melgratti for inviting me to visit Buenos Aires on a secondment, and I thank Javier Pimás for being a familiar face when I needed it most. Both of you helped and supported me in different ways and I would like to acknowledge how much of an impact both of you had on my time in Buenos Aires.

I would like to thank Stefan Marr, Mark Batty and Rogério de Lemos who have guided my progression over this period, in addition to Anna Jordanous and Sonnary Dearden who have both been accommodating and understanding whenever I have sought their advice. I thank my examiners Olaf Chitil and Alceste Scalas for their insightful comments and feedback on my thesis.

Last but certainly not least, I would like to thank my dearest partner, Max. I am so happy that we have been able to share this period of growth, and I am excited to explore whatever comes next for us both. Your ever-present support throughout the past three years cannot be overstated, and I only hope that I may do the same for you. Finally, our cherished cats, Mulch and Spook, have been an endless source of joy, warmth and mischief, and I feel they deserve an honourable mention here.

Contents

List of Figures

Chapter 1

Introduction

The internet is a vast computer network which enables different machines from across the globe to communicate and exchange information. Such interactions are orchestrated by *communication protocols*, which specify how each party (i.e., process or machine) is expected to behave, enabling two (or more) potentially unaffiliated parties to exchange information successfully, without any prior orchestration.

Session types (Bettini et al. 2008; Honda et al. 1998, 2008) formally model the behaviour of processes in concurrent systems that interact via message-passing as part of an ongoing communication *session*. A session type specifies how interactions should occur in a session – send or receive, the message types, and their causality with other interactions – and provide guarantees on certain behavioural properties (namely, **progress**). Until recently, such models have been unable to provide such guarantees for the certain behaviours, such as **mixed-choice** which is one of the primary focuses of this thesis. (We discuss mixed-choice in greater detail later, in Section 2.3.)

1.1 Prelude

It is desirable that individual processes of a concurrent system do not get stuck and instead continue to perform interactions until termination (i.e., for the system to make progress). Gouda et al. (1984) proved that while the progress of two Communicating Finite State Machines (CFSM) was undecidable in general, it could in fact be guaranteed given that together, both CFSM: (i) are **compatible**, (ii) are **deterministic**, (iii) communicate over **unbounded channels**, and (iv) have **no mixed-states**.

Ultimately, session types (Carbone et al. 2008; Honda et al. 2008, 2016; Yoshida et al. 2007) employ similar restrictions in order to guarantee behavioural properties, such as progress. Deniélou et al. (2012) extended the (binary) result of Gouda et al. (1984) to the multiparty setting, establishing a correspondence between CFSM and Multiparty Session Types (MPST) (Honda et al. 2008). Session types are intrinsically deterministic since in a choice no two labels may be the same, and also typically assume: (i) each pair of participants in a session are connected via pairs of unidirectional channels with **unbounded buffers**, and, in order to guarantee progress they require: (ii) a notion of **compatibility** amongst participants (e.g., duality in a binary system), and (iii) absence of **mixed-states** (Deniélou et al. 2013). Hereafter, we will only refer to **mixed-choice** since it also encompasses mixed-states. (Discussed later in Section 2.3.)

1.2 Motivations

Timed extensions of session types (Bartoletti et al. 2017; Bocchi et al. 2019, 2014) are able to formally model the behaviour of time-sensitive interactions such as those of real-time systems. Such models provide a natural link to certain time-sensitive patterns and protocols found in real-world systems (namely, for the sake of this thesis, *timeouts*). Timeouts are a common interaction structure found in protocols for asynchronous communication. However, for reasons we will discuss further in Section 2.3, the behaviour of timeouts is unable to be effectively modelled using (asynchronous) session types.

As we will discuss later in Section 2.1, session types have various applications – ranging from static verification to runtime monitoring to code generation – that can benefit real-systems. For example, static verification can be used to guarantee the presence (or absence) of certain behavioural properties before the program is executed, enabling circumstances that could lead to crashes caused by deadlocks to be checked for and verified against. However, since timeouts are unsupported (and typically barred) from definitions of asynchronous session types, systems featuring them (e.g., web and cloud services) are unable to benefit from any of these applications, regardless of how widespread timeouts have become. Therefore, it is beneficial to expand the descriptive capabilities of asynchronous session types so that a wider

range of real-world systems can be captured, modelled and verified. This is the primary motivation behind the contributions of this thesis, and to this end we focus our efforts on timeouts. The secondary motivation is to help bridge the gap between the theory and practice, since aside from improving the descriptive capabilities of the theoretical models, we must also provide the technical infrastructure for these advancements to be further used, developed and continue to have impact.

1.3 Thesis Overview

This thesis is primarily concerned with *asynchronous* communication between two participants in the *timed* scenario, and presents several contributions based on session types (Bettini et al. 2008; Honda et al. 2008). First, in Chapters 3 to 5 we discuss the contributions of Pears et al. (2023, 2024c) which are extensions of existing theory on timed, asynchronous session types by Bocchi et al. (2019). Later, in Chapter 6 we discuss the contribution of Pears et al. (2024a) which provides a link between the theory presented in Chapters 3 to 5 with real programs, by means of a proof-of-concept toolchain for automatically generating code implementations of a given protocol specification.

1.3.1 Outline

The remainder of this thesis is structured as follows:

Chapter 2 We first provide a background for this thesis, covering the relevant fields, notions and formalisms. This chapter also discusses related work, and goes into specific detail in either the pursuit of priming the reader for the later contribution chapters, or for the purpose of disambiguation.

Chapter 3 Introduces Timeout Asynchronous Session Types (TOAST), shows how to ensure a given mixed-choice is 'safe' and proves that progress can still be guaranteed in the presence of safe mixed-choice. The contributions of this chapter were first presented in Pears et al. (2023).

Chapter 4 Presents TOAST processes – which extend the π-calculus – for implementing TOAST types. First shown in Pears et al. (2023), updated in Pears et al. (2024b,c).

Chapter 5 This chapter introduces the typing system for TOAST, which was first presented in Pears et al. (2024b) and later updated in Pears et al. (2024c).

Chapter 6 Discusses a work-in-progress toolchain for generating Erlang stub programs from the theory of TOAST. This toolchain also includes a generic runtime monitoring program for Erlang, which can be configured for either runtime verification or enforcement. The contents of this chapter were first presented in Pears et al. (2024a), and aims to help bridge the gap between theory and implementations.

Chapter 7 We conclude this thesis with a summary of the contributions in Chapters 3 to 6, review against the relevant literature, and outline future work.

Appendix A Contains the proofs of type progress.

Appendix B Contains the proofs of subject reduction.

Appendix C Contains supplementary material, such as restated theory.

1.3.2 Objectives

Concretely, the contributions of this thesis aim to address the following:

Can we use mixed-choice to model timeouts and still guarantee progress?

This is the primary aim of the contributions of this thesis, and can be broken down into smaller sub-objectives:

1. **What is the relationship between the behaviour of timeouts and mixed-choice?**
   This serves as a preliminary to the rest of the thesis that we promptly address in Chapter 2, yielding a notion of '*safe*' mixed-choice which corresponds to the behaviour of timeouts.

2. **How can we ensure that a mixed-choice is *safe*?**
   Given that we aim to use mixed-choice to model timeouts, we need to be able to check that the behaviour prescribed is only that of a timeout (or something

dually similar). This is achieved in Chapter 3 using *well-formedness rules* of types to ensure the interaction structure of a given mixed-choice type is *safe*.

3. **Are types with *safe* mixed-choice guaranteed to preserve progress?**
   It is crucial to ensure that the inclusion of safe mixed-choice does not compromise guarantees of type progress for a system. We must show that safe mixed-choice do not introduce unsafe and undesirable behaviour. The proofs are discussed in Chapter 3 and can be found in Appendix A.

4. **How do we implement the *dual* of a timeout?**
   Since we are studying the behaviour of mixed-choice under the constraints of a timeout in the binary setting, our types inherently facilitate the notion of a *counterpart* of a given timeout (i.e., where a timeout receives first and then sends later, a *counterpart* must send first and receive later).

   In Chapter 4 we present a process calculus with unique features intended for implementing these behaviours found in the types. Then in Chapter 5 we show how to verify that an implementation adheres to the behaviour prescribed by the types. The relevant proofs can be found in Appendix B.

   Later, in Chapter 6 we put this in practice by showing how to implement such behaviour in Erlang, when we present a tool for generating Erlang programs.

1.4 Attributions

Before proceeding, we take a moment to formally disclose the relevant works of this thesis, and the degree to which the author contributed towards their creation.

I (Jonah Pears) am the author of this thesis, and the main author of each of the following works of which constitute the contributions of this thesis. Undoubtedly, my supervisor and co-authors were crucial throughout the course of creating each of these works by offering their guidance, providing feedback, engaging in insightful discussions and assisting in writing up. Nevertheless, for each of the following pieces of work I (Jonah Pears) provided the majority of the contribution.

1. Jonah Pears, Laura Bocchi, and Andy King (2023). “Safe asynchronous mixed-choice for timed interactions”. In: *COORDINATION*. vol. 13908. LNCS. Springer,

pp. 214–231. DOI: 10.1007/978-3-031-35361-1_12

This work builds upon Asynchronous Timed Session Types (ATST) (Bocchi et al. 2019) as a foundation. Extending the work of Bocchi et al. (2019), I primarily formalised the theory and produced the resulting proof.

2. Jonah Pears, Laura Bocchi, Maurizio Murgia, and Andy King (2024c). *Timeout Asynchronous Session Types: Safe Asynchronous Mixed-Choice For Timed Interactions*. arXiv: 2401.11197v2 [cs.LO]

   This work is an extended version of Pears et al. (2023). I continued extending the work of Bocchi et al. (2019), and again, formalised the theory and produced the resulting proof. Notably, while the new co-author provided some assistance with the proof, I remain the primary contributor.

3. Jonah Pears, Laura Bocchi, and Raymond Hu (2024a). "Erlang on TOAST: Generating Erlang Stubs with Inline TOAST Monitors". In: *Erlang Workshop*. ACM, pp. 33–44. ISBN: 9798400710988. DOI: 10.1145/3677995.3678192

   I developed this tool myself, following discussions with my co-authors. Notably, the tool is a fork of the tool that accompanied Bocchi et al. (2023) which I had originally used as a starting point and reference. The resulting tool serves a different purpose and contains very little of the original.

Chapter 2

Background

This chapter serves to: (i) set the scene for the rest of the thesis by discussing the literature of session types as a whole (and, when necessary other seminal works) in Sections 2.1 and 2.2, and (ii) review the works in the literature that are closely related with the contributions of this thesis in Section 2.3.

2.1 Session Types

In this section we discuss session types (Bettini et al. 2008; Carbone et al. 2008; Dardha et al. 2017; Honda et al. 2008, 2016; Scalas et al. 2019; Yoshida et al. 2007), which facilitate the independent design, development, verification and deployment of component processes in the context of a concurrent system.

2.1.1 Core Principles

Works on session types are typically comprised of two halves: (i) the session types themselves, and (ii) the process calculus, an extension of the π-calculus (Milner 1999); along with a typing system for checking that the behaviour of the process adheres to the behaviour prescribed by the types, and a set of reduction rules describing how the process may evolve through its lifetime. Following the π-calculus (Milner 1999), higher-order communication is another core component feature of session types, which enable ongoing sessions to be *delegated* by one process exchanging a pair of channels with endpoints that connect to another process.

Session types utilise type-checking in order to provide guarantees on the behaviour of a given process (typically derived from the π-calculus). In this way,

type-checking avoids the state-explosion encountered by other methods of static verification which exhaustively check every possible interleaving of interactions, with the aim of completely mapping out the state-space of the system to check for certain properties. Instead, type-checking entails checking that the behaviour of a process falls within a subset of the state-space corresponding to the types, where processes therein are guaranteed certain properties. The crux of this approach relies on the behaviour of types themselves being *decidable*.

Decidability

Gouda et al. (1984) proved that the progress of two CFSM is undecidable in general, but that it *is* decidable for two CFSM that are: (i) **deterministic**, and (ii) **compatible**. Additionally, Gouda et al. (1984) proved that in addition to (i) and (ii), if two CFSM: (iii) have **unbounded buffers**, and (iv) do not contain **mixed-choice**, then their behaviour is guaranteed to be *free from deadlocks*, i.e., **deadlock freedom**. As previously outlined in Section 1.1, these restrictions are inherent in session types; where (i), (iii) and (iv) are embedded into the syntax, and (ii) is instantiated as duality (in binary sessions). The correspondence between CFSM and MPST (Bettini et al. 2008; Honda et al. 2008, 2016) has since been formally established by Deniélou et al. (2012, 2013). Session types utilise type-checking in order to provide additional properties of their behaviour – primarily: **communication safety**, **progress** and **session fidelity**.

Properties

In session types, a typing system evaluates whether a given process correctly implements behaviour as prescribed by a given type, i.e., if the process is *well-typed*. A standard property of such typing systems is **subject reduction**, which states that a well-typed process will remain well-typed after making a reduction step; hence this property is also referred to as *type preservation*. It is often instrumental to prove **subject reduction** before being able to provide further guarantees of properties. We now discuss a few such properties, namely those relating to **progress**; see Dardha et al. (2017) and Scalas et al. (2019) for a comprehensive discussion.

Session Fidelity dictates that the behaviour of a well-typed process follows as prescribed by its corresponding type. Specifically, as defined by Scalas et al. (2019), given a process that is well-typed against a type, if the type can perform an action then so can the process. In this regard, **session fidelity** is the opposite of **subject reduction** (Brun et al. 2023) and typically follows from **subject reduction**.

Communication Safety entails that for any message that arrives in the channel buffer of a process: the message is *expected* and able to be successfully received by the process (**freedom from unspecified receptions**) and the message itself is as prescribed by the types (**type safety**).

Deadlock Freedom indicates a process will not get *stuck*, unable to make any steps while not terminated. For example, in the context of a (binary) session a deadlock can occur if one participant is waiting to receive a message that never enters their queue. The work of Kobayashi (2002), Kobayashi et al. (2010), and Padovani (2014) discuss the broader property of **lock freedom**, which encapsulates both **deadlock freedom** and **livelock freedom**, i.e., a 'livelock' refers to an instance where an individual may indefinitely make steps while progress of the overall system is inhibited. Recent works (Kobayashi et al. 2010; Scalas et al. 2019) refer to **lock freedom** as **liveness**.

Progress varies depending on the context, but broadly indicates that the system does not get stuck and that progress made is as intended. For instance, Carbone et al. (2014) explore the relationship between **lock freedom** and **progress** and argue that "*progress can be thought of as a generalisation of lock-freedom ...*" in their context. In general, progress does follow **deadlock freedom** and **liveness**, given that **communication safety** and **session fidelity** are often assumed to already hold. Altogether:

i the interaction structure will not cause errors to occur (**communication safety**),

ii interactions occur as prescribed by the types (**session fidelity**),

iii the system will not reach a deadlock (**deadlock freedom**),

iv if an action *can* be performed then an action *will* be performed (**liveness**).

See Bettini et al. (2008), Coppo et al. (2016), and Dezani-Ciancaglini et al. (2007) for a more detailed discussion on how to guarantee the **progress** of session types.

2.1.2 Applications in Programming Languages

Session types provide a seamless link between the theory and practice of programming languages, and open up the possibility of providing guarantees of a program's behavioural properties, i.e., how it will behave at runtime.

We now conduct a sweeping survey of the literature pertaining to applications of session types in programming languages. First, we focus on the seminal works, and then discuss existing implementations categorically. Later, in Section 2.3 we will return to review some of the works discussed here that are related to mixed-choice.

Java

Hu et al. (2008) was the first work that implemented session types into a programming language (Java), integrating them into a Java compiler. This work consisted of both (i) instrumenting the behaviour necessary to implement process in Java that correspond to the (then current) theory of session types (Gay et al. 2005; Honda et al. 1998; Takeuchi et al. 1994), and (ii) providing a means of verifying communication by: (a) statically type-checking individual implementations against a type, and (b) dynamically verifying that the types of each implementation are *compatible*.

Scribble

Originally presented by Honda et al. (2011), Scribble is a language for designing protocols using session types, and was later featured in a toolchain for multiparty protocols by Yoshida et al. (2014). We mention this now to acknowledge the significance of Scribble, since it serves as the foundation of many implementations of session types in other programming languages; e.g., the works of: Castro-Perez et al. (2019, 2023), Demangeon et al. (2015), Fowler (2016), Hou et al. (2024), Hu et al. (2016), Lagaillardie et al. (2020, 2022), Neykova (2013), and Neykova et al. (2017a, 2018, 2014, 2017b, 2013). For this reason we briefly survey the development of Scribble.

The toolchain by Yoshida et al. (2014) builds upon an implementation of session types in Java (Hu et al. 2010, 2008), extended with multiparty sessions. This toolchain was later extended by Neykova et al. (2019) who establish a correspondence between a 'Featherweight Scribble' and CFSM. A further extension by Yoshida et al. (2021) present *νScr*, an updated version of the Scribble toolchain with a closer correspondence to the formally grounded theory of session types (via the correspondence with CFSM and MPST by Deniélou et al. (2012, 2013)).

To briefly digress, one example of an implementation of session types that does *not* build upon Scribble is that of Castro-Perez et al. (2021), who present a Domain Specific Language (DSL) called Zooid, which is instead embedded within *The Coq Proof Assistant* (2024) and facilitates processes that are 'well-typed by construction'.

Static Verification

Hu et al. (2010) implement static type-checking of binary session types in Java for event-driven programming (extending Hu et al. (2008)). Mostrous et al. (2011) present a 'featherweight' calculus for type-checking a subset of Erlang programs, based on the theory of session types. Hou et al. (2024) and Lagaillardie et al. (2020, 2022) present varying implementations for type-checking Rust programs (discussed further later in Section 2.1.2). Other instances include for Scala (Scalas et al. 2016) and Internet Of Things (IoT) devices (Iraci et al. 2023).

Dynamic Verification

Hu et al. (2013), Neykova (2013), and Neykova et al. (2013) present a toolchain for Python: "the first implementation of runtime verification" using MPST. Protocols are first designed using Scribble, and then used to generate runtime monitors for each participant. In this way runtime monitors verify the behaviour of each participant *locally* at runtime, and even ensure that the protocol is not violated. (The latter refers to an *enforcement* monitor which may take actions to ensure the protocol is not violated. We discuss runtime monitors in further detail in Chapter 6.) Along with the toolchain, the authors present the "Conversation Application Programming Interface (CAPI)" library which is what enables Python programs to implement MPST, containing the

essential features for sessions and message passing. Notably, Hu et al. (2013) focus on event-driven programming and implement an extension of MPST with support for *interrupts*, a communication pattern that can describe **mixed-choice**.

Demangeon et al. (2015) formalise the work on interrupts by Hu et al. (2013), which later led to Fault Tolerant Multiparty Session Types (FTMPST) by Peters et al. (2022) which we discuss later in Section 2.3. Tangentially, Viering et al. (2021) implement FTMPST in Scala, building on the work by Demangeon et al. (2015) and Hu et al. (2013) for generating runtime monitors.

Neykova et al. (2014) present Multiparty Session Actors (MSA), a Python implementation which supports actor-based programming. After designing a distributed protocol in Scribble, an Finite State Machines (FSM) is generated which can be used for dynamic verification at runtime whereby each message received by the actor process is checked against the FSM. Fowler (2016) extends this work on MSA to Erlang for generating runtime monitors, utilising the supervision tree structure of Erlang for session coordination, instantiation and reporting violations. Later, Neykova et al. (2017a) implement a form of Timed Multiparty Session Types (TMPST) for runtime monitoring of Erlang programs following the theory laid out by Bocchi et al. (2013, 2014), with a strong focus on *enforcement* monitors (as in Neykova et al. (2013)).

As we can see, the tried and tested approach when implementing session types for generating runtime monitors is as follows: (1) take as input a protocol specification (typically Scribble), (2) extract equivalent FSM, and (3) generate runtime monitor. We discuss runtime monitoring in further detail in Section 6.3.3.

Code & API Generation

Hu et al. (2016) show how to generate Application Programming Interface (API) implementations for endpoints (i.e., the participating processes in a session) in Java. The methodology is similar to generating runtime monitors whereby it builds upon Scribble, except that instead of generating a runtime monitor to dynamically verify the communications of a participant, it provides the 'boilerplate' code necessary to correctly implement the behaviour specified by a protocol (again, written in Scribble).

The key difference here is that it actually provides a portion of the code that is **correct-by-construction**, due to the close correspondence between the code and model that has been statically verified. Notably, the programmer must still implement the rest of the functionality. Later, this approach was applied by Castro-Perez et al. (2019) who present an implementation in Go.

Presented in Neykova et al. (2018) is a toolchain for generating F# code from Scribble protocols (Yoshida et al. 2014), which are based on MPST. Neykova et al. (2018) offers an immersive, intuitive and accessible 'interface' for their code generation, which takes the form of code snippet hints/suggestions of the next action to be performed. This provides the programmer with 'live' feedback as they build their own implementation and insert the generated code snippets.

Rust

The work surrounding the Rust implementation of session types offers the full range of static and dynamic verification, in addition to code generation. First, Lagaillardie et al. (2020) implement MPST in Rust (extending the original work by Kokke (2019) on binary sessions), providing a means of type-checking Rust programs against protocols designed using the Scribble (Yoshida et al. 2014) toolchain (or designed manually). Lagaillardie et al. (2022) take this further with Affine Multiparty Session Types (AMPST) (an extension of the *binary* work by Mostrous et al. (2018)) along with a Rust implementation: "`MultiCrusty`, *a library for safe multiparty communication Rust programming*." `MultiCrusty` supports both (i) correct-by-construction code generation from a Scribble protocol (top-down approach), and (ii) static type-checking and compatibility checking of existing Rust programs (bottom-up approach).

Recently, Hou et al. (2024) extend the `MultiCrusty` library to support *timed* protocols, drawing upon the theory of ATST (Bocchi et al. 2019) and present Affine Timed Multiparty Session Types (ATMP). As a timed overhaul of Lagaillardie et al. (2022), they also required a timed overhaul of νScr. We discuss the work by Bocchi et al. (2019) in Section 2.2 and Hou et al. (2024) in Section 2.3.

2.2 Time-Sensitive Session Types

We now discuss the session type variants capable of modelling time-sensitive communication in real-time systems. Before we can discuss time-sensitive session types, we must first discuss Communicating Timed Automata (CTA) by Krcál et al. (2006) which, following the correspondence between CFSM and session types established by Deniélou et al. (2012, 2013), serves as the basis for variants of session types capable of modelling time-sensitive protocols (Bartoletti et al. 2017; Bocchi et al. 2019, 2014) of real-time systems.

2.2.1 Communicating Timed Automata

Krcál et al. (2006) combined Timed Automata (TA) by Alur et al. (1994) with CFSM by Brand et al. (1983) to produce CTA.

Compared with normal CFSM, the most distinguishing feature of CTA are the *timing constraints* imposed on each of the actions, specifying *when* they may be performed. Timing constraints specify whether an action may be performed depending on the valuation of a *set of local clocks*, with each participant owning their own set. In this way, each participant may keep track of their timeliness using these clocks, and may reset to zero. Actions in CTA are annotated with timing-constraints which determine when an action may be performed (serving as *pre*-conditions that must be satisfied) in addition to a set of clocks to be reset after the action has been performed, but before progressing to the next state (serving as *post*-conditions that must be adhered to). Notably, unlike most session types, CTA are able to describe mixed-choice since as they do not aim to provide the same innate behavioural guarantees, they are not constrained in the same way as session types.

Altogether, the features of CTA are general enough to encapsulate the behaviour of interactions in a real-time system. The timed scenario is intrinsically more complex and difficult to verify than the untimed, not only due to the encumberment for static verification caused by modelling the additions of the clocks and constraints, but due to the necessity to now check the *feasibility* of each action given its own timing constraints and those that preceded it. Compounded with checking all possible interleavings of interactions, the problem can quickly become intractable.

Timed & Urgent Semantics

Arguably the most significant feature of the work by Krcál et al. (2006) is the timed, asynchronous semantics of CTA. The synchronised semantics for asynchronous communication (or *asynchronous synchronised semantics*) requires the assumption that time passes at the same rate for each participant, evenly over the whole system and denote two distinct types of actions, one for time passing and another for discrete actions. Crucially, the semantics rely on the notion of *receive-urgency*, which dictates that if a message arrives in the channel-buffer (or queue) of a participant and the participant has a corresponding receiving action, then that action must be taken urgently (i.e., before any time may pass). In practice, this renders the asynchronous communication to appear rather *synchronous*, since CTA do not model latency and therefore, messages are received within the same time-step they were sent. It should be noted that both network latency and delays are outside the scope of what CTA capture, but can be accommodated for; e.g., modelling communication cost (Castro-Perez et al. 2020), Lamport timestamps (such as in Kuhn et al. (2023)), or by augmenting the timing constraints to compensate.

The urgent semantic presented by Bartoletti et al. (2018) builds upon the (timed) *synchronised semantics for asynchronous communication* of CTA by Krcál et al. (2006). The urgent semantics are crucial to enforce *receive urgency* and ensure that the *latest enabled* action is not missed in Bartoletti et al. (2018). Later, Bocchi et al. (2019) adopted this in ATST, which we discuss in Section 2.2.3.

2.2.2 Timed Multiparty Session Types

Building upon the work by Deniélou et al. (2012, 2013) which established the correspondence between CFSM and MPST, Bocchi et al. (2014) present TMPST, which effectively combined MPST and CTA, yielding a strain of asynchronous session types capable of modelling real-time systems. Similarly to CTA, the types of TMPST feature local clocks for each participant and, a set of timing-constraints and clocks to be reset on each action. Naturally, TMPST are deterministic, have a notion of *multiparty compatibility*, while also inheriting the restriction disallowing mixed-choice. The semantics (of types) of TMPST require that delays (time-steps) do not cause actions to

become *unsatisfiable*, requiring that at least one action may be performed in the future. (This is later called *future-enabled* by Bartoletti et al. (2018) and Bocchi et al. (2019).)

Notably, only the *types* of TMPST have clocks and the timing-constraints inherited from CTA. These *types* are then used to type-check processes, and verify that interactions occur at the correct times. However, the processes themselves only feature a single 'timed' process, the *delay* process, while the interaction processes were unchanged from standard MPST. Bocchi et al. (2014) highlighted that in the timed scenario, the term **error-freedom** suggests that the *timeliness* of the interactions will not cause an error, rather than the structure of the interactions (the key difference being *when* an interaction occurs, rather than *what* interaction). To this end, Bocchi et al. (2014) rely on the type-checking to ensure that interactions will only occur at correct timings as prescribed by the types and argue that it is sufficient to annotate the processes with the types in order to prove that the no errors occur due to the timeliness of interactions. Crucially, Bocchi et al. (2014) rely on results of prior (untimed) works (Bettini et al. 2008; Honda et al. 2008) in order to guarantee progress for TMPST processes, by removing all delays from processes to derive *untimed* counterparts.

2.2.3 Asynchronous Timed Session Types

Bocchi et al. (2019) present ATST in the binary setting. Crucially, Bocchi et al. (2019) intentionally diverge from TMPST (Bocchi et al. 2014) in order to address some of its shortcomings. Firstly, the semantics of types of TMPST is overly restrictive. The condition '*wait-freedom*' does not allow time to pass over a system whilst a participant is waiting to receive a message. To remedy this, Bocchi et al. (2019) takes an approach more similar to the *asynchronous synchronised semantics* first presented by Krcál et al. (2006) for CTA, and which had since been adapted by Bartoletti et al. (2018).

Secondly, ATST feature time-sensitive branch/receive processes which must specify an upper-bound of which a corresponding message *must* be received within, before then reaching an error state (unlike the semantics of TMPST, which does not allow time to pass while a process was waiting to receive a message). Furthermore, Bocchi et al. (2019) argue the increased significance of **safety** in the timed scenario, building upon the notion of **error-freedom** in TMPST by Bocchi et al. (2014). Since ATST can

actually reach an error state, Bocchi et al. (2019) are able to prove more directly that their typing system is sufficient to ensure that a well-typed process will not reach an error state. Bocchi et al. (2019) rely on results of prior (untimed) works (Bettini et al. 2008; Dezani-Ciancaglini et al. 2007; Honda et al. 2008) in order to guarantee progress of ATST processes, by deriving *untimed* counterparts (similarly to Bocchi et al. (2014)). However, as we shall later uncover in Chapter 5 and discuss further in Chapter 7, this approach turns out to be inadequate when modelling timed timeouts, since removing the timing-constraints yields potentially unsafe mixed-choice which can be problematic and compromise progress.

2.2.4 Affine Timed Multiparty Session Types

Recently, Hou et al. (2024) present ATMP, an extension of AMPST by Lagaillardie et al. (2022) combined with ATST by Bocchi et al. (2019), and are capable of modelling *timed* timeouts. Clearly, this work is closely related to the contributions of this thesis, since it closely aligns with some of the objectives this thesis aims to address (given in Section 1.3.2). Crucially, both the contributions of this thesis (namely TOAST given in Chapters 3 to 5) and AMPST by Hou et al. (2024) build upon the work of ATST by Bocchi et al. (2019). Therefore, we relegate the discussion of Hou et al. (2024) to Section 7.1.1, in light of Chapters 3 to 6 to allow for a more meaningful comparison and discussion between the works of TOAST and AMPST.[1]

2.2.5 Synchronous Timed Session Types

The work of Bartoletti et al. (2017) present (binary) Timed Session Types (TST) in the synchronous setting. Instead of relying on a notion of *duality*, Bartoletti et al. (2017) presents a means of determining if two parties are *compliant*, given their sets of constraints. This allows the timing constraints on actions to be constructed without considering the dual, and therefore, avoiding such issues as those encountered

[1]The timeliness of the work by Hou et al. (2024) (which appeared June 2024 – 3 months before this thesis was originally due to be submitted) further emphasises the decision to relegate such discussions to Section 7.1.1, since the initial work on TOAST (Pears et al. 2023) appeared a whole year before, and the extended version (Pears et al. (2024b) [v1]) first appeared on the arXiv in January 2024 (5 months before Hou et al. (2024)). Notably, Hou et al. (2024) make no reference to either of the contributions pertaining to this thesis that were available at the time (Pears et al. 2023, 2024b). We have since contacted the authors regarding this who have stated that they will amend this in future work.

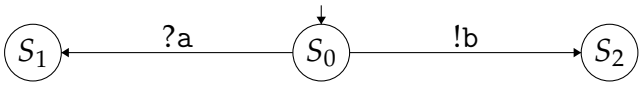
by so-called *junk-types*, which have infeasible timing constraints (discussed later in Example 3.2.2). For example, if one party expects to receive both messages a and c after 3 and before 5 time units respectively, then deadlock freedom can still be guaranteed if there is some *compliant co-party* that always sends both before 5.

2.3 Session Types, Timeouts & Mixed-Choice

In this section we discuss session types, timeouts and mixed-choice. We first explore the relationship between timeouts and '*safe*' mixed-choice. Next, we survey work on time-sensitive session types, since they are the most natural fit for timeouts. Then, we discuss other works that accommodate either timeouts or mixed-choice.

2.3.1 What is a Mixed-Choice?

In Eq. (2.3.1), an individual CFSM in state (S_0) can perform two actions: either receive (?) message '`a`' and continue as (S_1), or send (!) message '`b`' and continue as (S_2).



(2.3.1)

We say state (S_0) is *mixed* since it has outgoing edges that describe both a send (!) and receive (?) action. Though crucially, only one of the options (`?a` or `!b`) is actually taken from state (S_0) – a choice must be made between the two. A *mixed-choice* may have multiple send (!) and receive (?) actions able to be performed from the same state (though again, only one of them is actually performed).

Example 2.3.1 (Problematic Mixed-Choice)**.** We now illustrate the issue with mixed-choice in the *asynchronous* setting, as described by Gouda et al. (1984). Recall the CFSM show in Eq. (2.3.1), which we compose with its dual $(\overline{S_0})$ below:

(2.3.2)

The problematic behaviour arises since both: $(S_0) \xrightarrow{!b} (S_2)$ and $(\overline{S_0}) \xrightarrow{!a} (\overline{S_1})$ can occur at the same time. Such an occurrence is facilitated by the asynchrony of interactions, since the message of one may still be 'in transit' before arriving at the other to be

received. The consequences of this are that we no longer have any guarantees that the newly reached states of (S_2) and $(\overline{S_1})$ are capable of receiving '`a`' and '`b`' (respectively), which is typically ensured by duality (since dual types have corresponding actions). Aside from duality being broken, the newly arrived messages may be *unexpected*, unable to be received. Furthermore, both individuals now risk getting stuck from waiting to receive a message that will never be sent. Such concerns are a violation of **communication safety** (Carbone et al. 2008; Honda et al. 2008, 2016), which is crucial for guaranteeing progress. △

2.3.2 What is a Timeout?

Timeouts are an important mechanism for handling failure and unexpected delays, and are commonplace in protocols on the web. For instance, the Simple Mail Transfer Protocol (SMTP) stipulates:

> "An SMTP client MUST provide a timeout mechanism. It MUST use per-command timeouts rather than somehow trying to time the entire mail transaction. Timeouts SHOULD be easily reconfigurable, preferably without recompiling the SMTP code." – (Klensin 2008 Section 4.5.3.2)

Broadly speaking, a 'timeout' typically refers to a timer being set and then waiting for a certain condition to be met before the timer expires, after which a timeout is 'issued' (or 'triggered'). Generally, the intuition is that it is preferred that a timeout does not occur, and that the condition should be met before the timer expires. For example, timeouts can be imposed on access to a shared resource on a network (Peralta et al. 2003) in order to ensure individuals cannot seize the resource indefinitely.

Alternatively, in a client-server setting:

> "The objective of Time-Out Protocol (TOP) (Time-Out Protocol) is to assure that when clients keep the server idle for too long, their access to the server will be disabled. Then, other clients can also access the server." – (Esfarjani et al. 1998 Section 2.2)

In essence, "timeout mechanisms are initiated to escape from problematic situations" (Efthivoulidis et al. 1999 Section 3).

Timeouts in Asynchronous Communication

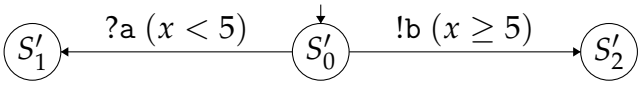
As we have so far seen, timeouts enable an individual process to react to the timeliness of their environment. Intuitively, in the asynchronous setting since send actions are *non-blocking* (i.e., they can be performed without waiting for the recipient to be ready – the 'fire and forget' approach) then timeouts do not make sense from the perspective of a sender. However, since receiving actions remain *blocking*, a recipient can only wait to receive a message and therefore, timeouts allow recipients to stop waiting and otherwise 'disengage' from an interaction.

> "...the sender sends a message, which is forwarded to the receiver's mailbox, and continues its job. A fault is detected when the delivery to the receiver's mailbox cannot be completed within a timeout period (specified by the application) or when the message is not retrieved from the mailbox by the receiver within a timeout period."
>
> – (Efthivoulidis et al. 1999 Section 4.1)

Note, unless otherwise stated, throughout the contributions of this thesis we assume that the medium over which messages are exchanged by perfect and without fault. Therefore, following the above (Efthivoulidis et al. 1999 Section 4.1), we assume that when a 'timeout' is issued, that a message was unable to be received by the recipient; i.e., either the channel buffer/queue (or mailbox) was empty, or there was no action that could receive the contents of the buffer/queue.

2.3.3 Timeouts & Mixed-Choice

We will now discuss timeouts and illustrate their relation with mixed-choice. Recall the CFSM with mixed-choice in Eq. (2.3.1). Observe how by annotating the actions with timing constraints we can describe the behaviour of a timeout:



(2.3.3)

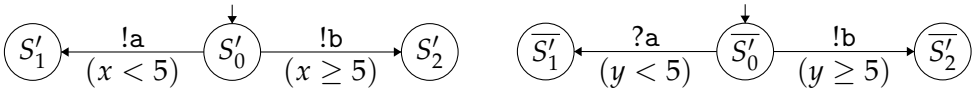
From state (S'_0) we may now only proceed to (S'_1) if we receive (?) message 'a' within 5 time units. Otherwise, 5 time units pass and we may only send (!) message 'b' and proceed to (S'_2). Returning once more to the SMTP:

> "An SMTP server SHOULD have a timeout of at least 5 minutes while it is awaiting the next command from the sender."
>
> – (Klensin 2008 Section 4.5.3.2.7)

Adapting Eq. (2.3.3), we can describe the above snippet of an SMTP server's behaviour by setting the time units to be **minutes**, message 'a' to be the next command from the sender, and message 'b' the timeout sent to the sender (i.e., the client).

Herein lies the crux of our contribution in Chapter 3: since timeouts correspond to a strict subset of mixed-choice (in models of asynchronous communication), then by extending timed asynchronous session types to model timeouts we inadvertently facilitate some kinds of mixed-choice. In Chapter 3 we present session types capable of describing the behaviour of timeouts, and prove that they correspond to a *safe* subset of mixed-choice, i.e., do not exhibit problematic behaviour, as in Example 2.3.1.

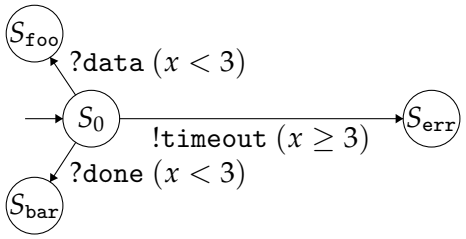
Example 2.3.2 (Safe Mixed-Choice)**.** Following Eq. (2.3.3), we once again extend the dual of Eq. (2.3.1) shown in Example 2.3.1 with timing constraints:



Note: throughout this thesis we use different clock labels (x, y) to better distinguish between the clocks owned by an individual and their dual. Regardless of the clock labels used, these states are in fact **dual** as their constraints yield the same behaviour.

```
1 fun s0() -> receive
2     {Q, data} -> foo();
3     {Q, done} -> bar();
4   after 3000  ->
5     Q ! {self(), timeout},
6     handle_timeout()
7   end.
```

(A) Erlang `receive-after` pattern



(B) Mixed-state CFSM representation

FIGURE 2.1: Timeout pattern in Erlang and CFSM

Observe, upon entering state (S_0') (or $(\overline{S_0'})$) the exchange of message 'a' is constrained to occur before 5 time units have passed, assuming all clocks are initially set to 0. Once 5 time units elapse, then message 'b' may be exchanged. Clearly, the timing constraints ensure the actions can only occur at distinctly separate moments, just like a timeout, and thus rule out the problematic behaviour in Example 2.3.1. △

Timeouts in Erlang

As we have established, it is intuitive that in the context of asynchronous communication, timeouts occur on the *receiving* end of an interaction. Since timeouts are commonplace in the protocols already deployed on the web, it is of no surprise that they may be already supported by programming languages with a focus on message passing; e.g., the '`receive-after`' expression in Erlang (Cesarini et al. 2016), or the '`time.After`' pattern in Go (Google 2024).

In Figure 2.1 we illustrate the correspondence between the Erlang `receive-after` expression in Figure 2.1a and the behaviour of timeouts in Figure 2.1b (similarly to Eq. (2.3.3) we are using a CFSM notation annotated with time constraints). Figure 2.1a describes waiting to receive from a process `Q`: either `data` and proceeding to `foo()`, or, `done` and proceeding to `bar()`. Otherwise, after `3000`ms (3 seconds) if neither message has been received then a `timeout` is sent to process `Q`, and the process proceeds to `handle_timeout()`. Clearly, Figure 2.1b describes the same thing, where state $(S_{\texttt{foo}})$ corresponds to `foo()`, state $(S_{\texttt{bar}})$ to `bar()`, and state $(S_{\texttt{err}})$ to `handle_timeout()`.

2.3.4 Time-Sensitive Session Types

Time-sensitive session types (Bartoletti et al. 2017; Bocchi et al. 2019, 2014), which extend session types with time constraints, inherit the same syntactic restrictions of session types and hence rule out mixed-states. This is unfortunate since in the timed setting, mixed-states are a useful abstraction for timeouts (as illustrated in Figure 2.1 and outlined in Section 2.3.3). Out of TMPST (Bocchi et al. 2014), TST (Bartoletti et al. 2017) and ATST (Bocchi et al. 2019), only the latter (ATST) addresses the issue of enforcing a prescribed upper-bound on a receiving process (denoting that an error-state is to be reached). While this is not itself a timeout, it is the only work in asynchronous timed session types that allows for distinguishing when a process should stop waiting to receive (i.e., excluding the very recent work of Hou et al. (2024) which, as previously mentioned we will discuss in detail in Section 7.1.1).

2.3.5 Mixed Sessions

Vasconcelos et al. (2020) introduce mixed-choice in untimed *synchronous* session types. They allow for mixed-choice with a single communication primitive. Choices in Vasconcelos et al. (2020) are non-deterministic and labels in a choice do not have to be unique. Additionally, options in a choice are either linear or *unrestricted*. This approach leads to patterns such as the producer/consumer pattern being elegantly represented by a single type, whereas in our system several would be needed. Effectively, the unrestricted action embeds into one of the options action within a choice, a recursive call that returns back to the choice. This is once again similar to the timeout-recursion behaviour found in Bocchi et al. (2022) and Iraci et al. (2023) (discussed later in Section 2.3.7), except that Vasconcelos et al. (2020) is *untimed* and *synchronous*.

Since Vasconcelos et al. (2020) work in the *synchronous* setting, parties must be actively engaging in the interaction at the same time (i.e., a 'handshake') and therefore, the issue showcased in Example 2.3.1 do not apply since multiple parties cannot send at the same time. In summary, Vasconcelos et al. (2020) do not address the issue of modelling *asynchronous* mixed-choice in session types.

2.3.6 Other Works

The timeout pattern naturally affords itself to fault-tolerance and failure handling, since it allows a process to stop waiting to receive in cases where it may get stuck. We now discuss session types that explore timeouts in such settings.

Multiparty Asynchronous Generalised π-calculus

Brun et al. (2023) present a strain of session types named Multiparty Asynchronous Generalised π-calculus (MAGπ) which are untimed and feature optional 'timeout branches' in the syntax of both branching types and processes. These 'timeout branches' are to be taken non-deterministically, and may in fact trigger regardless of whether another branch is viable (i.e., if a message is waiting to be received). This is because the work by Brun et al. (2023) focuses on handling failures of different kinds, including those where messages are lost or reordered. In fact, the semantics of MAGπ go as far as to enable messages denoted 'unreliable' from the queue of processes before they can be received, and this may explain why the timeouts in this work are non-deterministic. In addition to timeouts, MAGπ are also capable of modelling some forms of mixed-choice. However, since they are unstructured and non-deterministic, communication mismatches may occur and as such, the type/process must be designed to handle these failures. In order to model the counterpart of a timeout (i.e., where a send action is enabled for a duration before then waiting to receive) MAGπ stagger the timeout processes in an alternating pattern. While this is effective in their context, it does limit the descriptive capabilities of mixed-choice with only two distinct regions of sending and receiving actions.

Fault-Tolerant MPST

Peters et al. (2022, 2023) present FTMPST which, similarly to MAGπ by Brun et al. (2023), is untimed and includes an additional 'default branch' on branching processes that may be taken when no other message is received. Notably, this 'default branch' requires a default value to be provided, which is used by the continuation process in the case that nothing was received. Additionally, this approach relies on an external failure detector to inform the process when (or if) to take the default value. While

different from a timeout (since it is untimed) this behaviour does allow a process that would otherwise be stuck waiting forever, to instead make progress. By contrast, the connection to mixed-choice is far more muddied – in part due to the reliance on an external entity – but primarily due to the implication that the other party has indeed crashed, or that the communication medium has failed, and therefore, there is little reason for the subsequent sending of the so-called 'timeout' that would constitute the behaviour as a mixed-choice.

2.3.7 Outside Session Types

Further afield, coordination structures have been proposed that overlap with mixed-choice. For example, Deniélou et al. (2012) illustrate how fork and join, which permit messages within a fork (and its corresponding join) to be sent or received in any order, are reminiscent of mixed-choice. Affine sessions (Mostrous et al. 2018) support exception handling by enabling an end-point to perform a subset of the interactions specified by their type, but there is no consideration of time, hence timeouts. Before session types gained traction, timed processes (Berger et al. 2007) were proposed for realising timeouts, but lack any notion of counterpart.

Iraci et al. (2023) present Rate Based Session Types (RBST) for IoT systems. The "*periodic recursion*" in RBST features a similar construct where a timeout-recursive loop runs indefinitely, at a steady and fixed rate. Instead of timing constraints on individual actions, periods are imposed on recursive loops, specifying the rate at which each iteration must adhere, and optional deadlines which specify a fixed upper bound. A similar construct to timeout processes appears in Bocchi et al. (2022), which builds upon the Temporal Process Language (TPL) by Hennessy et al. (1995) and ATST by Bocchi et al. (2019) in a setting based upon mailbox-based communication. Bocchi et al. (2022) extend the branching process of ATST in a manner that resembles the `receive-after` pattern in Erlang which is capable of describing timeouts, as previously illustrated in Figure 2.1a.

Notations & Definitions

This thesis is self-contained and throughout any necessary notions, notations and definitions are provided. For convenience, all of these are later restated in Appendix C. We denote the end of a proof with '□' and the end of an example or remark with '△'. We write ($a \implies b$) to denote an implication from a to b. We write $a\,[b/c]$ to denote replacing every occurrence of c in a with b. We write 's.t.' as shorthand for '*such that*'.

Chapter 3

Introducing TOAST

This chapter presents the syntax, semantics and formation for TOAST, as first shown in work by Pears et al. (2023), and later featured in Pears et al. (2024c). TOAST are an extension of the (binary) ATST by Bocchi et al. (2019), with a well-disciplined (and hence safe) form of mixed-choice that are shown to preserve guarantees of progress, the corresponding proof of which can be found in Appendix A.

3.1 Clocks & Constraints

We start with a few preliminary definitions borrowed from TA by Alur et al. (1994). Let $\mathbb{X}$ be a finite set of clocks denoted x, y and z. A (clock) valuation ν is a map $\nu : \mathbb{X} \to \mathbb{R}_{\geq 0}$. The initial valuation is ν_0, where $\nu_0 = \{x \mapsto 0 \mid x \in \mathbb{X}\}$. Given a time offset $t \in \mathbb{R}_{\geq 0}$ and a valuation ν, $\nu + t = \{x \mapsto \nu(x) + t \mid x \in \mathbb{X}\}$. Given ν and $\lambda \subseteq \mathbb{X}$, $\nu\,[\lambda \mapsto 0] = \{\text{if } (x \in \lambda)\ 0 \text{ else } \nu(x) \mid x \in \mathbb{X}\}$. Observe $\nu\,[\varnothing \mapsto 0] = \nu$. $\mathbb{G}(\mathbb{X})$ denotes the set of clock constraints, where a clock constraint δ takes the form:

$$\delta ::= \texttt{true} \mid x > n \mid x = n \mid x - y > n \mid x - y = n \mid \neg\delta \mid \delta_1 \wedge \delta_2 \qquad (\text{where } n \in \mathbb{N}) \tag{3.1.1}$$

We write $\nu \models \delta$ to denote that the clock valuations in ν satisfy the constraints δ. Defined formally, $(\nu \models \delta) = (\forall x \in \texttt{fn}(\delta).\delta\,[\nu(x)/x])$ to say that, for all clocks in the *free-names* of δ, we substitute the clock name with the corresponding valuation in ν. For example, $\nu[x \mapsto 3] \models (x > 2)$ holds since $(x > 2)\,[\nu(x)/x] = (3 > 2)$.

We write $\downarrow \delta$ to denote the *weak past* of constraint δ, which effectively removes any lower-bounds on clocks, only preserving the upper-bounds (formally given in Definition 3.1.1). For example, given $\delta = (3 < x < 5)$ then $\downarrow \delta = (x < 5)$, and

given $\delta = (x > 2)$ then $\downarrow \delta = \texttt{true}$. In practice, $\downarrow \delta$ allows us to reason on constraints being satisfiable at some point in the future, since we only require that the clocks' valuation does not exceed the upper-bound of the constraint.

Definition 3.1.1 (Weak Past)**.** We write $\downarrow \delta$ (the past of δ) for a constraint δ' such that $v \models \delta'$, if and only if $\exists t$ such that $v + t \models \delta$.

3.2 Syntax of TOAST

The syntax of TOAST (or just *types*) is given in Eq. (3.2.1). A type S is either a choice $\{\texttt{C}_i\}_{i \in I}$, recursive definition $\mu\alpha.S$, call α, or termination type `end`.

$$\begin{array}{lcl@{\qquad}lcl} S & ::= & \{\texttt{C}_i\}_{i \in I} \mid \mu\alpha.S \mid \alpha \mid \texttt{end} & \texttt{C} & ::= & \square\, l\, \langle T \rangle\, (\delta, \lambda).S \\ T & ::= & (\delta,\ S) \mid \texttt{Unit} \mid \texttt{Nat} \mid \texttt{Bool} \mid \texttt{String} \mid \ldots & \square & ::= & \texttt{!} \mid \texttt{?} \end{array} \tag{3.2.1}$$

Type $\{\texttt{C}_i\}_{i \in I}$ models a choice among options i ranging over a non-empty set I. Each option i is a selection/send action if $\square = \texttt{!}$, or alternatively a branching/receive action if $\square = \texttt{?}$. An option sends (resp. receives) a label l and a message of a specified data type T is delineated by $\langle \cdot \rangle$. The sending or receiving action of an option is guarded by a time constraint δ. After the action, the clocks within λ are reset to 0. Data types, ranged over by $T, T_i, \ldots$ can be higher-order types $(\delta,\ S)$ to model session delegation, `Unit` types, or *base types* (e.g., `Nat`, `Bool`, `String`). We omit a `Unit` type when the payload is nothing. Only the message label is exchanged when the data type is `None`. We discuss the structure of delegation types further in Section 3.4.1. Labels of the options in a choice are pairwise distinct.

Recursion and termination types are as standard (Bettini et al. 2008; Carbone et al. 2008; Honda et al. 2008, 2016; Scalas et al. 2019; Vasconcelos 2012; Yoshida et al. 2007).

Remark 3.2.1 (Notation)**.** One convention is to model the exchange of payloads as a separated action with respect to the communication of branching labels. In this work we follow Bocchi et al. (2015) and Yoshida et al. (2021), and model them as unique actions. When irrelevant we omit the payload, yielding a notation closer to TA. △

3.2.1 Feasibility & Junk Types

Unfortunately, when annotating session types with time constraints one may obtain protocols that are infeasible, as shown in Example 3.2.2. This is a known problem, which has been addressed by providing additional conditions or constraints on timed session types, for example compliance Bartoletti et al. (2017), feasibility Bocchi et al. (2014), interaction enabling Bocchi et al. (2015), and well-formedness Bocchi et al. (2019). (We address feasibility in Section 3.4.1.)

Example 3.2.2 (Junk Types)**.** Consider the type S defined below:

$$S = ?a\,(x > 3, \varnothing)\,.\left\{\begin{array}{l} !b\,(y = 2, \varnothing)\,.\mathtt{end}, \\ ?c\,(2 < x < 5, \varnothing)\,.\mathtt{end} \end{array}\right\}$$

Initially all clocks are 0. After a is received all clocks hold values greater than 3. Therefore, b is never able to be sent since $(y > 3)$ and, c is only *sometimes* able to be sent if a is received when $(x < 5)$, which constraint $(x > 3)$ does nothing to guarantee. Types with unsatisfiable constraints are called *junk types* in Bocchi et al. (2019). Below we show an amended S:

$$S = ?a\,(x > 3, \{x\})\,.\left\{\begin{array}{l} !b\,(y = 2, \varnothing)\,.\mathtt{end}, \\ ?c\,(2 < x < 5, \varnothing)\,.\mathtt{end} \end{array}\right\}$$

Since clock x is now reset it holds that receiving c is now an enabled action. However, sending b remains a 'junk action' that can never occur. In order to amend this we can either reset clock y after receiving a, similarly to how we enabled receiving c, or augment the time constraints on sending b to be enabled some time after $(y > 3)$. △

3.3 Semantics of TOAST

We define the semantics using three layers: configurations in Eq. (3.3.2), configurations with queues $\mathtt{M}$ that model asynchronous interactions in Eq. (3.3.3), and systems that model the parallel composition of configurations with queues in Eq. (3.3.4).

The semantics, for all layers, are defined over the transition labels ℓ given below:

$$\ell ::= \square m \mid t \mid \tau \qquad \square ::= ! \mid ? \qquad m ::= l\,\langle T \rangle \qquad \mathtt{M} ::= \varnothing \mid m;\,\mathtt{M} \tag{3.3.1}$$

where transition label ℓ can be either an interaction ($\square m$), time-step ($t \in \mathbb{R}_{\geq 0}$) or silent action ($\tau$), and m is a message and $\mathtt{M}$ is a First in First out (FIFO) message queue. Communication directions $\square$ are the same as in Eq. (3.2.1), where ! denotes a send/output action and ? a receive/input action. Hereafter, we clearly highlight relevant changes from ATST by Bocchi et al. (2019). (Note, additions are not highlighted in this way.) Eq. (3.3.1) is essentially unchanged from Bocchi et al. (2019).

Definition 3.3.1 (Future-enabled Configurations)**.** Configuration (ν, S) is *fe*, written either as '$(\nu, S) \overset{\square}{\Rightarrow}$' or '$(\nu, S)$ is $\mathtt{fe}$', iff $\exists t, m$ s.t. $(\nu, S) \xrightarrow{t\ \square m}$ via the rules in Eq. (3.3.2).

3.3.1 Configurations

A configuration $\mathbf{s}$ is a pair (ν, S). The semantics for configurations are defined by a Labeled Transition System (LTS) over configurations, the labels in Eq. (3.3.1) and the rules given in Eq. (3.3.2). A transition of the form $(\nu, S) \xrightarrow{t\ \square m} (\nu', S')$ indicates $(\nu, S) \xrightarrow{t} \mathbf{s}'' \xrightarrow{\square m} (\nu', S')$, where $\mathbf{s}''$ is some intermediate configuration.

$$\frac{\nu \models \delta_j \quad m = l_j\,\langle T_j \rangle \quad j \in I}{(\nu, \left\{\square_i\, l_i\,\langle T_i \rangle\,(\delta_i, \lambda_i).S_i\right\}_{i \in I}) \xrightarrow{\square_j m} (\nu\,[\lambda_j \mapsto 0],\, S_j)}\ [\mathtt{act}]$$

$$\frac{(\nu, S\,[\mu\alpha.S/\alpha]) \xrightarrow{\ell} (\nu', S')}{(\nu, \mu\alpha.S) \xrightarrow{\ell} (\nu', S')}\ [\mathtt{unfold}] \qquad (\nu, S) \xrightarrow{t} (\nu + t, S)\ \ [\mathtt{tick}] \tag{3.3.2}$$

By rule [act] a configuration may perform one action $j \in I$, provided the corresponding δ_j is satisfied by the current valuation of clocks ν. The resulting configuration has all clocks in λ_j reset to 0. Since Bocchi et al. (2019) did not have our single communication primitive, the corresponding behaviour was modelled using 4 rules: send, receive, branching and selection. Rule [tick] describes time passing and rule [unfold] unfolds recursive types – both are unchanged from Bocchi et al. (2019).

3.3.2 Configurations with queues

A configuration with queues $\mathbf{S}$ is a triple $(\nu, S, \mathtt{M})$, where $\mathtt{M}$ is a FIFO queue of messages which have been received but not yet processed. A queue takes the form $\mathtt{M} ::= \varnothing \mid m; \mathtt{M}$ and thus is either empty, or has a message at its head. The semantics of configurations with queues is defined by an LTS over the labels in Eq. (3.3.1) and the rules in Eq. (3.3.3). The transition $\mathbf{S} \xrightarrow{t\ \square m} \mathbf{S}'$ is defined analogously to $(\nu, S) \xrightarrow{t\ \square m} (\nu', S')$.

$$\frac{(\nu, S) \xrightarrow{!m} (\nu', S')}{(\nu, S, \mathtt{M}) \xrightarrow{!m} (\nu', S', \mathtt{M})}\ [\mathtt{send}] \qquad \frac{(\nu, S) \xrightarrow{?m} (\nu', S')}{(\nu, S, m; \mathtt{M}) \xrightarrow{\tau} (\nu', S', \mathtt{M})}\ [\mathtt{recv}]$$

$$(\nu, S, \mathtt{M}) \xrightarrow{?m} (\nu, S, \mathtt{M}; m) \quad [\mathtt{que}]$$

$$\frac{\begin{array}{cc} (\nu, S) \xrightarrow{t} (\nu', S') & \text{(configuration)} \\ ((\nu, S) \text{ is } \mathtt{fe}) \implies ((\nu', S') \text{ is } \mathtt{fe}) & \text{(persistency)} \\ \forall t' < t : (\nu + t', S, \mathtt{M}) \not\xrightarrow{\tau} & \text{(urgency)} \end{array}}{(\nu, S, \mathtt{M}) \xrightarrow{t} (\nu', S', \mathtt{M})}\ [\mathtt{time}] \tag{3.3.3}$$

Rule [send] sends a message. Message reception is handled by two rules: rule [que] inserts a message at the back of $\mathtt{M}$, and rule [recv] removes a message from the front of $\mathtt{M}$. Rule [time] is the only rule that differs from Bocchi et al. (2019) and is for time passing, which is formulated in terms of a future-enabled configuration, given in Definition 3.3.1. The second condition in the premise of rule [time] (persistency) ensures the "latest-enabled" action is never missed by advancing the clocks. Notably, this differs from the corresponding behaviour in Bocchi et al. (2019) which only required that the "latest-enabled" *send* action never be missed, and proved to be too restrictive in the presence of mixed-choice; by not allowing a participant to pass its last sending action, the latest-enabled receiving actions can never be reached (i.e., miss sending to receive a timeout). The third condition (urgency) models an urgent semantics, ensuring messages are processed as they arrive. Urgency is critical for reasoning about progress.

3.3.3 Systems

Systems are the parallel composition of two[1] configurations with queues, written as $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2)$ or $\mathbf{S}_2 \mid \mathbf{S}_2$. The semantics of systems is defined by an LTS over the labels in Eq. (3.3.1) and the transition rules in Eq. (3.3.4).

$$\frac{\mathbf{S}_1 \xrightarrow{!m} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{?m} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2'}\ [\mathtt{com\text{-}l}] \qquad \frac{\mathbf{S}_1 \xrightarrow{\tau} \mathbf{S}_1'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2}\ [\mathtt{par\text{-}l}] \qquad \frac{\mathbf{S}_1 \xrightarrow{t} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{t} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{t} \mathbf{S}_1' \mid \mathbf{S}_2'}\ [\mathtt{wait}]$$

$$\frac{\mathbf{S}_1 \xrightarrow{?m} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{!m} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2'}\ [\mathtt{com\text{-}r}] \qquad \frac{\mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1 \mid \mathbf{S}_2'}\ [\mathtt{par\text{-}r}] \tag{3.3.4}$$

Rule [com-l] handles $\mathbf{S}_1$ asynchronously sending m to the queue of $\mathbf{S}_2$. (Rule [com-r] is symmetric.) Rule [par-l] allows $\mathbf{S}_1$ to process the message at the head of $\mathtt{M}_1$ via [recv]. (Rule [par-r] is symmetric.) By rule [wait] time passes consistently across systems. These rules are essentially unchanged from Bocchi et al. (2019).

Example 3.3.2 (Unsafe Mixed-choice)**.** The use of mixed-choice in asynchronous communications may result in infeasible protocols or, more concretely, systems (or types) that get stuck. A mixed-choice is considered *unsafe* if actions of different directions compete to be performed (i.e., they are both viable at the same point in time). Consider the system $(\nu_0, S_1, \varnothing) \mid (\nu_0, S_2, \varnothing)$, where S_1 and S_2 are dual:

$$S_1 = \left\{ \begin{array}{l} ?a\,(x < 5, \varnothing)\,.\mathtt{end}, \\ !b\,(x = 0, \varnothing)\,.S_1' \end{array} \right\} \qquad S_2 = \left\{ \begin{array}{l} !a\,(y < 5, \varnothing)\,.\mathtt{end}, \\ ?b\,(y = 0, \varnothing)\,.S_2' \end{array} \right\} \tag{3.3.5}$$

In the system $\mathbf{S}_1 \mid \mathbf{S}_2$, it is possible for both $(\nu_0, S_1, \varnothing) \xrightarrow{!b} \mathbf{S}_1'$ and $(\nu_0, S_2, \varnothing) \xrightarrow{!a} \mathbf{S}_2'$ to occur at the same time. The resulting system $(\nu_0, S_1', a) \mid (\nu_0, \mathtt{end}, b)$ is unable to receive either of message a or b, and $\mathbf{S}_1'$ may be stuck waiting for interactions from $\mathbf{S}_2'$ indefinitely. Even though a deadlock can be avoided if message a is a delegation and S_1' prescribes receiving a, since message b is never received by $\mathbf{S}_2$, then Eq. (3.3.5) still violate communication safety (Dardha et al. 2017). △

[1] As we consider binary types.

3.4 Duality, Well-formedness, and Progress (of TOAST)

In the untimed scenario, the composition of a well-formed binary type with its dual characterises a *protocol*, which specifies the "correct" set of interactions between a party and its co-party. The dual of a type, formally defined below, is obtained by swapping the directions (! or ?) of each interaction:

Definition 3.4.1 (Type Duality)**.** Given a type S, we define its *dual* type $\overline{S}$ as follows:

$$\overline{\texttt{end}} = \texttt{end} \qquad \overline{\alpha} = \alpha \qquad \overline{\mu\alpha.S} = \mu\alpha.\overline{S} \qquad \overline{?} = ! \qquad \overline{!} = ?$$
$$\overline{\left\{\Box_i\, l_i\, \langle T_i\rangle\, (\delta_i, \lambda_i).S_i\right\}_{i\in I}} = \left\{\overline{\Box}_i\, l_i\langle T_i\rangle(\delta_i, \lambda_i).\overline{S}_i\right\}_{i\in I}$$

Notice that the constraints on actions are the same for a type and its corresponding dual type. In our examples, we will often change the names of the clocks used in the constraints of each type and its dual in order to emphasise that the constraints of each participant are to be satisfied by their own local set of clock valuations. For example, we say configurations $(\nu_0, !a\,(x = 3, \varnothing)\,.S')$ and $(\nu_0, ?a\,(y = 3, \varnothing)\,.\overline{S'})$ have dual types since, despite each having constraints on clocks with different names, since the context indicates the clocks within the constraints are effectively the same. Remark 3.4.3 outlines the reason for our 'simple' definition of duality in regard to recursive types, as opposed to Bono et al. (2012) and Lindley et al. (2016).

3.4.1 Well-formedness

We extend the concept of well-formedness in Bocchi et al. (2019) so that progress is guaranteed in presence of mixed-choice. The formation rules for types are given in Eq. (3.4.1). (Rules differing from Bocchi et al. (2019) are highlighted.)

Types are evaluated against judgements of the form: $A; \delta \vdash S$ where A is an environment containing recursive variables, and δ is a constraint over all clocks characterising the times in which state S can be reached.

$$\frac{\begin{array}{rl}\forall i \in I : A;\ \gamma_i \vdash S_i \ \wedge\ \delta_i\,[\lambda_i \mapsto 0] \models \gamma_i & \text{(feasibility)}\\ \forall i,j \in I : i \neq j \implies \delta_i \wedge \delta_j \models \texttt{false} \vee \square_i = \square_j & \text{(mixed-choice)}\\ \forall i \in I : T_i = (\delta',\ S') \implies \varnothing;\ \gamma' \vdash S' \ \wedge\ \delta' \models \gamma' & \text{(delegation)}\end{array}}{A;\ \downarrow \bigvee_{i\in I} \delta_i \vdash \left\{\square_i\, l_i\, \langle T_i\rangle\, (\delta_i, \lambda_i).S_i\right\}_{i\in I}}\ [\texttt{choice}]$$

$$\frac{}{A;\ \texttt{true} \vdash \texttt{end}}\ [\texttt{end}] \qquad \frac{A, \alpha : \delta;\ \delta \vdash S}{A;\ \delta \vdash \mu\alpha.S}\ [\texttt{rec}] \qquad \frac{}{A, \alpha : \delta;\ \delta \vdash \alpha}\ [\texttt{var}] \tag{3.4.1}$$

Rule [choice] checks well-formedness of choices with three conditions. The first and third conditions are from the branching and delegation rules in Bocchi et al. (2019), respectively. The second condition is new and critical to ensure progress of mixed-choice. By the first condition (feasibility) a choice is well-formed with respect to the weakest past among all options ($\downarrow \bigvee_{i\in I} \delta_i$) given that each continuation S_i is well-formed with respect to and environment γ_i which models the corresponding guard, updated with resets λ_i. By abuse of notation we write $\delta[\lambda \mapsto 0] \models \gamma$ to say $\forall \nu : (\nu \models \delta[\lambda \mapsto 0]) \implies (\nu \models \gamma)$. This ensures that in every choice, there is always at least one viable action, and it would, for example, rule out the type in Example 3.2.2. The second condition (mixed-choice) requires all actions that can happen at the same time to have the same (send/receive) direction. This condition allows for types modelling timeouts, as shown later in Example 3.4.7, and rules out scenarios as the one in Example 3.3.2. The third condition (delegation) checks for well-formedness of each delegated session with respect to their corresponding initialisation constraint δ'. Similar to the (feasibility) premise, by abuse of notation we write $\delta' \models \gamma'$ to say $\forall \nu : (\nu \models \delta') \implies (\nu \models \gamma')$.

Rule [end] ensures termination types are *always* well-formed. Rule [rec] associates variable α with an invariant δ in A. Rule [var] ensures recursive calls are defined.

Definition 3.4.2 (Well-formed Configurations)**.** Given (ν, S), S is *well-formed* against ν if, $\exists \delta$ s.t. $\varnothing; \delta \vdash S$ and $\nu \models \delta$. A type S is *well-formed* if it is *well-formed* against ν_0.

The rules in Eq. (3.4.1) check that in every reachable state (which includes every possible clock valuation) it is possible to perform the next action immediately or at

some point in the future, unless the state is final. (This is formalised as the progress property in Definition 3.4.4.) By these rules, the type in Examples 3.2.2 and 3.3.2 would not be well-formed.

Remark 3.4.3 (Delegation of Higher-Order TOAST)**.** Recall Eq. (3.2.1) in Section 3.2, where higher-order types for session delegation are defined as (δ, S). By condition (delegation) in the premise of rule [choice] in Eq. (3.4.1), for any interaction with a delegation payload $T = (\delta', S')$, delegation type S' must be *well-formed* by $\varnothing; \gamma' \vdash S'$ where environment γ' corresponds to δ' and the environment for recursive variables is empty. In short, a *well-formed* TOAST cannot delegate a recursive variable (α). Furthermore, higher-order types must be *flattened* (i.e., finitely represented) and TOAST is unable to express self-referential recursive higher-order types (e.g., $S' = \mu\alpha.!l\,\langle(\delta', S')\rangle\,(\delta, \varnothing).\alpha$). This contrasts with Bernardi et al. (2016) and Bono et al. (2012), which allow recursive variables to be delegated directly, and Lindley et al. (2016) which discusses self-referential recursive higher-order types. We feel this is orthogonal to the primary focus of this work. △

3.4.2 Progress of Types

Well-formedness, together with the persistency and receive urgency featured in rule [time] of the semantics in Eq. (3.3.3) ensures that the composition of a well-formed type S with its dual $\overline{S}$ *enjoys progress*. A system *enjoys progress* if its configurations with queues can continue communicating until reaching the end of the protocol. (Formally given below in Definition 3.4.4.)

Definition 3.4.4 (Type Progress)**.** A configuration with queues $(\nu, S, \mathtt{M})$ is *final* if both $S \overset{\text{unfold}}{\equiv} \mathtt{end}$ and $\mathtt{M} = \varnothing$. A system $\mathbf{S}_1 \mid \mathbf{S}_2$ *satisfies progress* if, for all $\mathbf{S}_1' \mid \mathbf{S}_2'$ reachable from $\mathbf{S}_1 \mid \mathbf{S}_2$ either:

1. $(\nu_1', S_1', \mathtt{M}_1')$ and $(\nu_2', S_2', \mathtt{M}_2')$ are *final*
2. or, $\exists t \in \mathbb{R}_{\geq 0}$ such that $(\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2') \xrightarrow{t\ \tau}$.

We give the definition of ($\overset{\text{unfold}}{\equiv}$) in Definition A.1.5. We write $S \overset{\text{unfold}}{\equiv} \mathtt{end}$ if S is equivalent to end, up-to the unfolding of recursive types. (e.g., $\mu\alpha_1 \ldots \mu\alpha_i.\mathtt{end} \overset{\text{unfold}}{\equiv} \mathtt{end}$.)

Theorem 3.4.5 (Progress of Systems). *If S is* well-formed *against ν_0, then:*

$$(\nu_0, S, \varnothing) \mid (\nu_0, \overline{S}, \varnothing) \text{ satisfies progress.}$$

Proof. The thesis follows immediately from Lemma A.7.3. □

The main result of this section is that for a system composed of a *well-formed* S and its dual $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, \overline{S}_2, \mathtt{M}_2)$, any state reached is either *final*, or allows for further communication – i.e., the system *satisfies progress*. Progress is critical to ensure, via type-checking, that a protocol implementation does not reach deadlock and is free from communication mismatches.[2]

The main differences from Bocchi et al. (2019) are not in the formulation of the theory (e.g., Definition 3.4.4 and the statement of Theorem 3.4.5 are basically unchanged) but in proving rule [choice] is sufficient to ensure progress of asynchronous mixed-choice. Additionally, the proof of progress in Bocchi et al. (2019) relies on a notion of persistency to a participant does not reach a point where there are no viable actions, and does so by requiring time-steps do not pass beyond the participants latest-enabled receiving action. Due to mixed-choice, it is necessary to reformulate (and relax) this condition in the semantic rule [time] in Eq. (3.3.3) to require only the latest-enabled action is not missed. Later, Example 3.4.7 discusses this further.

3.4.3 Compatibility

The proof of Theorem 3.4.5 is in Appendix A, and proceeds by showing that a property of systems called *compatibility* is preserved by transitions (given formally in Definition 3.4.6). As typical for binary systems, our proof of progress builds upon a notion of *duality* between participants, since communication is asynchronous each party may break duality with their co-party. Compatibility allows the duality of participants to be broken, only if, doing so does not violate *communication safety*. More specifically, compatibility requires that any message that arrives in a message queue is expected and therefore, able to be received and that the resulting configurations are still *compatible*. In practice, compatibility allows each party to behave independently, regardless the state of the other party, while retaining the essence of *duality*, and guaranteeing that all messages will eventually be received.

[2] A typing system for TOAST is presented in Chapter 5.

Definition 3.4.6 (Compatibility)**.** We write $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ iff $(\nu_1, S_1, \mathtt{M}_1)$ and $(\nu_2, S_2, \mathtt{M}_2)$ are *compatible*. Given $(\nu_1, S_1, \mathtt{M}_1)$ and $(\nu_2, S_2, \mathtt{M}_2)$, we define *compatibility* as the largest relation satisfying each of the following:

1. $(\mathtt{M}_1 = \varnothing)$ or $(\mathtt{M}_2 = \varnothing)$
2. $(\mathtt{M}_1 = m; \mathtt{M}_1') \implies ((\nu_1, S_1) \xrightarrow{?m} (\nu_1', S_1')$ and $(\nu_1', S_1', \mathtt{M}_1') \perp (\nu_2, S_2, \mathtt{M}_2))$
3. $(\mathtt{M}_2 = m; \mathtt{M}_2') \implies ((\nu_2, S_2) \xrightarrow{?m} (\nu_2', S_2')$ and $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2', S_2', \mathtt{M}_2'))$
4. $(\mathtt{M}_1 = \varnothing = \mathtt{M}_2) \implies (S_1 = \overline{S_2}$ and $\nu_1 = \nu_2)$

Intuitively, $\mathbf{S}_1$ and $\mathbf{S}_2$ are compatible, written $\mathbf{S}_1 \perp \mathbf{S}_2$, if: (1) at most one of their queues is non-empty (equivalent to a half duplex automaton), (2–3) a type is always able to process any message that arrives in its queue, and (4) if both queues are empty then $\mathbf{S}_1$ and $\mathbf{S}_2$ have dual types and same clock valuations.

Example 3.4.7 (Weak Persistency)**.** In the language-based approach to timed semantics by Krcál et al. (2006) (in CTA) time-passing actions are always possible, even if they bring the model into a stuck state by preventing available actions (i.e., by 'missing' or 'skipping' them). Execution traces are then filtered a posteriori, removing all 'bad' traces (defined on the basis of final states). In contrast, and to facilitate the reasoning on process behaviour, we adopt a process-based approach (as in the works of Bartoletti et al. (2018) and Bocchi et al. (2019)) that only allows for actions that characterise *intended* executions of the model. Precisely, we build on the semantics by Bartoletti et al. (2018) for asynchronous timed automata with mixed-choice, where timed actions are possible only if they do not disable: (a) the latest-enabled sending action, and (b) the (latest-enabled) receiving action if the queue is not empty. This ensures that time-passing actions preserve the viability of at least one action, i.e., *weak-persistency*. In our scenario where we have mixed-choice, constraint (a) is too strict. Consider type S and its dual $\overline{S}$ below:

$$S = \left\{ \begin{array}{l} !data\langle \mathtt{String} \rangle\, (x < 3, \varnothing)\, .S', \\ ?timeout\, (x \geq 4, \varnothing)\, .\mathtt{end} \end{array} \right\} \qquad \overline{S} = \left\{ \begin{array}{l} ?data\langle \mathtt{String} \rangle\, (y < 3, \varnothing)\, .\overline{S'}, \\ !timeout\, (y \geq 4, \varnothing)\, .\mathtt{end} \end{array} \right\}$$

According to (a), it would never be possible for S to take the timeout branch since a time action of $t \geq 3$ would disable the latest-enabled send. This is reasonable

in Bartoletti et al. (2018) because, in their general setting, there is no guarantee that a timeout will indeed be received. Unlike the work of Bartoletti et al. (2018), we can rely on duality of $\overline{S}$ and hence on the guarantee that $\overline{S}$ sends a timeout in all executions where $y \geq 4$. Our new [time] rule – condition (persistency) – implements a more general constraint than (a), requiring that one latest-enabled (send or receive) action is preserved. Constraint (b) remains to implement urgency and, for instance, prevents $\overline{S}$ from sending a timeout if a message is waiting in the queue when $y < 3$.

For example, our semantics permit the following sequence of transitions:

$$(v_0, S, \varnothing) \xrightarrow{t=2} (v[x \mapsto 2], S, \varnothing) \xrightarrow{!data\langle \texttt{String} \rangle} (v[x \mapsto 2], S', \varnothing)$$

where *data* is sent after 2 time units. Alternatively:

$$(v_0, S, \varnothing) \xrightarrow{t=4} (v[x \mapsto 4], S, \varnothing) \xrightarrow{?timeout} (v[x \mapsto 4], S, timeout; \varnothing) \overset{t'}{\nrightarrow} \qquad (t' \in \mathbb{R}_{\geq 0})$$

where after 4 time units have passed condition (urgency) of rule [time] in Eq. (3.3.3) ensures that the message (*timeout*) is received as soon as possible via rule [recv]: $(v[x \mapsto 4], S, timeout; \varnothing) \xrightarrow{\tau} (v[x \mapsto 4], \texttt{end}, \varnothing)$. △

Chapter 4

TOAST Processes

This chapter introduces the syntax and reduction rules of the process calculus of TOAST, corresponding to the types (of TOAST) presented in Chapter 3. The contents of this chapter were first shown in work by Pears et al. (2023), which has since been updated in the work by Pears et al. (2024c).

4.1 A Calculus for Processes with Timeouts

We now present a new calculus for timed processes which extends existing timed, asynchronous session calculi (Bocchi et al. 2015, 2019) with timeouts and time-sensitive conditional statements. Timeouts are defined on receive actions and may be immediately followed by sending actions, hence providing an instance of mixed-choice – which is normally not supported. Time-sensitive conditional statements (i.e., `if-then-else` with conditions on process timers) provide a natural counter-part to the timeout construct and enhance the expressiveness of the typing system in ATST by Bocchi et al. (2019). To better align processes with TOAST (types), send and select actions have been streamlined by each message consisting of both a label l and either some message variable v or value $\mathtt{v}$, which is either data or a delegated session – the same holds for receive/branch actions.

Processes are defined by the grammar given in Eq. (4.1.1). Participants are the *endpoints* of session and are denoted by p and q. Within a (binary) session, endpoints p and q communicate over channels pq and qp, where p sends to q over pq, and q sends to p over qp. We also use a and b as ad-hoc roles, along with their respective endpoints ab and ba, when discussing multiple binary sessions.

$$
\begin{array}{lll}
P, Q ::= \texttt{set}\,\textcircled{x}.P & \mid & (\nu pq)\ P \\
\mid\ p \lhd l\,(w).P & \mid & P \mid Q \\
\mid\ p^{e} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} & \mid & \texttt{0} \\
\mid\ p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}\ \texttt{after}\ Q & \mid & pq : h \\
\mid\ \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q & w ::= v \mid \texttt{v} & \\
\mid\ \texttt{delay}\,(d)\,.P & e ::= \diamond n \mid \infty \mid \leq 0 \qquad (n \in \mathbb{R}_{>0}) & \\
\mid\ \texttt{delay}\,(t)\,.P & \diamond ::= \ < \mid \leq & \\
\mid\ \texttt{def}\ X(\vec{v};\vec{r}) = P\ \texttt{in}\ Q & c ::= \textcircled{x} \diamond n \mid n \diamond \textcircled{x} \mid \textcircled{x} = n & \\
\mid\ X\langle \vec{v};\vec{r} \rangle & d ::= \texttt{t} \diamond n \mid n \diamond \texttt{t} \mid \texttt{true} & \\
 & h ::= \ \varnothing \mid h \cdot l\texttt{v} &
\end{array}
\tag{4.1.1}
$$

Processes are equipped with timers that can be set and reset during the process execution. Let $\mathbb{T}$ be the set of timers denoted by $\textcircled{x}$, $\textcircled{y}$ and $\textcircled{z}$. While *clocks* are used in types to express the time constraints of a protocol (i.e., a type), *timers* are used by processes to acquire awareness of relative time-passing during execution – such as timer primitives in Go and Erlang. Process $\texttt{set}\,\textcircled{x}.P$ creates a process timer named $\textcircled{x}$, initialises it to 0 and continues as P. If a process timer named $\textcircled{x}$ already exists then it is reset to 0. Note, there is **no correspondence** between the **clocks** used in the constraints of types and **process timers**, regardless of any matching names/labels.

Process $p \lhd l\,(w).P$ is the select/send process: it selects label l and sends payload w (which may be either a variable v or value v) to endpoint pq, and continues as P. Its counterpart is the branch/receive process $p^{e} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}$ which upon receiving one of the labels l_i, instantiates variable v_i with the received payload and continues as P_i. Parameter e is a deadline for the receive action. It can be either $\diamond n$, ∞ or ≤ 0. If e is $\diamond n$ then n is the upper bound of the receive action and $\diamond$ specifies whether the wait duration is exclusive $(<)$ or inclusive $(\leq)$. For example, setting e to (≤ 3) specifies waiting *up-to and including* 3 time units, while (< 3) specifies waiting *strictly less than* 3 time units. Setting e to (≤ 0) models a non-blocking receive action on branches that expect messages to be immediately available to receive. If e is ∞, then the receive action will block until a message is received, waiting potentially forever.

Process $p^{\diamond n} \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I}$ `after` Q is the timeout process, an extended version of the branch/receive process, where $\diamond n$ specifies the receive deadline, after which the timeout process Q gets triggered (and is then taken). We specify timeouts to be $(\diamond n)$ to tactically ensure timeouts cannot be ∞, which would make Q dead code and would be equivalent to a branch/receive action without a timeout. For simplicity, in receive actions and timeouts we omit: (a) brackets in the case of a single option (i.e., $|I| = 1$) and (b) message variables (v) for options with no payloads.

Process `if` c `then` P `else` Q is a conditional statement. The condition c is a constraint on process timers, and is notably simpler than those constraints found in Eq. (3.1.1) for types. Process `delay` $(d)\,.P$ models a non-deterministic delay with a duration d, which can either be an upper-bound ($\mathtt{t} \diamond n$), a lower-bound ($n \diamond \mathtt{t}$) or forever (`true`). The symbol `t` is used as a placeholder for an actual duration of time t, which is realised at runtime. As we shall see later in Section 4.2, at runtime, process `delay` $(d)\,.P$ is reduced to process `delay` $(t)\,.P$, for some $t \models d\,[t/\mathtt{t}]$ (e.g., if $d = \mathtt{t} \leq 5$ then t must be valued such that d is satisfied, i.e., be between 0 and 5).

Recursive processes are defined by a process variable X and parameters $\vec{v}$ and $\vec{r}$, containing *base type* values and session channels, respectively. As standard (Bettini et al. 2008; Bocchi et al. 2014; Honda et al. 2008; Vasconcelos 2012; Yoshida et al. 2007) the process calculus allows parallel processes $P \mid Q$ and scoped processes $(\nu pq)\ P$ between endpoints p and q. The end process is `0`. Endpoints communicate over pairs of channels pq and qp, each with their own unbounded FIFO buffers h, which store labelled-value payloads $l\mathtt{v}$.

4.1.1 Well-formed Processes

We have adopted the simplifying assumption in Bocchi et al. (2019) that sessions are already instantiated. Therefore, rather than relying on reduction rules to produce correct session instantiation, we rely on a syntactic well-formedness assumption. A *well-formed process* P consists of sessions of the form $(\nu pq)\ P' \mid Q \mid qp : h \mid pq : h'$ and can be checked syntactically by function $\mathtt{wf}(P)$ given in Definition 4.1.1, which extends the one by Bocchi et al. (2019) for ATST. Hereafter, we assume that any given process is *well-formed*, unless stated otherwise.

Definition 4.1.1 (Well-formed Process)**.** The function $\texttt{wf}(P)$ is defined inductively as:

$$\texttt{wf}(P) = \begin{cases} \texttt{true} & \text{if } P \in \left\{ \texttt{0},\ X\langle \vec{v}; \vec{r} \rangle,\ qp : h \right\} \\ \texttt{wf}(P') \wedge (\texttt{fq}(P') = \{pq, qp\}) & \text{if } P = (\nu pq)\ P' \\ \texttt{wf}(P') \wedge (\texttt{fq}(P') = \varnothing) & \text{if } P \in \left\{ \begin{array}{l} p \lhd l\,(w).P',\ \texttt{set}\,\textcircled{x}.P', \\ \texttt{delay}\,(d)\,.P',\ \texttt{delay}\,(t)\,.P' \end{array} \right\} \\ \texttt{wf}(P') \wedge \texttt{wf}(Q) & \text{if } P \in \left\{ \begin{array}{l} \texttt{if}\ c\ \texttt{then}\ P'\ \texttt{else}\ Q, \\ \texttt{def}\ X(\vec{v}; \vec{r}) = P'\ \texttt{in}\ Q \end{array} \right\} \\ \texttt{wf}(P') \wedge \texttt{wf}(Q) \wedge (\texttt{ft}(P') \cap \texttt{ft}(Q) = \varnothing) & \text{if } P = P' \mid Q \\ \bigwedge_{i \in I} \texttt{wf}(P_i) \wedge (\texttt{fq}(P_i) = \varnothing) & \text{if } P = p^{e} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \\ \texttt{wf}(p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}) \wedge \texttt{wf}(Q) & \text{if } P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{after}\ Q \end{cases}$$

where function $\texttt{fq}(P)$ returns the set of queues present in process P that are *not* contained within the respective scope of their corresponding session, and function $\texttt{ft}(P)$ returns the set of process timers used in process P.[1] By Definition 4.1.1, a *well-formed* process must: (a) not contain any *free queues* other than those belonging to ongoing (binary) sessions contained within *scoped processes* $(\nu pq)\ P$, which should each contain the two queues necessary for participants to exchange messages, and (b) not contain any parallel processes that make use of the *same* process timer (i.e., process timers must only be used by a single process). We also assume that a *well-formed* process has no *free names* (i.e., all variables are bound).

Remark 4.1.2 (Payloads, Variables & Values)**.** In the syntax of TOAST processes in Eq. (4.1.1), processes may send payloads w (which can be either variables v or values $\texttt{v}$), while receive into variables v, and queues may only contain value $\texttt{v}$. For the case of receive processes, the reason is simple: a variable v is used to store a value $\texttt{v}$ received by the process. Additionally, since we assume processes to be *well-formed*, and all variables are bound, then it follows that any variable v denoted to be sent my a process will have been instantiated with some value $\texttt{v}$ prior to being sent. △

[1] We relegate the definitions of $\texttt{fq}(P)$ and $\texttt{ft}(P)$ to Definitions B.1.1 and B.1.2.

Definition 4.1.3 (Structural Congruence)**.** We define $P \equiv Q$ as the smallest relation on processes:

$$P \mid \mathtt{0} \equiv P \qquad (P \mid Q) \equiv (Q \mid P) \qquad (P_1 \mid P_2) \mid Q \equiv (P_1 \mid Q) \mid P_2 \qquad P \mid Q \equiv Q \mid P$$
$$(\nu pq)\, \mathtt{0} \equiv \mathtt{0} \qquad \mathtt{delay}\,(0)\,.P \equiv P \qquad (\nu pq)\ (\nu ab)\ P \equiv (\nu ab)\ (\nu pq)\ P$$
$$(\nu pq)\ P \equiv (\nu qp)\ P \qquad (\nu pq)\ P \mid Q \equiv (\nu pq)\ (P \mid Q) \quad \text{if } pq \notin \mathtt{fn}(Q)$$
$$(\mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ P') \mid Q \equiv \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ (P' \mid Q) \quad \text{if } X\langle\vec{v'};\vec{r'}\rangle \notin \mathtt{fpv}(Q)$$

where functions $\mathtt{fn}(Q)$ and $\mathtt{fpv}(Q)$ return the set of free names and process variables in Q, respectively. (Formally defined in Definitions B.1.3 and B.1.4.)

4.2 Process Reduction

The semantics of processes are given in categorical groups in Eqs. (4.2.1 to 4.2.4) (and are restated together in Figure C.3), as a reduction relation on pairs of the form (θ, P) (i.e., a processes P with an environment θ that maps timers to their values in the current state). We formally define the *timer environment* θ in Definition 4.2.1.

Definition 4.2.1 (Timer Environment)**.** Recall $\mathbb{T}$ is the set of timers, ranged over by $\textcircled{x}$, $\textcircled{y}$ and $\textcircled{z}$. A *timer environment* θ is a linear map $\theta : \mathbb{T} \mapsto \mathbb{R}_{\geq 0}$, from timers to valuations, defined: $\theta ::= \varnothing \mid \theta,\ \textcircled{x} : n$ where $n \in \mathbb{R}_{\geq 0}$.

We define $\theta + t = \{\textcircled{x} \mapsto \theta(\textcircled{x}) + t \mid \textcircled{x} \in \mathbb{T}\}$ so that time may pass over θ and the timers contained within. We define $\theta\,[\textcircled{x} \mapsto 0]$ to be the map $\theta\,[\textcircled{x} \mapsto 0]\,(\textcircled{y}) =$ if $(\textcircled{x} = \textcircled{y})$ 0 else $\theta(\textcircled{y})$. We write $\mathrm{Dom}(\theta)$ for the domain of θ, and θ_1, θ_2 for $\theta_1 \cup \theta_2$ when $\mathrm{Dom}(\theta_1) \cap \mathrm{Dom}(\theta_2) = \varnothing$. We write $\theta \models c$ to denote that θ satisfies c, formally: $(\theta \models c) = (\forall \textcircled{x} \in \mathtt{fn}(c).c\,[^{\theta(\textcircled{x})}/_{\textcircled{x}}])$. (This is similar to the definition of $\nu \models \delta$.)

4.2.1 Reduction Rules

The reduction relation is defined on two kinds of reduction: instantaneous communication actions ($\rightharpoonup$), and time-consuming actions ($\rightsquigarrow$). We write $\longrightarrow$ to denote a reduction that is either by ($\rightharpoonup$) or ($\rightsquigarrow$). The reduction rules are categorically grouped: standard rules in Eq. (4.2.1), communication rules in Eq. (4.2.2), recursion rules in Eq. (4.2.3), and time-sensitive rules in Eq. (4.2.4).

Standard Rules, Eq. (4.2.2)

Rules [Str], [Scope], [Par-L] and [Par-R] are as standard following the work of Honda et al. (2008, 2016), Scalas et al. (2019), and Vasconcelos (2012).

$$\frac{P \equiv P' \quad (\theta,\, P') \longrightarrow (\theta',\, Q') \quad Q \equiv Q'}{(\theta, P) \longrightarrow (\theta',\, Q)}\ [\texttt{Str}]$$

$$\frac{(\theta,\, P) \rightharpoonup (\theta',\, P')}{(\theta,\, (\nu pq)\ P) \rightharpoonup (\theta',\, (\nu pq)\ P')}\ [\texttt{Scope}]$$

$$\frac{(\theta_1,\, P) \rightharpoonup (\theta_1',\, P')}{(\theta_1, \theta_2,\, P \mid Q) \rightharpoonup (\theta_1', \theta_2,\, P' \mid Q)}\ [\texttt{Par-L}]$$

$$\frac{(\theta_2,\, Q) \rightharpoonup (\theta_2',\, Q')}{(\theta_1, \theta_2,\, P \mid Q) \rightharpoonup (\theta_1, \theta_2',\, P \mid Q')}\ [\texttt{Par-R}] \tag{4.2.1}$$

Rule [Str] is for structural congruence (formally defined in Definition 4.1.3) and is the only rule that applies to both instantaneous ($\rightharpoonup$) and time-consuming ($\rightsquigarrow$) reductions. Crucially, rule [Str] **does not** allow one kind of reduction to be turned into another, since (as with all the other rules) the arrows of the conclusion and premise must match, i.e., if the premise makes a $\rightsquigarrow$ transition then so must the conclusion, and vice versa.

Communication Rules, Eq. (4.2.2)

Rules [Send] and [Recv] are standard, as in the π-calculus by Milner (1999).

$$(\theta,\, p \lhd l\,(\mathtt{v}).P \mid pq : h) \rightharpoonup (\theta,\, P \mid pq : h \cdot l\mathtt{v}) \quad [\texttt{Send}]$$

$$\frac{j \in I \quad l = l_j}{(\theta,\, p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \mid qp : l\mathtt{v} \cdot h) \rightharpoonup (\theta,\, P_j\,[\mathtt{v}/v_j] \mid qp : h)}\ [\texttt{Recv}] \tag{4.2.2}$$

$$\frac{j \in I \quad l = l_j}{(\theta,\, p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{ after } Q \mid qp : l\mathtt{v} \cdot h) \rightharpoonup (\theta,\, P_j\,[\mathtt{v}/v_j] \mid qp : h)}\ [\texttt{Recv-T}]$$

By rule [Send] a process inserts a message $l\mathtt{v}$ into the queue of the other party, while by rule [Recv] a process may remove an expected message from their queue and

continue on the corresponding branch, with any occurrences of variable v_j replaced with value $\mathtt{v}$ (recall Remark 4.1.2). Rule [Recv-T] is similar to rule [Recv] except it allows the timeout branch Q to be taken after $\diamond n$ passes without receiving a message.[2]

Recursion Rules, Eq. (4.2.3)

Both rules [Def] and [Call] are unchanged from those of ATST by Bocchi et al. (2019).

$$\frac{(\theta,\, Q) \;\rightharpoonup\; (\theta',\, Q')}{(\theta,\, \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q) \;\rightharpoonup\; (\theta',\, \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q')}\ [\mathtt{Def}]$$

$$(\theta,\, \mathtt{def}\ X(\vec{v'};\vec{r'}) = P\ \mathtt{in}\ X\langle\vec{v};\vec{r}\rangle \mid Q) \;\rightharpoonup\; (\theta,\, \mathtt{def}\ X(\vec{v'};\vec{r'}) = P\ \mathtt{in}\ P\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q)\ [\mathtt{Call}] \tag{4.2.3}$$

Rule [Def] defines a recursive process P and allows the recursive body Q to reduce. Rule [Call] handles recursive calls of predefined recursive processes and yields a process configuration consisting of the next iteration of the recursive process using the parameters provided by the process variable X.

Time-sensitive Rules, Eq. (4.2.4)

Rule [Reset] resets a process timer $\textcircled{x}$ in the timer environment θ to 0. Rule [If-T] selects branch P if time-sensitive condition c holds for θ. Rule [If-F] is symmetric, selecting Q if c does not hold for θ. Rule [Det] determines the exact duration t of a non-deterministic runtime delay modelled by d (by replacing placeholder $\mathtt{t}$ with t).

$$(\theta, \textcircled{x} : n,\, \mathtt{set}\ \textcircled{x}.P) \rightharpoonup (\theta, \textcircled{x} : 0,\, P)\ [\mathtt{Reset}]$$

$$\frac{\theta \models c}{(\theta,\, \mathtt{if}\ c\ \mathtt{then}\ P\ \mathtt{else}\ Q) \;\rightharpoonup\; (\theta,\, P)}\ [\mathtt{If\text{-}T}]$$

$$\frac{\theta \not\models c}{(\theta,\, \mathtt{if}\ c\ \mathtt{then}\ P\ \mathtt{else}\ Q) \;\rightharpoonup\; (\theta,\, Q)}\ [\mathtt{If\text{-}F}] \tag{4.2.4}$$

$$\frac{t \models d\,[t/\mathtt{t}]}{(\theta,\, \mathtt{delay}\,(d).P) \;\rightharpoonup\; (\theta,\, \mathtt{delay}\,(t).P)}\ [\mathtt{Det}] \qquad (\theta, P) \rightsquigarrow (\theta + t,\, \Phi_t(P))\ [\mathtt{Delay}]$$

[2] We include rule [Recv-T] for consistency, but hereafter only refer to rule [Recv].

Rule [Delay] outsources time-passing to function $\Phi_t(P)$ which returns process P after the elapsing of t units of time, and updates timers within θ accordingly. We immediately explore the definition of function $\Phi_t(P)$ in Section 4.2.2.

4.2.2 Time Passing

Definition 4.2.2 (Time Passing Function)**.** The time-passing function $\Phi_t(P)$ is a partial function only defined for the cases below:

$$\Phi_t\left(p^{\diamond n} \rhd \{\mathtt{B}_i\}_{i\in I}\ \mathtt{after}\ Q\right) = \begin{cases} p^{\diamond n-t} \rhd \{\mathtt{B}_i\}_{i\in I}\ \mathtt{after}\ Q & \text{if } (t \diamond n) \\ \Phi_{t-n}(Q) & \text{otherwise} \end{cases}$$

$$\Phi_t\left(p^{e} \rhd \{\mathtt{B}_i\}_{i\in I}\right) = \begin{cases} p^{e} \rhd \{\mathtt{B}_i\}_{i\in I} & \text{if } e = \infty \\ p^{\diamond n-t} \rhd \{\mathtt{B}_i\}_{i\in I} & \text{if } e = \diamond n \text{ and } (t \diamond n) \end{cases}$$

$$\Phi_t\left(\mathtt{delay}\,(t').P\right) = \begin{cases} \mathtt{delay}\,(t'-t).P & \text{if } t' \geq t \\ \Phi_{t-t'}(P) & \text{otherwise} \end{cases}$$

$$\Phi_t(P_1 \mid P_2) = \Phi_t(P_1) \mid \Phi_t(P_2) \quad \text{if } (\mathtt{Wait}(P_i) \cap \mathtt{NEQ}(P_j) = \varnothing) \text{ and } i \neq j \in \{1,2\}$$

$$\Phi_t\left(\mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q\right) = \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ \Phi_t(Q) \qquad \Phi_0(P) = P$$

$$\Phi_t\left((\nu pq)\ P\right) = (\nu pq)\ \Phi_t(P) \qquad \Phi_t(pq : h) = pq : h \qquad \Phi_t(\mathtt{0}) = \mathtt{0}$$

The first two cases in Definition 4.2.2 model the effect of time passing on branching and timeout processes. The third case is for time-consuming processes. The fourth case distributes time passing in parallel compositions and ensures that time passes for all parts of the system equally. The remaining cases define the processes where time is allowed to pass, and denote that a time passing of 0 is the same as no time passing at all (i.e., $\Phi_0(P) = P$). The auxiliary functions $\mathtt{Wait}(P)$ and $\mathtt{NEQ}(P)$ are given in Figure 4.1. Together, they indicate which role (if any) is able to receive, and are used in Definition 4.2.2 to enforce *receive urgency*, by requiring that there are no processes that are both waiting to receive and have a non-empty queue.

$$\texttt{Wait}(P) = \begin{cases} \{p\} & \text{if } P \in \left\{ \begin{array}{l} p^{\diamond n} \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{ after } Q, \\ p^{e} \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I} \end{array} \right\} \\ \texttt{Wait}(Q) \setminus \{p, q\} & \text{if } P = (\nu pq)\ Q \\ \texttt{Wait}(Q) & \text{if } P = \texttt{def } X(\vec{v}; \vec{r}) = P' \texttt{ in } Q \\ \texttt{Wait}(P') \cup \texttt{Wait}(Q) & \text{if } P = P' \mid Q \\ \varnothing & \text{otherwise} \end{cases}$$

(A) Definition of $\texttt{Wait}(P)$.

$$\texttt{NEQ}(P) = \begin{cases} \{p\} & \text{if } P = qp : h \wedge h \neq \varnothing \\ \texttt{NEQ}(Q) \setminus \{p, q\} & \text{if } P = (\nu pq)\ Q \\ \texttt{NEQ}(Q) & \text{if } P = \texttt{def } X(\vec{v}; \vec{r}) = P' \texttt{ in } Q \\ \texttt{NEQ}(P') \cup \texttt{NEQ}(Q) & \text{if } P = P' \mid Q \\ \varnothing & \text{otherwise} \end{cases}$$

(B) Definition of $\texttt{NEQ}(P)$.

FIGURE 4.1: Definition of $\texttt{Wait}(P)$ and $\texttt{NEQ}(P)$.

Informally, $\texttt{Wait}(P)$ returns the set of channels on which P is waiting to receive a message, and $\texttt{NEQ}(P)$ returns the set of endpoints with a non-empty inbound queue.

Partial Time Passing

In Definition 4.2.2 we define $\Phi_t(P)$ as a *partial function* in order to: (a) ensure that certain processes are always non-blocking and can never be arbitrarily delayed (e.g., send processes) and (b) make it easier to differentiate between a process that can not be delayed, and a process that can be delayed for 0. Definition 4.2.2 enforces that certain processes, such as send processes, should never be delayed under any circumstances. (The only exception being a formally defined `delay` process prefixed to a send process.) However, if $\Phi_t(P)$ were to be defined for all cases of P including send processes, the reduction of our example process P becomes unclear and may be indefinitely reduced via rule [`Delay`] for a delay of 0. Furthermore, by the current definition of structural congruence in Definition 4.1.3, sending *data* may be skipped altogether. Therefore, we tactically define $\Phi_t(P)$ to be only a partial function in order to distinctly group delayable and non-delayable processes. For instance, processes that are not allowed to be delayed are send processes, time-sensitive conditional statements and *undetermined* non-deterministic delays.

Send Processes In the case of send processes, consider the process:

$$P = \texttt{delay}\,(5)\,.P' \mid p \lhd data.P''$$

Under the current definition of $\Phi_t(P)$, it is clear that the *data* must be sent prior to any delay occurring and, in this instance, Definition 4.2.2 enforces that $\Phi_t(P)$ is **not defined**, not even for $t = 0$, as send processes must **not** be delayed. Rather, the sending of *data* must occur before any delay can occur, and therefore $\Phi_t(\texttt{delay}\,(5)\,.P' \mid P'')$ *may* be defined for $t = 5$, iff $\Phi_5(P'')$ is also defined.

Time-sensitive Conditional Statements As their name indicates, these processes may yield different outcomes at different points in time. Rules [If-T] and [If-F] in Eq. (4.2.4) show how the outcome of the reduction depends entirely on whether the valuation of process timers within θ satisfy some condition c. As with send processes, this behaviour needs to happen *urgently*, since if we allowed them to be arbitrarily delayed they would become practically useless.

Non-deterministic Delays Similarly to the other cases, we simply require that the corresponding instantaneous reduction occurs *urgently* via rule [Det] in Eq. (4.2.4), which will determine the exact duration of the yet-to-be-determined non-deterministic delay. Once the delay has been determined, then can a delay occur.

Example 4.2.3 (Time Passing over Timeout)**.** Consider the process below:

$$P = (\nu pq)\; p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{after}\; Q \mid qp : \varnothing \mid Q'$$

For a time-consuming action of t to occur on P it is required that $\Phi_t(P)$ is defined for all parallel components in P. Suppose $t = n + 1$ so that we can observe the expiring of the timeout. The evaluation of $\Phi_t(p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{after}\; Q)$ results in the evaluation of $\Phi_1(Q)$. If Q were e.g., a sending process, then $\Phi_{t'}(Q)$ is not defined for any $t > 0$. Otherwise, if $\Phi_1(Q)$ and $\Phi_t(Q')$ are defined, then t is allowed to pass and: $\Phi_t(P) = (\nu pq)\,\texttt{0} \mid qp : \varnothing \mid \Phi_t(Q)$. △

4.3 Expressiveness

In this section we reflect on the expressiveness of our mixed-choice extension, particularly in regard to ATST by Bocchi et al. (2019), using examples to illustrate differences. Given the increase in expressiveness, we discuss how type-checking becomes more interesting with the inclusion of `receive-after` and give an intuition on the relationship between the TOAST types and processes.

4.3.1 Missing deadlines

The process corresponding to $?a\,(\texttt{true}, \varnothing)\,.S$ is merely $p^{\infty} \triangleright a : P$, which waits to receive a forever. By way of contrast, $?b\,(x < 3, \varnothing)\,.S'$, cannot receive when $x \geq 3$, requiring the process to take the form: $p^{<3} \triangleright b : P'$. More generally, if an action is enabled when $x \geq 3$: $\{?b\,(x < 3,\ \varnothing).S',\ ?c\,(3 < x < 5,\ \varnothing).S''\}$ then, amending the previous process, $p^{<3} \triangleright b : P'$ `after` $p^{<2} \triangleright c : P''$.

4.3.2 Ping-pong protocol

The example in this section illustrates the usefulness of time-sensitive conditional statements. The ping-pong protocol consists of two participants exchanging messages between themselves on receipt of a message from the other (as featured in the work of Lagaillardie et al. (2022)). One interpretation of the protocol is the following:

$$\mu\alpha.\left\{\begin{array}{l} !ping(x \leq 3,\ \{x\}).\left\{?ping(x \leq 3,\ \{x\}).\alpha,\ ?pong(x > 3,\ \{x\}).\alpha\right\} \\ !pong(x > 3,\ \{x\}).\left\{?ping(x \leq 3,\ \{x\}).\alpha,\ ?pong(x > 3,\ \{x\}).\alpha\right\} \end{array}\right\}$$

where each participant exchanges the role of sender, either sending *ping* early, or *pong* late. Without time-sensitive conditional statements, the setting of ATST by Bocchi et al. (2019) only allows implementations where the choice between the '*ping*' and the '*pong*' branch are hard-coded. In presence of non-deterministic delays (e.g., `delay` $(t < 6)$), the hard-coded choice can only be for the latest branch to 'expire', and the highlighted fragment of the ping-pong protocol above could be naively

implemented as follows (omitting Q for simplicity):

$$\texttt{def}\ X(p) = \texttt{delay}\,(t < 6)\,.p \lhd \mathit{pong}.Q\ \texttt{in}\ X\langle r\rangle$$

The choice of sending *ping* is *always* discarded as it may be unsatisfied in *some* executions. The calculus for TOAST presented in Eq. (4.1.1) allows us to *potentially* honour each branch, thanks to the time-awareness deriving from a process timer $\textcircled{x}$:

$$\texttt{def}\ X(p) = \left(\begin{array}{r}\texttt{set}\,\textcircled{x}.\texttt{delay}\,(t < 6)\,.\texttt{if}\big(\textcircled{x} \leq 3\big)\ \texttt{then}\ p \lhd \mathit{ping}.Q \\ \texttt{else}\ p \lhd \mathit{pong}.Q'\end{array}\right)\ \texttt{in}\ X\langle r\rangle$$

4.3.3 Mixed-choice Ping-pong protocol

An alternative interpretation of the ping-pong protocol can result in an implementation with mixed-choice, as shown below:

$$\mu\alpha.\left\{\begin{array}{l}?\mathit{ping}(x \leq 3,\ \{x\}).\left\{!\mathit{pong}(x \leq 3,\ \{x\}).\alpha,\ ?\mathit{timeout}(x > 3,\ \varnothing).\mathit{end}\right\} \\ !\mathit{pong}(x > 3,\ \{x\}).\left\{?\mathit{ping}(x \leq 3,\ \{x\}).\alpha,\ !\mathit{timeout}(x > 3,\ \varnothing).\mathit{end}\right\}\end{array}\right\}$$

where '*pings*' are responded by '*pongs*' and vice versa.

Notice that if a timely *ping* is not received, a *pong* is sent instead, which if not responded to by a *ping*, triggers a timeout. Similarly, once a *ping* has been received, a *pong* must be sent on time to avoid a timeout. Such a convoluted protocol can be fully implemented with the TOAST process $\texttt{def}\ X(p) = P\ \texttt{in}\ X\langle r\rangle$ where P:

$$\begin{array}{r}P = p^{\leq 3} \rhd \mathit{ping} : \texttt{set}\,\textcircled{x}.\texttt{delay}\,(t \leq 4)\,.\texttt{if}\big(\textcircled{x} \leq 3\big)\ \texttt{then}\ p \lhd \mathit{pong}.X\langle p\rangle \\ \texttt{else}\ p^{\infty} \rhd \mathit{timeout} : \texttt{0}\end{array}$$

$$\texttt{after}\ p \lhd \mathit{pong}.p^{\leq 3} \rhd \mathit{ping} : X\langle p\rangle$$

$$\texttt{after}\ p \lhd \mathit{timeout}.\texttt{0}$$

4.3.4 Message throttling

A real-world application of the previous example is *message throttling*. The rationale behind message throttling is to cull unresponsive processes, which do not keep up with the message processing tempo set by the system. This avoids a server from

becoming overwhelmed by a flood of incoming messages. In such a protocol, upon receiving a message, a participant is permitted a grace period to respond before receiving another. The grace period is specified as a number of unacknowledged messages, rather than a period of time. Below we present a fully parametric implementation of this behaviour, where m is the maximum number of unacknowledged messages before a timeout is issued.

$$\begin{aligned} S_0 &= \mu\alpha^0.!msg(x \geq 3, \{x\}).S_1 \\ S_i &= \mu\alpha^i.\left\{?ack(x < 3, \{x\}).\alpha^{i-1},\ !msg(x \geq 3, \{x\}).S_{i+1}\right\} \\ S_m &= \left\{?ack(x < 3, \{x\}).\alpha^{m-1},\ !tout(x \geq 3,\ \varnothing).\mathtt{end}\right\} \end{aligned} \tag{4.3.1}$$

which has the corresponding processes:

$$\begin{aligned} \mathtt{def}\ X_0(p) &= p \lhd msg.X_1\langle p\rangle \\ \mathtt{def}\ X_i(p) &= p^{<3} \rhd ack : X_{i-1}\langle p\rangle\ \mathtt{after}\ p \lhd msg.X_{i+1}\langle p\rangle \\ \mathtt{def}\ X_m(p) &= p^{<3} \rhd ack : X_{m-1}\langle p\rangle\ \mathtt{after}\ p \lhd tout.\mathtt{0} \end{aligned} \tag{4.3.2}$$

We now concretize this example with the $m = 2$ instance for type S and process P:

$$S = \mu\alpha^0.!msg(x \geq 3, \{x\}).\mu\alpha^1.\left\{\begin{array}{l} ?ack(x < 3, \{x\}).\alpha^0, \\ !msg(x \geq 3, \{x\}).\left\{\begin{array}{l} ?ack(x < 3, \{x\}).\alpha^1, \\ !tout(x \geq 3, \varnothing).\mathtt{end} \end{array}\right. \end{array}\right\}$$

$$\begin{aligned} P = \mathtt{def}\ X_0(p) = p \lhd msg.\mathtt{def}\ X_1(p) = p^{<3} \rhd ack : X_0\langle p\rangle \\ \mathtt{after}\ p \lhd msg.p^{<3} \rhd ack : X_1\langle p\rangle \\ \mathtt{after}\ p \lhd tout.\mathtt{0} \end{aligned}$$

The system shown in Figure 4.2 also illustrates the instance of $m = 2$. △

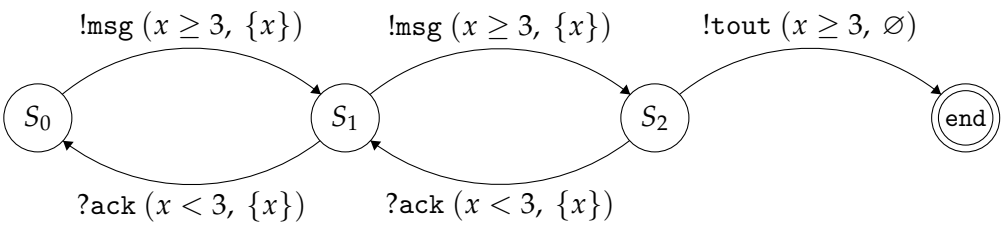


FIGURE 4.2: Message throttling protocol for $m = 2$.

Chapter 5

A Typing System for TOAST

This chapter provides a typing system for the TOAST types presented in Chapter 3 and the corresponding processes presented in Chapter 4. The contributions of this chapter were first presented in Pears et al. (2024b) [v1] – which is an extended version of Pears et al. (2023) – and has since been updated in Pears et al. (2024c) (which is still currently under review).

5.1 TOAST Typing Judgements

Processes are validated against types using judgements of the form: $\Gamma; \theta \vdash P \blacktriangleright \Delta$.

$$\begin{aligned} \Gamma &::= \varnothing \mid \Gamma,\ v : T \mid \Gamma,\ X\colon (\vec{T}; \boldsymbol{\theta}; \boldsymbol{\Delta}) \\ \theta &::= \varnothing \mid \theta,\ \textcircled{x} : n \qquad (\text{recall Definition 4.2.1}, n \in \mathbb{R}_{\geq 0}) \\ \Delta &::= \varnothing \mid \Delta,\ p : (\nu, S) \mid \Delta,\ qp : \mathtt{M} \end{aligned} \tag{5.1.1}$$

Variable Environments Γ map values v to data types T and process variables X to triples $X\colon (\vec{T}; \boldsymbol{\theta}; \boldsymbol{\Delta})$, where $\vec{T}$ is a vector of messages (consisting of labels and data types), and $\boldsymbol{\theta}$ and $\boldsymbol{\Delta}$ are sets of timer environments and session environments, respectively and both are possibly infinite. With respect to usual formulations of session types, we use the mapping θ of timers (denoted $\textcircled{x}$, $\textcircled{y}$ and $\textcircled{z}$) as a virtual state to correctly type-check time-sensitive conditional statements. As usual, *Session Environments* Δ map roles p and q to configurations and session endpoints qp and pq to queues $\mathtt{M}$. Similarly to the syntax of processes in Section 4.1, we may use a and b as ad-hoc roles, along with endpoints ab and ba, during discussions of more than one session.

5.1.1 Auxiliary Definitions & Notation

We write $\mathrm{Dom}(\Delta)$ for the domain of Δ, and Δ_1, Δ_2 for $\Delta_1 \cup \Delta_2$ when $\mathrm{Dom}(\Delta_1) \cap \mathrm{Dom}(\Delta_2) = \varnothing$. Lifting directly from Bocchi et al. (2019), we define $\Delta(p) = (\nu, S)$ iff $p \in \Delta$ and $\exists \Delta'$ such that $\Delta = \Delta', p : (\nu, S)$, and define $\mathtt{val}(p) = \nu$ an $\mathtt{type}(p) = S$.

Given a time offset $t \in \mathbb{R}_{\geq 0}$, we define $p + t = (\mathtt{val}(p) + t, \mathtt{type}(p))$ and we define $\Delta + t = \{p + t \mid p \in \mathrm{Dom}(\Delta)\}$. Additionally, recall Definition 4.2.1, where we defined: $\theta + t = \{\textcircled{x} \mapsto \theta(\textcircled{x}) + t \mid \textcircled{x} \in \mathbb{T}\}$. We say that Δ is t-reading if within the duration t, there exist roles within Δ that are able to perform a receiving action.

Definition 5.1.1 (t-reading Δ)**.** Δ is t-reading if, $\exists t' < t, p \in \mathrm{Dom}(\Delta)$ such that $\Delta(p) = (\nu, S) \implies (\nu + t', S) \xrightarrow{?m}$. We write Δ is $\diamond t$-reading if, $\exists t' \diamond t, p \in \mathrm{Dom}(\Delta)$ such that $\Delta(p) = (\nu, S) \implies (\nu + t', S) \xrightarrow{?m}$.

We say that Δ is *not* t-reading if there are no roles within Δ able to receive within the duration of t. Definition 5.1.1 is useful to check against violations of *receive urgency*, as we shall see later in Example 5.3.2. Finally, we extend the definition of well-formed configurations (Definition 3.4.2) to session environments in the obvious way:

Definition 5.1.2 (Well-formed Δ)**.** Δ is *well-formed* if, for all $p \in \mathrm{Dom}(\Delta)$ such that $\Delta = \Delta', p : (\nu, S) \implies (\nu, S)$ is *well-formed* by Definition 3.4.2.

5.2 Type Checking Rules

The typing rules are given in categorical groups in Eqs. (5.2.1 to 5.2.6) (and are restated together in Figures C.4 and C.5). We shall now discuss each rule in turn.

5.2.1 Standard Rules, Eq. (5.2.1)

Terminated processes and empty channel buffers are typed using axioms [`End`] and [`Empty`], respectively. Rule [`Weak`] handles the type-checking of a session where an individual process has successfully reached termination, and where the rest of the session may be ongoing. Such an instance is to be expected in the asynchronous setting, where one party may perform a sequence of non-blocking sending actions and finish, leaving the other party to process and receive. Rule [`Par`] divides the

session environment into two disjoint environments, Δ_1 and Δ_2, so that each session is carried by only one of P and Q. The timer environment is also split into two disjoint environments, θ_1 and θ_2, as processes would otherwise be capable of indirectly communicating or influencing the behaviour of other processes outside of the interactions prescribed by the types.

$$\Gamma;\theta \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}] \qquad \Gamma;\theta \vdash pq : \varnothing \blacktriangleright pq : \varnothing \quad [\mathtt{Empty}]$$

$$\frac{\Gamma;\theta \vdash P \blacktriangleright \Delta}{\Gamma;\theta \vdash P \blacktriangleright \Delta,\ p : (\nu, \mathtt{end})}\ [\mathtt{Weak}] \qquad \frac{\Gamma;\theta_1 \vdash P \blacktriangleright \Delta_1 \quad \Gamma;\theta_2 \vdash Q \blacktriangleright \Delta_2}{\Gamma;\theta_1,\theta_2 \vdash P \mid Q \blacktriangleright \Delta_1,\Delta_2}\ [\mathtt{Par}]$$

$$\frac{\begin{array}{c}\Gamma;\theta \vdash P \blacktriangleright \Delta,\ p : (\nu_1, S_1),\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2 \\ (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2) \qquad \forall i \in \{1,2\}.\, S_i \text{ well-formed against } \nu_i\end{array}}{\Gamma;\theta \vdash (\nu pq)\ P \blacktriangleright \Delta}\ [\mathtt{Res}]$$

$$\frac{\begin{array}{c}\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta} : \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta});\ \theta' \vdash P \blacktriangleright \vec{r} : (\vec{v},\ \vec{S}) \\ \Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta});\ \theta \vdash Q \blacktriangleright \Delta\end{array}}{\Gamma;\theta \vdash \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q \blacktriangleright \Delta}\ [\mathtt{Rec}]$$

$$\frac{(\vec{v},\vec{S}) \in \boldsymbol{\Delta} \quad \theta \in \boldsymbol{\theta} \quad \forall i : \Gamma \vdash \vec{v}_i : \vec{T}_i}{\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta});\ \theta \vdash X\langle\vec{v};\vec{r}\rangle \blacktriangleright \vec{r} : (\vec{v},\vec{S})}\ [\mathtt{Var}] \tag{5.2.1}$$

Rule [Res] ensures scoped processes are well-typed against binary sessions composed of compatible configurations (by Definition 3.4.6). Notably, the only change from Bocchi et al. (2019) in this rule is the well-formedness requirement of S_i against ν_i (see Definition 3.4.2). In Bocchi et al. (2019), well-formedness was naturally entailed by the premises of their rules type-checking sending/receiving processes. In order to tackle the complexity of checking mixed-choice (branching and selections), we have used multiple simpler rules that make it difficult to encapsulate such a requirement. Therefore, without loss of expressiveness, we have made the requirement explicit.

Rules [Rec] and [Var] are essentially unchanged from Bocchi et al. (2019), only extended to accommodate process timers. Rule [Rec] handles recursive definitions by first type-checking the recursive body P, and collects all messages ($\vec{T}$), timers ($\boldsymbol{\theta}$) and channels ($\boldsymbol{\Delta}$) necessary for P to be *well-typed* within the triple $X : (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta})$.

Since this triple contains everything necessary for P to be *well-typed*, whenever we encounter a recursive call within Q we simply need to check (via rule [Var]) that the required messages, timers and channels are provided to enable the recursive call to occur. To this end, rule [Var] requires that a given recursive call is already defined, and that the provided messages, channels and timers satisfy the requirements for the corresponding process to the recursive call to be *well-typed*.

5.2.2 Time-sensitive Rules, Eq. (5.2.2)

Rule [Timer] handles setting or resetting of timer $\textcircled{x}$. We require $\textcircled{x}$ to be already within θ in case of setting a timer (also handled by this rule) in order to avoid race conditions on timers with concurrent sessions, which would compromise subject reduction. Inspection of the syntactic structure of a process is enough to know beforehand which timers will be used. Rule [Del-d] checks that P is well-typed against all solutions of d. Rule [Del-t] reflects time passing on θ and the session Δ. The right-hand side premise of 'Δ not t-reading' enforces *receive urgency* in processes by requiring that no participants in Δ are able to receive a message from their queue for the duration of t, i.e., inverse to Definition 5.1.1.

$$\frac{\Gamma;\theta,\textcircled{x}:0 \vdash P \blacktriangleright \Delta}{\Gamma;\theta,\textcircled{x}:n \vdash \texttt{set}\,\textcircled{x}.P \blacktriangleright \Delta}\ [\texttt{Timer}]$$

$$\frac{\forall t \in d : \Gamma;\theta \vdash \texttt{delay}\,(t).P \blacktriangleright \Delta}{\Gamma;\theta \vdash \texttt{delay}\,(d).P \blacktriangleright \Delta}\ [\texttt{Del-}d]$$

$$\frac{\Gamma;\theta+t \vdash P \blacktriangleright \Delta+t \quad \Delta \text{ not } t\text{-reading}}{\Gamma;\theta \vdash \texttt{delay}\,(t).P \blacktriangleright \Delta}\ [\texttt{Del-}t] \tag{5.2.2}$$

$$\frac{\theta \models c \quad \Gamma;\theta \vdash P \blacktriangleright \Delta}{\Gamma;\theta \vdash \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q \blacktriangleright \Delta}\ [\texttt{IfTrue}]$$

$$\frac{\theta \not\models c \quad \Gamma;\theta \vdash Q \blacktriangleright \Delta}{\Gamma;\theta \vdash \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q \blacktriangleright \Delta}\ [\texttt{IfFalse}]$$

Rules [IfTrue] and [IfFalse] are for typing time-sensitive conditional statements. If the timers in θ satisfy the condition c, then by rule [IfTrue] the evaluation continues with P. Else, by the symmetric rule [IfFalse] the evaluation continues with Q.

5.2.3 Sending Rules, Eq. (5.2.3)

Rules [VSend] and [DSend] are for typing processes that send messages against a choice type, by finding a matching pair of labels in the process and type. Rule [VSend] applies for sending base-types, requiring that payload w (which is either a variable v or value $\mathtt{v}$) is well-typed against T, and the continuation process P is well-typed against the continuation S, with any clock resets applied. Rule [DSend] is similar, but for delegating an ongoing session with role b to role p, and then removes role b from the session environment.

$$\frac{\begin{array}{c}\exists i \in I : (l = l_i) \wedge (\nu \models \delta_i) \wedge (T_i \text{ base-type}) \wedge (\Gamma \vdash w : T_i) \wedge \\ (\square_i = !) \wedge \Gamma; \theta \vdash P \blacktriangleright \Delta, p : (\nu\,[\lambda_i \mapsto 0]\,, S_i)\end{array}}{\Gamma; \theta \vdash p \lhd l\,(w).P \blacktriangleright \Delta,\ p : (\nu,\ \left\{\square_i\, l_i\,\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I})}\ [\mathtt{VSend}] \tag{5.2.3}$$

$$\frac{\begin{array}{c}\exists i \in I : (l = l_i) \wedge (\nu \models \delta_i) \wedge (T_i = (\delta',\ S')) \wedge (\nu' \models \delta') \wedge \\ (\square_i = !) \wedge \Gamma; \theta \vdash P \blacktriangleright \Delta, p : (\nu\,[\lambda_i \mapsto 0]\,, S_i)\end{array}}{\Gamma; \theta \vdash p \lhd l\,(b).P \blacktriangleright \Delta,\ p : (\nu,\ \left\{\square_i\, l_i\,\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I}),\ b : (\nu', S')}\ [\mathtt{DSend}]$$

5.2.4 Branching Rules, Eq. (5.2.4)

Rule [Branch] checks a branching process against a choice type and is comprised of a single premise. The first line of the premise requires that either the type or the processes has more than one element, to ensure that type-checking is deterministic (if both are singleton sets, then either rule [DRecv] or rule [VRecv] in Eq. (5.2.5) should be used). The second (and third) line of the premise requires that for each *enabled* action in the type, the action must be receiving and there must exist a corresponding branch in the receiving process such that has matching labels and is well-typed (which is

going to be by either rule [VRecv] or rule [DRecv] in Eq. (5.2.5)).

$$\frac{\begin{array}{c}\neg(|J| = |I| = 1) \\ \forall j \in J : (\nu \models \delta_j) \implies (\Box_j = ?) \wedge \exists i \in I : (l_i = l_j) \wedge \\ \Gamma; \theta \vdash p^e \triangleright l_i(v_i) : P_i \blacktriangleright \Delta,\ p : (\nu, \Box_j\, l_j \langle T_j \rangle (\delta_j, \lambda_j).S_j)\end{array}}{\Gamma; \theta \vdash p^e \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I} \blacktriangleright \Delta,\ p : (\nu, \left\{ \Box_j\, l_j \langle T_j \rangle (\delta_j, \lambda_j).S_j \right\}_{j \in J})} \text{[Branch]} \tag{5.2.4}$$

$$\frac{\begin{array}{c}\Gamma; \theta \vdash p^{\diamond n} \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I} \blacktriangleright \Delta,\ p : (\nu, \left\{ \mathtt{C}_j \right\}_{j \in J}) \\ \Gamma; \theta + n \vdash Q \blacktriangleright \Delta + n,\ p : (\nu + n, \left\{ \mathtt{C}_j \right\}_{j \in J})\end{array}}{\Gamma; \theta \vdash p^{\diamond n} \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{ after } Q \blacktriangleright \Delta,\ p : (\nu, \left\{ \mathtt{C}_j \right\}_{j \in J})} \text{[Timeout]}$$

Rule [Timeout] checks that a `receive-after` process is well-typed against a choice type, decomposing the checking in two parts: that of the receiving process, and that of the process Q to execute in case of timeout, and requiring that both are well-typed. The first line in the premise forms a judgement to be checked by rule [Branch] by discarding the 'timeout' part of the original process. The second line of the premise checks the process is still well-typed in the instance that the timeout 'triggers' (i.e., time n passes). The second line of the premise checks that the process is still well-typed in the instance that the 'timeout triggers' (i.e., a duration of $\diamond n$ passes). Put simply, the second premise requires that all receptions specified by the type to be currently viable must have a corresponding branch in the process that, when paired together, form a judgement that is well-typed.

5.2.5 Reception Rules, Eq. (5.2.5)

Rule [VRecv] is applied when evaluating individual base-type receptions. The first line of the premise checks that waiting the duration of n adheres to the timings specified by the type's constraints δ, this line also requires that there are no other roles in Δ that have enabled receiving actions. The second line of the premise evaluates the proceeding process P against the continuation type S, given any possible delay that could occur ($t \diamond n$). Rule [DRecv] is similar, but for evaluating individual *delegation*

receptions, which are then added to the session environment.

$$\frac{\begin{array}{c} T \text{ base-type} \quad \Delta \text{ not } e\text{-reading} \quad \forall t : (\nu + t \models \delta) \implies (t \in e) \\ \forall t \in e : \Gamma, v : T;\ \theta + t \vdash P \blacktriangleright \Delta + t,\ p : (\nu + t\,[\lambda \mapsto 0]\,, S) \end{array}}{\Gamma; \theta \vdash p^e \triangleright l(v) : P \blacktriangleright \Delta,\ p : (\nu, ?l\langle T\rangle\,(\delta, \{\lambda\})\,.S)}\ [\texttt{VRecv}]$$

$$\frac{\begin{array}{c} T = (\delta',\ S') \quad \nu' \models \delta' \quad \Delta \text{ not } e\text{-reading} \quad \forall t : (\nu + t \models \delta) \implies (t \in e) \\ \forall t \in e : \Gamma; \theta + t \vdash P \blacktriangleright \Delta + t,\ p : (\nu + t\,[\lambda \mapsto 0]\,, S),\ q : (\nu', S') \end{array}}{\Gamma; \theta \vdash p^e \triangleright l(q) : P \blacktriangleright \Delta,\ p : (\nu, ?l\langle T\rangle\,(\delta, \{\lambda\})\,.S)}\ [\texttt{DRecv}] \tag{5.2.5}$$

5.2.6 Queue Rules, Eq. (5.2.6)

Non-empty buffers and queues are typed by [VQue] or rule [DQue]. The former is for base-type payloads and the latter for delegated sessions.

$$\frac{T \text{ base-type} \quad \Gamma \vdash \mathtt{v} : T \quad \Gamma; \theta \vdash qp : h \blacktriangleright \Delta,\ qp : \mathtt{M}}{\Gamma; \theta \vdash qp : l\mathtt{v} \cdot h \blacktriangleright \Delta,\ qp : l\,\langle T\rangle; \mathtt{M}}\ [\texttt{VQue}]$$

$$\frac{T = (\delta,\ S) \quad \nu \models \delta \quad \Gamma; \theta \vdash qp : h \blacktriangleright \Delta,\ qp : \mathtt{M}}{\Gamma; \theta \vdash qp : lq \cdot h \blacktriangleright \Delta,\ qp : l\,\langle T\rangle; \mathtt{M},\ q : (\nu, S)}\ [\texttt{DQue}] \tag{5.2.6}$$

5.3 Type Checking Examples

The handling of scope restriction, parallel composition, and recursion is similar to existing formulations. We give an example of the less obvious rules to handle timers and mixed-choice. Precisely, we show in Example 5.3.1 how to type-check an implementation of a simplified (non-recursive) version of the mixed-choice ping-pong protocol featured in Section 4.3.3. Later, in Example 5.3.2 we illustrate the issues that can arise from type-checking multiple binary sessions with *incompatible interleavings*.

Example 5.3.1 (Type Checking Ping-pong)**.** Consider the protocol S below:

$$S = \left\{ \begin{array}{l} ?ping(x \leq 3,\ \{x\}).\left\{ !pong(x \leq 3,\ \{x\}).\texttt{end},\ ?timeout(x > 3,\ \varnothing).\texttt{end} \right\} \\ \\ !pong(x > 3,\ \{x\}).\left\{ ?ping(x \leq 3,\ \{x\}).\texttt{end},\ !timeout(x > 3,\ \varnothing).\texttt{end} \right\} \end{array} \right\} \tag{5.3.1}$$

and its candidate implementations P:

$$\begin{aligned} P = p^{\leq 3} \triangleright ping : \texttt{set}\,ⓩ.\texttt{delay}\,(t \leq 4)\,.\texttt{if}\big(ⓩ \leq 3\big)\ \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \\ \texttt{after}\ p \triangleleft pong.p^{\leq 3} \triangleright ping : \texttt{0} \\ \texttt{after}\ p \triangleleft timeout.\texttt{0} \end{aligned} \tag{5.3.2}$$

We will now discuss how our type system can check $\varnothing; ⓩ : 0 \vdash P \blacktriangleright\ p : (\nu_0, S)$. From here, one can apply rule [Timeout], which decomposes the judgement into two: (i) one for branching process given in Eq. (5.3.3), and (ii) another for the timeout continuation Q given in Eq. (5.3.4).

$$\varnothing; ⓩ : 0 \vdash p^{\leq 3} \triangleright ping : \texttt{set}\,ⓩ.\texttt{delay}\,(t \leq 4)\,.\texttt{if}\,(ⓩ \leq 3) \ldots \blacktriangleright\ p : (\nu_0, S) \tag{5.3.3}$$

$$\varnothing; ⓩ : 3 \vdash p \triangleleft pong.R \blacktriangleright\ p : (\nu_0[x \mapsto 3], S) \tag{5.3.4}$$

where $R = p^{\leq 3} \triangleright ping : \texttt{0}\ \texttt{after}\ p \triangleleft timeout.\texttt{0}$. In Eq. (5.3.4), note that the values of process timer ⓩ and virtual clock x have incremented by the duration of the timeout, i.e., 3. Following up Eq. (5.3.3), one can apply rule [Branch], which checks that all viable actions in S are receiving actions, and that each is correctly implemented in the process being typed. Clearly this holds, since the only viable receive action in S is $?ping\,(x \leq 3, \{x\})$, which matches the only branch $p^{\leq 3} \triangleright ping : \ldots$ in the process. We next apply rule [VRecv], yielding:

$$\varnothing; ⓩ : n \vdash \texttt{set}\,ⓩ.\texttt{delay}\,(t \leq 4)\,.\texttt{if}\,(ⓩ \leq 3) \ldots \blacktriangleright\ p : (\nu_0,\ S') \qquad (t \leq 3) \tag{5.3.5}$$

with $n \leq 3$ and $S' = \{!pong(x \leq 3,\ \{x\}).\texttt{end},\ ?timeout(x > 3,\ \varnothing).\texttt{end}\}$. Since clock x has been reset to 0, and is x the only clock used by our type S, the set of

clock valuations is back to ν_0. Application of [Set] then resets timer Ⓩ to 0. Next, [Del-d] decomposes the judgement in Eq. (5.3.5) into a set of judgements, one for each solution t' of $d = (t \leq 4)$. Below follows the premise of [Del-t]:

$$\varnothing; Ⓩ : t' \vdash \texttt{if}\ (Ⓩ \leq 3)\ \texttt{then}\ p \lhd \mathit{pong}.\mathbf{0}\ \texttt{else}\ p^{\infty} \rhd \mathit{timeout} : \mathbf{0} \blacktriangleright\ p : (\nu_0,\ S') \quad (5.3.6)$$

Most notably, if t' is smaller than or equal to 3 (i.e., $t' \leq 3$) we can apply rule [IfTrue]:

$$\varnothing; Ⓩ : 3 \vdash\ p \lhd \mathit{pong}.\mathbf{0} \blacktriangleright\ p : (\nu_0[x \mapsto 3],\ S') \quad (5.3.7)$$

and if t' is greater than 3 (e.g., $t' = 4$) we can apply rule [IfFalse]:

$$\varnothing; Ⓩ : 4 \vdash\ p^{\infty} \rhd \mathit{timeout} : \mathbf{0} \blacktriangleright\ p : (\nu_0[x \mapsto 4],\ S') \quad (5.3.8)$$

To proceed the evaluation of Eq. (5.3.7), observe that the process is selecting a label *pong*. By rule [VSend], we only need to check that S' has a sending branch with label *pong* viable when $(x = 3)$, which is indeed the case (i.e., $!\mathit{pong}(x \leq 3,\ \{x\}).\mathbf{end}$). Similarly, looking at Eq. (5.3.8), we need to use the rule branching to isolate all (one in this case) the receive actions viable at the current time. The remaining of both derivations is straightforward. The case for the 'after' process in Eq. (5.3.4) is similar, except it starts with a shifted time in the timers and virtual clocks.

We show the full derivation tree in Appendix C.3. △

Example 5.3.2 (Incompatible Interleavings)**.** Consider the two following Δ's:

$$\begin{aligned} \Delta_{ab} &= a : (\nu_1, S_1),\ b : (\nu_2, S_2),\ ba : \mathtt{M}_1,\ ab : \mathtt{M}_2 \\ \Delta_{pq} &= p : (\nu_3, S_3),\ q : (\nu_4, S_4),\ qp : \mathtt{M}_3,\ pq : \mathtt{M}_4 \end{aligned}$$

where both Δ_{ab} and Δ_{pq} are *well-formed* and composed together, under the session environment $\Delta = (\Delta_{ab}, \Delta_{pq})$. Both p and b are roles performed by the same participant in two distinct sessions. All clocks within Δ are 0. Below we define S_2 and S_3:

$$S_2 = \left\{ \begin{array}{l} ?\mathit{request}\langle \mathtt{String} \rangle\ (x < 5, \{x\})\ .S_2', \\ !\mathit{timeout}\ (x \geq 5, \{x\})\ .S_2'' \end{array} \right\} \qquad S_3 = ?\mathit{message}\ (y > 3, \varnothing)\ .S_3'$$

Consider the following implementation P, where role b first waits 5 time units for a *request* before receiving *message* as role p.

$$P = ab^{<5} \triangleright request(\texttt{req\_str}) : qp^{\infty} \triangleright message : P'$$
$$\texttt{after}\ ba \triangleleft timeout.qp^{\infty} \triangleright message : P''$$

Assume *message* arrives in the queue qp at time 3.5 and *request* arrives at time 4. Process P will receive *request* at time 4 while *message* is still pending in the queue, breaking *receive urgency*. In fact, P is not 4-reading since there exists a role p in Δ that may be able to receive before 4 time units. Definition 5.1.1 is utilised by the typing rules in Eqs. (5.2.1 to 5.2.6) to ensure such implementations are *not* well-typed. In this case, one may observe that there exists no well-typed implementation of S_2 and S_3 that satisfies receive urgency, hence the roles are fundamentally incompatible and unable to be interleaved. △

5.3.1 Limitations

Example 5.3.3 (Cascading Receptions)**.** Consider the following type S:

$$S = \left\{ ?a\,(x < 5, \varnothing)\,.S_1, \quad ?b\,(x < 7, \varnothing)\,.S_2 \right\}$$

Type S describes behaviour where some message a may be received for the first 5 time units, and message b for the first 7 time units. Crucially, for the first 5 time units both messages a and b may be received at the same time. Such behaviour may be implemented using the following process P:

$$P = p^{<5} \triangleright \left\{ a : P_1, \quad b : P_2 \right\} \texttt{after}\ p^{<2} \triangleright \left\{ b : P_2 \right\}$$

Notice in the above that the reception of message b is split across the two receive processes, hence we say such a reception is a *cascading reception*. Unfortunately, the typing rules for TOAST given in Section 5.2 are too restrictive to be able to correctly type-check this example, i.e., the type-checking will incorrectly reject this implementation. The issue lies within the premise of rules [VRecv] and [DRecv] in Eq. (5.2.5), which requires that: $\forall t : (\nu + t \vdash \delta) \implies (t \in e)$, which (if we assume

an initial set of clocks) causes the following issue when type-checking the branch of message b in the outermost receive process: $\forall t : (\nu_0 + t \vdash x < 7) \implies (t < 5)$, which clearly does not hold; e.g., if $t = 6$ then $(6 < 7) \implies (6 < 5)$. We aim to address this in future work. One possible solution is to 'flip' the implication and instead require that: $\forall t : (t \in e) \implies (\nu + t \vdash \delta)$. A similar approach appears in the recent work of ATMP by Hou et al. (2024), which also extends the work of Bocchi et al. (2019) and relaxes the corresponding condition in a similar way. △

5.4 Subject Reduction

In this section we establish subject reduction for the typing system for TOAST presented in Eqs. (5.2.1 to 5.2.6) (and Figures C.4 and C.5). Namely, we establish a relation between well-typed processes and their types that is preserved by reduction.

As usual, subject reduction is given for closed systems (Corollary 5.4.5) with Γ and Δ empty. The proof relies on two lemmas that establish subject reduction for open systems, for time-consuming steps (Theorem 5.4.3) and instantaneous steps (Theorem 5.4.4). We give an overview in this section and relegate the full proofs to Appendix B. Crucially, by rule [Res] in Eq. (5.2.1), in any typing judgement of an open system, the sessions in Δ are both *well-formed* and comprised of compatible configurations, enabling us to utilise progress of TOAST types (Theorem 3.4.5). Such sessions we say are fully-balanced. The notion of balanced session, inspired by the one in Chen et al. (2017) is given in Definitions 5.4.1 and 5.4.2.

Definition 5.4.1 (Balanced Δ)**.** Let Bal be the set containing all session environments Δ, such that if $\Delta \in$ Bal, then Δ adheres to the following:

1. $\Delta = \Delta', p : (\nu, S), qp : m;\mathtt{M} \implies (\nu, S) \xrightarrow{?m} (\nu', S')$ and $\Delta', p : (\nu', S'), qp : \mathtt{M} \in$ Bal.
2. $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2) \implies \exists \mathtt{M}_2 : (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$.
3. $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2), pq : \mathtt{M}_2 \implies (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$.

Definition 5.4.2 (Fully-Balanced Δ)**.** A balanced Δ is said to be fully-balanced if:

1. $\Delta = \Delta',\ p : (\nu_1, S_1) \implies$
$$\exists \Delta'',\ q,\ \nu_2,\ S_2,\ \mathtt{M}_1,\ \mathtt{M}_2 \quad \text{s.t.} \quad \Delta' = \Delta'',\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2$$

2. $\Delta = \Delta',\ qp : \mathtt{M}_1 \implies$
$$\exists \Delta'',\ \nu_1,\ S_1,\ \nu_2,\ S_2,\ \mathtt{M}_2 \quad \text{s.t.} \quad \Delta' = \Delta'',\ p : (\nu_1, S_1),\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2$$

Theorem 5.4.3 (Time Step)**.** *Let Δ be fully-balanced and* well-formed. *If $\Gamma; \theta \vdash P \blacktriangleright \Delta$ and $\Phi_t(P)$ is defined, then $\Gamma; \theta + t \vdash \Phi_t(P) \blacktriangleright \Delta + t$ and $\Delta + t$ is fully-balanced and* well-formed.

By Theorem 5.4.3 we prove that, given a well-typed process P, for any value of t such that $\Phi_t(P)$ is defined by Definition 4.2.2, t passing evenly over the session environment preserves both 'well-formedness' and 'balancedness'. (Similar to the preservation of 'well-formedness' and *compatibility* for system configurations in Lemmas A.6.1 and A.6.3 of Theorem 3.4.5.) Clearly, by inspection of the typing rules in Eqs. (5.2.1 to 5.2.6), such processes would require several divisions of the initial session environment (and map of process timers) before each individual process could be evaluated. Therefore, we rely on the notion of 'balancedness' (Definition 5.4.1). Naturally, a fully-balanced session must also be balanced, and this allows us to reason on specific balanced components of a larger fully-balanced session environment.

Theorem 5.4.4 (Action Step)**.** *Let Δ be balanced and* well-formed. *If $\Gamma; \theta \vdash P \blacktriangleright \Delta$ and $(\theta, P) \rightharpoonup (\theta', P')$ then $\exists \Delta' : \Delta \longrightarrow^* \Delta'$ and $\Gamma; \theta' \vdash P' \blacktriangleright \Delta'$ and Δ' is balanced and* well-formed.

By Theorem 5.4.4 we prove, amongst other things, that any sending or receiving action performed by a well-typed process corresponds to an action prescribed by the type of the corresponding configuration, at the correct time. The best illustration of this is perhaps Case 2 in Theorem 5.4.4, which shows the case of reduction via rule [Recv] in Eq. (4.2.3), and that the reduction preserves well-typedness. We relegate the rule for session environment reduction to the appendix. (See Figure B.1.)

Corollary 5.4.5 (Subject Reduction)**.**

$$\textit{If } \varnothing;\theta \vdash P \blacktriangleright \varnothing \textit{ and } (\theta, P) \longrightarrow (\theta', P') \textit{ then } \varnothing;\theta' \vdash P' \blacktriangleright \varnothing.$$

Proof. Since $\Delta = \varnothing$, then by Definition 5.4.2, Δ is fully-balanced. We proceed by induction on the nature of the reduction ($\longrightarrow$):

Case 1. If $\longrightarrow = \rightsquigarrow$ then the thesis follows Theorem 5.4.3.

Case 2. If $\longrightarrow = \rightharpoonup$ then the thesis follows Theorem 5.4.4. □

As a safety property of our typing system, subject reduction guarantees that any action performed by a well-typed process is as prescribed by the types (i.e., is timely and correct). The full proofs of Theorems 5.4.3 and 5.4.4 are relegated to Appendices B.5.1 and B.5.2 respectively, in Appendix B.

Chapter 6

Generating Erlang Stubs with Inline TOAST Monitors

This chapter presents the contents of Pears et al. (2024a), which is written for an audience not as familiar with session types as the intended audience of Pears et al. (2023, 2024c) in Chapters 3 to 5. For this reason, in this chapter we use a streamlined set of terminology when referring to the theory of TOAST; namely, we use 'TOAST protocols' to refer to the *types* of TOAST, and 'TOAST models' refer to the collective theory of TOAST.

Additionally, throughout this chapter for the sake of visual simplicity, we avoid the use of recursive definitions and variables and instead, by abuse of notation we denote a recursive call by simply re-stating a type or process; e.g., while in *standard* TOAST we would write "$S = \mu\alpha.!\mathsf{a}(x > 1, \{x\}).\alpha$" hereafter we write $S = !\mathsf{a}(x > 1, \{x\}).S$, even though we mean the former. This change is intended to be purely aesthetic and is to simply the theory for the intended audience.

6.1 Chapter Overview

In this chapter, we build on TOAST theory (Pears et al. 2023, 2024c) to present our ongoing work towards a practical toolchain *Erlang on TOAST* (2024) for specifying and verifying asynchronous communication protocols featuring timeouts in Erlang. Our toolchain aims to (a) automate the generation of correct-by-construction program stubs with timeouts (and other mixed-choice) in Erlang from TOAST processes that implement TOAST protocols, and (b) provide a transparent inline monitoring

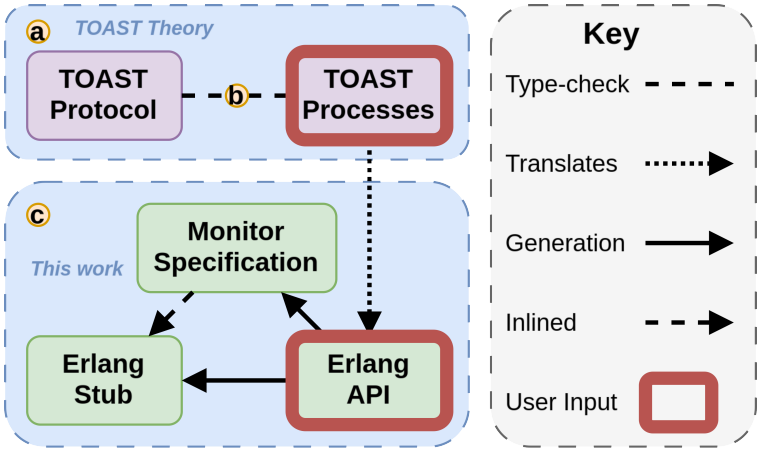


FIGURE 6.1: Erlang-on-TOAST Toolchain Overview

framework for TOAST protocols that can be integrated with Erlang supervision trees, to ensure our stubs continue to adhere to their source TOAST model once users expand upon them with functionality for their specific use case. The code generation builds upon the formal correspondence between session types and CFSM by Deniélou et al. (2012), which allows us to build the stubs incrementally, state by state.

Altogether, our toolchain ensures that the Erlang implementation and execution of a TOAST process that correctly implements a well-designed TOAST protocol is *safe* by a combination of correct-by-construction code generation and inline monitoring: the protocol is free of *stuck*-errors such as deadlocks, and all communication behaviours are either verified or enforced to comply with the protocol (dependent on configuration). We show that our toolchain can (a) facilitate the design of application-level protocols, (b) offer formal guarantees of safety, and (c) aid in the development of implementations whose behaviour is faithful and closely corresponds to the model.

6.1.1 Toolchain Overview

Shown in Figure 6.1 is an overview of the toolchain we present in this work, for generating Erlang stub programs with runtime monitors from TOAST models. Our work builds upon the theoretical work ⓐ of Pears et al. (2023, 2024c) which presented TOAST types and TOAST processes. Hereafter, we refer to TOAST types as *protocols*. An extension of Pears et al. (2023) is currently in progress (Pears et al. 2024c), which presents ⓑ, a typing system checking a TOAST process correctly implements a given TOAST protocol. Our toolchain ⓒ can take as input either a TOAST process

formulated using our Erlang API, or the TOAST process itself which our tool can automatically translate into our Erlang API. We discuss the differences between the two further in Section 6.3.

6.1.2 Outline

Section 6.2 provides an introduction to the theoretical background of TOAST. An outline of our toolchain is presented in Section 6.3, and Section 6.4 provides details on how we generate stubs with some examples, and Section 6.5 describes our runtime monitors. In Section 6.7.2 we study a use case protocol, 'Two-Factor Authentication' and discuss the difficulties that can arise when bridging the gap between designing and implementing time-sensitive protocols. Section 7.2.1 discusses related work. In Section 7.2.2 we outline future work.

6.2 Background on TOAST

Session types Bettini et al. (2008) and Honda et al. (1998, 2008) formally model the behaviour of processes in concurrent systems that interact via message-passing as part of an ongoing communication *session*. A session type specifies how interactions should occur in a session: send or receive, the message types, and their causality with other interactions. *Timed* session types Bocchi et al. (2019, 2014) further express timing constraints for the interactions. TOAST by Pears et al. (2023, 2024c) is an extension with safe mixed-choice that enables modelling behaviours previously out of reach, such as timeouts. In TOAST protocols, each interaction consists of: (i) a direction of communication, either sending (!) or receiving (?), (ii) a message label, (iii) a condition tuple of time constraints and a set of virtual clocks to reset, and (iv) are prefixed by '.' to the continuation protocol. Eventually, a protocol either reaches with an '`end`' or loops recursively (potentially indefinitely).

Example 6.2.1 (Running Example)**.** Consider the TOAST protocol S defined below:

$$S = \left\{ ?\mathtt{a}(x < 5, \varnothing).\mathtt{end}, \quad !\mathtt{b}(x = 5, \{x\}).S \right\} \tag{6.2.1}$$

As defined above in Eq. (6.2.1), S models a server that waits for a message `a` with a *timeout* of 5 time units (e.g., seconds). If message `a` is not received within 5 seconds, then S instead sends a notification `b` and retries to execute S.

Concretely, the behaviour of S as defined in Eq. (6.2.1), is a choice (curly brackets) between: (i) to receive a message `a` within 5s and then terminate, or (ii) to send a message `b` at any time after 5 seconds and continue as S (recursively).[1] Such choices are called 'mixed-choice' since they offer a 'mix' of both send and receive actions.

Time constraints are expressed on a clock x that is local to S and is initially set to 0. A clock can be reset to express relative time constraints. For instance, in S, clock x is reset after the sending of `b`. Since x is reset before each iteration of the loop, this defines the behaviour of S as repeatedly send `b` at a rate of 5s until `a` is received and then S terminates. In line with the semantics of `receive-after` in Erlang, the timeout action (`!b`) can only happen if no message matching `a` is received while $(x < 5)$. △

6.2.1 Safety, Feasibility & Stuck-Freedom

One advantage of *timed session types* is the notion of *well-formedness* which acts as a sanity check that protocols are not intrinsically flawed, namely that they are *safe* (Bocchi et al. 2019; Pears et al. 2023) – recall Definitions 3.4.2, 4.1.1 and 5.1.2. In the scope of this work, we say a protocol is *safe* if it satisfies two properties: *feasibility* and *stuck-freedom*. We shall now explore what both of these mean in our context.

Feasibility

A protocol is *feasible* if there exists an implementation that satisfies its constraints (recall Example 3.2.2). For example, in Eq. (6.2.2) the TOAST protocol S_F models a server that waits for a request that will arrive by time 5, and then replies by time 3.

$$S_F \;=\; \texttt{?request}(x \leq 5).\,\texttt{!response}(x \leq 3).\texttt{end} \tag{6.2.2}$$

In a contract between a client and server, the server is guaranteed to receive `request` by time 5, yet is required to send `response` by time 3. While there are instances

[1] Recall, this chapter uses a simplified notation for recursion from the TOAST types given in Chapter 3. In practice, we will follow the theory of TOAST, and therefore recursion is achieved via *variables* and not types themselves. (As distinguished in Remark 3.4.3.)

where S_F is able to progress and terminate successfully, the protocol is *infeasible* since, if `request` arrives at time 4, then the server is unable to fulfil its obligation of sending `response` by time 3. In order to amend the protocol in Eq. (6.2.2) one could, for example, reset x after receiving `request`, or postpone the deadline for sending `response`. (Note, for the sake of simplicity we do not consider any network latency or delays, even though they can be encoded into the model, as discussed in Section 2.2.1.) Unlike the protocol in Eq. (6.2.1), S_F does not have a 'timeout branch' specifying how the server should behave if no `request` arrives by time 5.

Stuck-Freedom

A protocol is *stuck-free* if, for all parties, the system will never reach a state where they one party is stuck waiting for a message that will never arrive. Consider the protocol S_S together with its *dual* client C_S in Eq. (6.2.3).

$$\begin{aligned} S_S &= \Big\{ ?\mathtt{a}(x \leq 5).\mathtt{end}, \quad !\mathtt{b}(x > 0, \{x\}).?\mathtt{a}(x > 0).?\mathtt{c}(x > 0).\mathtt{end} \Big\} \\ C_S &= \Big\{ !\mathtt{a}(y \leq 5).\mathtt{end}, \quad ?\mathtt{b}(y > 0, \{y\}).!\mathtt{a}(y > 0).!\mathtt{c}(y > 0).\mathtt{end} \Big\} \end{aligned} \tag{6.2.3}$$

Recall Definition 3.4.1, we say C_S and S_S are dual since they share the same interaction structure and timing constraints but have opposing communication directions (and they have independent local clocks x and y). In Eq. (6.2.3) S_S is *not* stuck-free: if after 2 seconds both parties decide to send `a` and `b`, respectively, then the client will terminate while the server will consume `a` and wait to receive `c` forever, getting *stuck*. In order to make Eq. (6.2.3) stuck-free, we must amend the constraint on exchanging `b` to occur after 5 time units. Stuck-freedom also rules out *deadlocks*, where a party gets stuck waiting to receive from the other party.

6.2.2 Safe Mixed-choice and Timeouts

The expressive power of TOAST lies within its ability to model *safe* mixed-choice, which encapsulate the behaviour of timeouts and dually, *co-timeouts* along with other behaviours. (We discuss co-timeouts in Section 6.4.6. Put briefly, while a timeout waits to receive before sending, a *co-timeout* must send within a given time range

before then waiting to receive.) A mixed-choice is an interaction pattern where a process is able to perform either a send or receive action from the same state. For example, both Eqs. (6.2.1) and (6.2.3) feature a mixed-choice between exchanging `a` or `b` from one party to another. Mixed-choice, especially when combined with asynchronous communication, have been ruled out by previous work on session types, as they may be unsafe – for reasons outline in Example 2.3.1. In fact, we observed in Eq. (6.2.3) that mixed-choice with unruly time-constraints can even lead dual participants to get stuck by becoming incompatible.

The main contribution of TOAST (Pears et al. 2023) is how to structure mixed-choice in such a way that it is *safe* and hence capable of modelling timeouts, by using a notion of *well-formedness*. Concretely, well-formedness is a decidable algorithm that, for each state, ensures: (i) there is always at least one viable future (feasibility) and, (ii) there only actions of the same direction may occur at the same time, and actions with differing directions must occur in disjoint periods of time (stuck-freedom). For example, recall protocol Eq. (6.2.3) is *not* stuck-free since there is a mixed-choice between the client sending `a` and receiving `b` at the same time, since $(x \leq 5)$ and $(y > 0)$ are both true when initially $x = y = 0$ and $(y = x) \leq 5$. By contrast, protocol Eq. (6.2.1) *is* stuck-free, since the timing constraints on sending and receiving actions have no intersection.

Timeouts in Erlang

TOAST allows us to express mixed-choice, which underpin a very common programming pattern: *timeouts*, both those that would yield a termination of a session, and those with continuations such as ones that could be expressed with the Erlang '`receive-after`' expression. More complex behaviour can also be expressed, such as more than one interleaved sending and receiving intervals (nested timeouts) and, interactions with multiple clocks (useful to model absolute and relative constraints in the same protocol). The main contribution of the (under review) extension by Pears et al. (2024c) of TOAST (Pears et al. 2023) is a typing system for checking TOAST processes adhere to TOAST protocols. Both the well-formedness and the type-checking algorithm rely on a Simple Theorem Prover (STP) Solver to reason on time constraints.

Put briefly: (i) a *well-formed* TOAST protocol is guaranteed to be free from deadlocks, communication mismatches and enjoy progress (Pears et al. 2023), and (ii) a *well-typed* TOAST process has been statically type-checked to only behave as prescribed by a TOAST protocol. In this chapter we address the problem of generating safe implementations and monitors from safe TOAST processes. We assume that a given TOAST process is *well-typed* against a *well-formed* TOAST protocol.

6.3 A TOAST-to-Erlang Toolchain

Our toolchain relies on our Erlang API, which is a DSL for defining TOAST processes which implement TOAST protocols. Following Pears et al. (2023, 2024c), our Erlang API supports the construction of protocol implementations composed of send and receive actions with time constraints, safe mixed-choice and recursion. We require the following adaptations to the time-sensitive components of TOAST in order to integrate with Erlang. Firstly, due to the behaviour of the '`receive-after`' pattern in Erlang, we can model the behaviour of *inclusive* constraints (i.e., $\geq$, $\leq$ and $=$) but not *exclusive* constraints (i.e., $>$,$<$ or $\neq$). Secondly, the timers native to Erlang count *downwards* from a set value, and upon reaching 0 send a message to a predefined process. However, the timers in TOAST processes count *upwards* from 0 when set, and their values can be used for conditional statements for time-sensitive selection. Therefore, when defining a TOAST process using our Erlang API, it is typical to define a new timer in the Erlang API for each condition imposed on a timer, as this provides the upper-bound necessary. (See Example 6.3.1.) In practice, either can be used as input since our toolchain can translate TOAST processes to our Erlang API.

Example 6.3.1 (Process Timers & Erlang API)**.** Consider the following TOAST process:

$$P = \texttt{set}(x).\texttt{delay}(t \leq 7).\texttt{if}\ (x = 0)\ \texttt{then}\ Q_1\ \texttt{else}\ \texttt{if}\ (x < 5)\ \texttt{then}\ Q_2\ \texttt{else}\ Q_3 \tag{6.3.1}$$

In Eq. (6.3.1), some TOAST process P first sets a timer x and then experiences a non-deterministic delay of up to 7 time units in duration. If there is no delay (i.e.,

$t = 0 = x$) then P proceeds as process Q_1. Else, P proceeds as Q_2 if $x < 5$ and otherwise as Q_3. Such a TOAST process translates to the following Erlang API:[2]

```
1 _P() ->{timer,"x0",1,{timer,"x5",5000,_P1()}}.
2 _P1()->{delay,7000,{if_timer,"x0",_P2(),_Q1()}}.
3 _P2()->{if_timer,"x5",_Q2(),_Q3()}.
```

where `_Q1()`, `_Q2()` and `_Q3()` are further Erlang API which correspond to Q_1, Q_2 and Q_3 in Eq. (6.3.1) respectively. Notice on line 1 of the above: (a) timer `x0` is set with duration 1, and (b) for each occurrence of x in the time constraints of the TOAST process in Eq. (6.3.1), we defined distinct timers in the Erlang API.

For both (a) and (b), consider the Erlang snippet below:

```
1 P() -> erlang:start_timer(1,self(),timer_x0),
2        erlang:start_timer(5000,self(),timer_x5),P1().
3 P1() -> timer:sleep(rand:uniform_real()*7000),
4   receive {timeout,_TID,timer_x0} -> P2() after 0 -> Q1() end.
5 P2() -> receive {timeout,_TID,timer_x5} -> Q2() after 0 -> Q3() end.
```

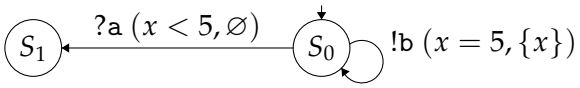
The above corresponds to Eq. (6.3.1) and the previously shown Erlang API. (Note: the above is a simplified Erlang code snippet implementing Eq. (6.3.1), and only serves to expose the approach used by our tool when generating Erlang stubs.)

As before, `P()` (lines 1–2) initialises the timers and proceeds to `P1()`. While it is possible to read the value of an Erlang timer, since an upper-bound is always required, this approach has shown to be simple and effective for our purposes.

Observe `P1()` on lines 3–4 which reads, delay for a random duration between 0 and 7 seconds, if '`timer_x0`' has completed, then proceed to `P2()`, since following Eq. (6.3.1) we should only proceed to `Q1()` if no delay has been experienced. Otherwise, in the case of a delay we proceed to `P2()`. Similarly, observe `P2()` on line 5, which more intuitively follows Eq. (6.3.1) and reads, if '`timer_x5`' has completed then proceed to `Q2()`, else `Q3()`. The use of `after 0` enables the mailbox to be checked for the message from the timer in a *non-blocking* manner. If the timer has not completed yet (i.e., less than 5 time units have passed) then we are able to proceed to perform the corresponding action in `Q3()`. Otherwise, the timer has completed and the protocol has been violated. △

[2] An overview of the Erlang API is given in full, later in Section 6.4.1.

Example 6.3.2 (Simple Timeout)**.** Recall our running example given in Example 6.2.1. Below we present the TOAST protocol in Eq. (6.2.1) as an FSM:



(6.3.2)

where $S = S_0$. Below we define an implementation using the Erlang API:

```
1 S() -> {timer,"x5",5000, {rec,"A", {act,r_a,endP, aft,"x5", {
2           act,s_b,{timer,"x5",5000, {rvar,"A"}}}}}}.
```

Observe that our Erlang API simply uses '`act`' to denote when an action should occur, and does not itself differentiate between sending and receiving actions. Instead, it is the data being 'acted upon' that denotes whether it is a send or receive action, by means of a prefix (i.e., the '`s_`' in `s_a` denotes the act of sending message `a` and vice versa, the '`r_`' in `r_b` denotes the act of receiving message `b`).

While in the theory all *clocks* are initially 0, in our Erlang API, all *timers* must be set to an initial value. Above, we set timer `x5` to 5000 milliseconds, define a recursive point `A` and then, wait to receive `a` until timer `x5` reaches 0, at which point if `a` has not been received, we then send `b`. After sending `b`, we then reset timer `x5` before then entering a tail-recursive loop back to `A`.

Alternatively, Eq. (6.2.1) can be expressed without using timers:

```
1 S() -> {rec, "A", {act, r_a, endP, aft, 5000, {act, s_b, {rvar, "A"}}}}.
```

We discuss when timers must be used in Section 6.4. △

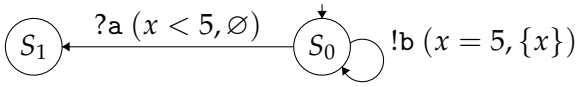
Remark 6.3.3 (Equality Constraints)**.** Recall Example 6.3.1, where for the constraint $(x = 0)$ we utilise a timer with minimal duration (1ms) in order to determine if there had been any delay at all. When implementing the behaviour of other such constraints, where for some timer y, we say $(y = n)$, then we follow similarly to Example 6.3.1, by defining two timers, one set to n in milliseconds and the other of $n + 1$ms. Note, implementing the behaviour of other $(y = n)$ constraints, we follow similarly, except that we create two timers, one of duration n in milliseconds, and another of n+1 milliseconds. △

6.3.1 From TOAST Processes to Finite-State Machines

Given a TOAST process defined using our Erlang API, our tool begins by extracting an FSM which is used internally for generating code. This is common in other works that generate code from session types following the close correspondence between session types and CFSM by Deniélou et al. (2012) – see Section 2.1.2 for a discussion of such works. Similarly to extending the protocols in Section 6.3. we have also extended the internal FSM to be able to capture the behaviour of TOAST.

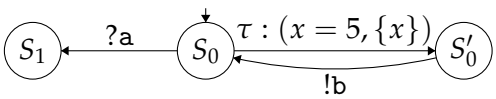
Due to the encoding of TOAST clocks and constraints using Erlang timers discussed in Section 6.3, we use an intermediate representation of FSMs that differs from the formal CTA by Krcál et al. (2006) that most directly correspond to TOAST theory. This intermediate representation is *only* used internally by the tool, for the joint purposes of generating Erlang stubs and protocol specifications used by our monitors.

The internal FSM representation is *untimed* since we do not model the behaviour of clocks. Instead, we extend the kinds of *edges* the FSM supports to encapsulate the additional information. Rather, we have simply expanded the kinds of *edges* of the FSM, and how the different kinds of outgoing edges from one state determine the nature of the state. Recall our running example, Example 6.2.1, and the FSM representation given in Example 6.3.2:



(6.3.2)

where $S = S_0$. However, upon using as input to our tool Eq. (6.2.1) written using the Erlang API (shown in Example 6.3.2), our tool instead produces the following internal representation:



(6.3.3)

As shown in Eq. (6.3.3), mixed-choice are transformed from one 'mixed' state, to a series of non-mixed-states, connected via silent actions (τ) that correspond to an interval between sending or receiving actions. The FSM representation simplifies the process of generating our stubs, by allowing each kind of edge to be built in isolation.

6.3.2 From FSMs to Erlang Stubs

Using the internal FSM extracted from a given protocol, our tool then proceeds to traverse the FSM depth-first in order to generate the stubs. Put briefly, we handle each state depending on a classification that reflects the kinds of outgoing edges it has. We build our stubs by combining snippets that correspond to both the state as a whole, and each outgoing edge. We handle recursion by moving the body of the loop into a new tail-recursive function, which is entered via a function call in the outer-scope. We discuss this further in Section 6.4.

6.3.3 From FSMs to Runtime Monitoring

Once we have obtained the internal FSM, our tool then converts it to a more accessible format that our generic monitoring program can use to either perform runtime verification of a process, or runtime enforcement. We now provide an introduction to these concepts, and then after return to discuss the TOAST monitors of our toolchain.

Brief Overview of Runtime Monitoring

Runtime monitoring (Aceto et al. 2024; Bartocci et al. 2018; Francalanza et al. 2017, 2018) is a form of dynamic verification that observes the behaviour of a program as it executes. An in-depth discussion can be found in Cassar et al. (2017). Put briefly: (1) Runtime verification monitors analyse the trace of the program as it executes, and report any violations to the system. In Erlang/OTP, this could be a supervisor, which would then deploy a predefined recovery strategy. (2) Runtime adaptation monitors also detect violations, but instead will reactively attempt to change the behaviour of the monitored process to stop the error from occurring again. (3) Runtime enforcement monitors fully mediate the communication of the monitored process, acting on their behalf and ensuring no violation occurs.[3]

TOAST Monitoring Template for Erlang

Our monitors are intended to be used within Erlang supervision trees, so that any violation of their prescribed protocol yields a measured response from the system,

[3]For more details on runtime monitoring see Aceto et al. (2024) and Francalanza et al. (2017).

as specified. Our runtime monitors are a single Erlang file containing a generic monitoring program, which is capable of monitoring the interactions of an assigned process against a protocol specification provided at runtime. The protocol specification itself is automatically generated from the Erlang API when generating the stub programs. As part of our toolchain, this protocol specification is derived from our TOAST protocols, via the internal FSM. We discuss our monitors further in Section 6.5.

6.4 Erlang Stub Generation

We explain how our toolchain generates program stubs from TOAST processes that are well-typed against a well-formed TOAST protocol. We first provide further details of our Erlang API which our tool uses as input and can be automatically translated from TOAST processes. We rely upon the fact that a TOAST process that is well-typed against a well-formed TOAST protocol, since it provides us with certain guarantees of the behaviour of the TOAST process, without having to worry about reasoning on the potentially far more complex timing constraints of TOAST protocols in Erlang. This streamlines the process of going from the theory of TOAST processes to our Erlang API, and ultimately, Erlang stubs. The mapping between TOAST processes and our Erlang API is included in the tool *Erlang on TOAST* (2024). This section provides a conceptual overview of the mapping between these two.

6.4.1 Erlang API for TOAST Processes

Below we provide an overview of the constructors provided by our Erlang API: (a) `{timer, t, n, S}` creates an Erlang timer named `t` with a duration `n` and continues to `S`. (b) `{act, a, S}` allows action `a` to be performed with continuation `S`. (c) A simple timeout: `{act, a, S1, aft, d, S2}` behaves similarly to (b) except action `a` must be performed before duration `d` elapses, where `d` is either: (i) an integer duration in milliseconds or, (ii) a *timer* `t` from (a). Output selection and input branching are specified by `{select, [{a1, S1},...]}` and `{branch, [{a1, S1},...]}` respectively. Both are capable of specifying a *timeout*, as in (c). Beyond timeouts, `{if_timer, t, S1, S2}` behaves as `S1` if timer `t` *has* completed, or as `S2` otherwise. The opposite behaviour is corresponds to `{if_not_timer, t, S1, S2}`. Recursive definitions and calls are specified

by `{rec, r, S}` and `{rvar, r}` respectively, where `r` is a recursive variable. Delays are specified by `{delay, d, S}` which behave as `S` after delay `d` has completed. Successful terminations are denoted by `endP` while protocol violations are denoted by `error`.

We now proceed to explain how behaviour specified by TOAST protocols and TOAST processes corresponds to Erlang code through examples of core communication protocol patterns and interaction structures, and aim to provide some insight into how our tool builds Erlang stubs.

6.4.2 Delays, Constraints & Errors

Consider the TOAST protocol given below, which features two interactions, each constrained by an upper and lower bound:

$$E = \left\{ ?\mathtt{a}(1 < x \leq 5, \varnothing).\mathtt{end}, \quad !\mathtt{b}(7 < x \leq 9, \varnothing).\mathtt{end} \right\} \tag{6.4.1}$$

The time constraints of E denote two intervals separated by a 2s gap: $1-5$ and $7-9$. The protocol in Eq. (6.4.1) can be written using the Erlang API as follows:

```
1 E() -> {delay, 1000, {act, r_a, endP, aft, 4000, {
2         delay, 2000, {act, s_b, endP, aft, 2000, error}}}}.
```

Above, the `delay` requires that the process waits 1 second before starting to receive `a`, matching the lower bound of constraint $(1 < x \leq 5)$ in Eq. (6.4.1). The upper bound is rendered using `aft`. Next, we model the lower-bound of sending `b` with a 2s delay. Finally, in order to capture the upper-bound of 9s for sending `b`, we introduce a timeout leading to `error` . Here, reaching error corresponds to violating the protocol, and can be handled by adding appropriate action at the level of the implementation. An alternate way to write E using the Erlang API is by using explicit timers:

```
1 E() -> {timer, "x5", 5000, {timer, "x7", 7000, {timer, "x9", 9000,
2           {delay, 1000, {act, r_a, endP, aft, "x5",
3           {delay, "x7", {act, s_b, endP, aft, "x9", error}}}}}}}.
```

While either of the above could be valid mappings of Eq. (6.4.1), in our automation of the mapping we are opting for the latter for generality, to rely on the flexibility of timers with respect to hard-coded constraints. A hard-coded value for an 'after' branch is prone to allowing a process to receive for longer than intended, in the case

where the process experiences prior delays. However, we can use timers to address this issue, since they are a separate process, they are unaffected by delays the main process experiences (intended or otherwise).

Interleaved Mixed-choice

TOAST allows for protocols where a mixed-state has multiple intervals in which send and receive action interleave. For example:

$$S_1 = \left\{ ?\mathtt{a}(x \leq 5, \varnothing).S_2, \quad !\mathtt{b}(5 < x \leq 7, \varnothing).S_3, \quad ?\mathtt{c}(x > 7, \varnothing).S_4 \right\}$$

Implementing the behaviour of such a protocol (or its dual) as a TOAST process is straightforward, by using timers combined with time-sensitive conditional statements. Such a TOAST process maps to the following Erlang API:

```
1 S1() -> {timer,"x5",5000, {timer,"x7",7000, {act,r_a,S2(), aft,"x5",
2               {if_timer,"x7", {act,s_b,S3()}, {act,r_c,S4()}}}}}.
```

Basically, `S1()` behaves as a timeout in interval $(0,5)$ and as a *co-timeout* in interval $(5,7)$ (namely as the party that has to provide a message `b` within a deadline). The input notation of `S1()` uses a conditional statement on a timer `x7` to express this situation: if you are in time then send `b` otherwise receive `c`.

Multiple clocks

Below is a protocol illustrating a recursive 'feeds-server' that uses multiple clocks.

$$S_F = \left\{ !\mathtt{stop}(20 \leq y).\mathtt{end}, \quad !\mathtt{feed}(x \leq 1 \wedge y < 20, \{x\}).S_F \right\}$$

S_F models a server that repeatedly produces feeds at the pace of 1s and stops after 20s. Section 6.4.2 corresponds to a TOAST process which maps to the Erlang API:

```
1 Sf() -> {timer,"x1",1000, {timer,"y20",20000,
2   {rec,"Sf", {if_timer,"y20",
3     {act,s_stop,endP},
4     {delay,"x1", {act, s_feed, {timer,"x1",1000, {rvar,"Sf"}}}}}}}}}.
```

TABLE 6.1: Interaction features & patterns in TOAST models.

Features & Patterns	**TOAST Protocols** (Pears et al. 2023)	**TOAST Processes** (Pears et al. 2024c)	**Erlang API**
Timeouts	✓	✓	✓
Co-timeouts	✓	✓	✓
Selection	✓	×	✓
Safe Mixed-choice	✓	✓	✓
Unsafe Mixed-choice	×	×	×
Producer-Consumer	✓	✓	✓
Diagonal Constraints	✓	◊	◊
Complex Constraints	✓	◊	◊

Key: ✓ Supported × Unsupported ◊ Indirectly supported

Above, the timer `y20` is used to track the total amount of time the feeds-server has been looping. Only once 20s have elapsed does the feeds-server send `stop`. Before `y20` reaches 0, the feeds-server sends `feed` at a rate no faster than every 1s.

6.4.3 Supported Features

Table 6.1 illustrates the descriptive capabilities of TOAST protocols, TOAST processes and our Erlang API, where (✓) denotes supported, (×) unsupported and (◊) indicates the feature or pattern is *indirectly* supported.

In Pears et al. (2024c), TOAST processes do *not* support output *selection* (i.e., the ability to *select* one sending action from several options). However, our Erlang API has not inherited this limitation and is fully capable of describing selection. Additionally, our toolchain is fully capable of generating Erlang stubs with selections and, similarly to non-blocking payloads, will automatically provide a means of making the selection in a non-blocking manner. For example, the protocol '$!\texttt{a}(x < 3, \varnothing).\texttt{end}$' can be directly expressed with a *maximal implementation* TOAST process '$\texttt{delay}(t < 3).p \triangleleft \texttt{a}.\texttt{0}$', and defined using the Erlang API '`{act, s_a, aft, 3000, error}`'. While, the protocol '$\{!\texttt{b}(\texttt{true}, \varnothing).S, !\texttt{c}(\texttt{true}, \varnothing).S'\}$' has no TOAST processes that are maximal implementations, since TOAST processes require send actions be explicitly defined.

Complex Constraints

TOAST processes and our Erlang API can *indirectly* (◊) implement diagonal constraints (which compare the difference between two clock valuations; e.g., $x - y = 2$) constraints with multiple upper-bounds and constraints with negation. Both TOAST

processes and our Erlang API do not support such constraints to be defined directly. However, since we can rely on our TOAST process being well-typed against a well-formed TOAST protocol that *does* support these constraints, and may feature them, then we know that such a TOAST process (and the corresponding Erlang API) are guaranteed to adhere to these constraints, even if they themselves do not feature them. This can either be achieved by: (a) using a *non-maximal implementation* (i.e., where not all the interactions actions are implemented into the process), or (b) making use of the time-sensitive structures in the processes, as shown in Section 6.7.2 or even (c) simplifying the timing constraints to make use of fewer process timers.

Limitations

The *Erlang on TOAST* (2024) toolchain follows the theory of TOAST by Pears et al. (2024c) and therefore, is limited to binary sessions. Additionally, in Erlang we cannot distinguish between inclusive ($\geq, \leq$) and exclusive ($>, <$) bounds featured in the timing constraints of both TOAST protocols and processes.

As discussed previously in Section 6.3 and Example 6.3.1, the timers native to Erlang count *downwards* from some upper-bound value. Unsurprisingly, this makes it less straightforward to model the timing constraints of TOAST protocols, which count *upwards* from 0. Naturally, there will be some discrepancies when it comes to the behaviour of a real-world system and an entirely theoretical model such as TOAST protocols, but we feel these are not too significant.

For instance, modelling the behaviour of an equality constraint (e.g., $(x = 1)$) is not as simple as modelling a bounded constraint (e.g., $(x \leq 1)$) when using Erlang timers, since they only indicate if a certain amount of time has passed, and do not provide any guarantees of how recently this was. In order to model the behaviour of the equality constraint $(x = 1)$, we must allow for a 1ms degree of precision and use two timers: one for 1000ms, and another for 1001ms (as in Example 6.3.1). This allows us to check if the message is received after the first timer, but before the second.

Additionally, it is difficulties such as these that arise from using the Erlang timers that chose to only *indirectly* support some of the more complex time constraints, as outlined in Table 6.1. Outside these caveats necessary for our implementation, there

are no other significant discrepancies introduced by the Erlang timers with regard to the clocks used in the theory of TOAST protocols.

6.4.4 Stub Behaviour

By default, generated stubs include a file named `stub.hrl`[4] which contains a palette of useful functions that are used by our tool when generating the stubs. We choose to move this behaviour to an Erlang header file to improve the readability and mitigate duplicated code. Each function in our stubs take two parameters: (1) the Process ID (`PID`) of the other party in the sessions (`CoParty`) and, (2) a map to store necessary data accumulated as the program runs (`Data`). When initially generated, the map `Data` is used to store the timers started and any messages received by the program, although, the user may store additional information in `Data` when they extend the stub.

6.4.5 Simple Timeout Stub

Erlang is capable of implementing the behaviour of a timeout directly, thanks to the '`receive-after`' expression. A programmer may choose to implement the behaviour of Eq. (6.2.1) from our running example (Example 6.2.1) in Erlang using:

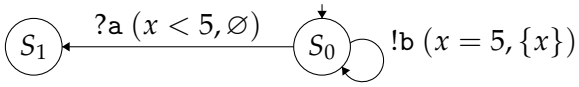
```
1 S(CoParty, Data) ->
2   receive {CoParty, a, Payload} -> exit(normal)
3   after 5000 -> CoParty ! {self(), b, some_data}, S(CoParty, Data) end.
```

Note the format of the messages exchanged: a triple comprised of the `PID`, message label and message payload. Our tool utilises the '`receive-after`' expression of Erlang, and also systematically implements functionality to aid the user when expanding the stubs, including: (1) State-by-state data isolation, providing the user with a distinct `Data` variable for each state, before any actions are performed. (2) Automatic saving of received messages to `Data`. (3) Placeholder functions for acquiring payloads to send as part of a message. Below is a sample of the kind of code generated by our tool from the protocol (with timers) of our running example in Example 6.3.2 for Eq. (6.2.1):

```
1 main(CoParty, Data) ->
2     {Data1, _TID_x5} = set_timer(x5, 5000, Data), S(CoParty, Data1).
```

[4] See https://github.com/jonahpears/Erlang-on-TOAST-sample-app/blob/main/include/stub.hrl

The function call '`set_timer(x5,5000,Data)`' on line 2 will create a new timer (via `erlang:start_timer(x5,self(),5000)`) and returns the PID of the timer process and the updated `Data` containing the timer under the name `x5`. If a timer under that name already exists, then it is cancelled and replaced with the new one (effectively acting as a reset). At the end of line 2 we enter a function `S` since the initial state is immediately re-entered after sending `b`. We restate the FSM of our running example below:



(6.3.2)

Therefore, after first initialising the relevant Erlang timers, we then enter a new scope:

```
4 S(CoParty, Data) ->
5   {ok, _TID_x5} = get_timer(x5,Data),
6   receive {CoParty,a,Payload_A} -> Data2 = save_msg(a,Payload_A,Data1),
7                                    stopping(CoParty,Data2);
8           {timeout, _TID_x5, timer_x5} ->
9               {Data2,Payload_B} = get_payload_b(Data1),
10              CoParty ! {self(), b, Payload_B},
11              {Data3,_TID_x5} = set_timer(x5,5000,Data2),
12              S(CoParty, Data3) end.
```

Following Eq. (6.2.1) or Eq. (6.3.2), the process must first wait to receive `a` for the duration of 5s. Shown above, the process first retrieves the PID for the corresponding timer (`_TID_x5`) and proceeds to wait for the duration. If `a` is received from `CoParty`, then the message is saved to `Data2` and the process terminates successfully (via `stopping(...)`). Otherwise, if timer `_TID_x5` completes, then the process: (1) sends `b` (along with some payload `Payload_B`) to `CoParty` (2) resets the timer `x5`, and, (3) loops via tail-recursion. Functions such as `get_payload_b` can introduce vulnerabilities into processes, if `get_payload_b` never completes; e.g., function `get_payload_b` may be *blocking*, but since there is no upper-bound for sending `b` this is not an issue. These vulnerabilities are somehow orthogonal to the problem space of TOAST that only guarantees stuck-freedom of the interaction structure. However, in the design of a TOAST protocol, if one foresees that some actions are likely to be diverging, a timeout should be introduced.

Remark 6.4.1 (Timer Timeouts)**.** Shown on line 9 in the code snippet above is a method for using Erlang timers as timeouts, rather than the value-based timeout of the '`receive-after`' expression. To reiterate Section 6.4.2, while both approaches are functionally equivalent, using timers as in this approach can be more robust in the presence of delays. △

6.4.6 Co-Timeouts

We will now discuss stub generation for timeouts from the perspective of the other role involved: the one that needs to provide a timely message. We will refer to these co-parties as *co-timeouts*. First, recall our running example (Example 6.2.1). Let us consider the *dual* D of the protocol in Eq. (6.2.1):

$$D = \left\{ !\mathtt{a}(y < 5).\mathtt{end}, \quad ?\mathtt{b}(y = 5, \{y\}).D \right\} \tag{6.4.2}$$

Naturally, Eq. (6.4.2) describes a client that is either able to send `a` within 5 time units, or afterwards wait to receive `b`. Receiving `b` causes clock y to be reset, and loops back to the start. The implementation of a co-timeout is less intuitive than the one of its co-party. There are three possible ways of implementing D: (a) The process that is always too late to send `a`. (b) The process that is always early and sends `a`. (c) The process that can send `a` by the deadline *sometimes*. The case of (b) effectively makes receiving `b` redundant. We now discuss how to implement cases (a) and (c).

Arriving Too Late

In the case of (a) some prior delay must have occurred, causing the upper-bound of sending `a` to be missed. Following Remark 6.4.1, we implement using timers:

```
1 D(CoParty,Data) -> TID_x5 = get_timer(x5, Data),
2   receive {timeout, TID_x5, timer_x5} ->
3     receive {CoParty,b,Payload_B} -> Data1 = save_msg(b, Payload_B, Data),
4                                      D(CoParty, Data1) end;
5   after 0 -> {Data1, Payload_A} = get_payload_a(Data),
6              CoParty ! {self(), a, Payload_A},
7              {Data2, _TID_x5} = set_timer(x5,5000,Data1),
8              D(CoParty, Data2) end.
```

Above, we implement a non-blocking `receive-after` by having a timeout of duration of 0s, ensuring that `D` must be able to receive the signal that the timer `x5` has completed *immediately*. In the case that timer `x5` has completed and `D` is able to receive `{timeout, TID_x5, timer_x5}`, then `D` begins to wait to receive `b`. Otherwise, less than 5s have passed, and `D` is able to go ahead and prepare to send `a`. What if `get_payload_a` is *unreliable* and may get stuck? Such a scenario falls under (c).

Unreliable Dependencies

In the case of (c) then there is some factor that sending `a` is dependent upon. Such as, a function for obtaining the payload to send along with `a` which is *unreliable* or in the least, not guaranteed to complete within the time frame. In our case, this is function `get_payload_a`. We need a way of calling `get_payload_a` while ensuring the main process remains non-blocked and responsive. We present `nonblocking_payload`:

```
1 nonblocking_payload(Fun, Args, PID, Timeout) when is_integer(Timeout) ->
2   spawn( fun() ->
3     Waiter = self(),
4     Timer = erlang:start_timer(Timeout,self(),nb_p),
5     TimeConsumer = spawn( fun() -> Waiter ! {self(), ok, Fun(Args)} end ),
6     receive {TimeConsumer, ok, Result} -> PID ! {self(), ok, Result};
7             {timeout, Timeout, nb_p}   -> PID ! {self(), ko},
8                                           exit(TimeConsumer, normal)
9     end end );
```

Above, `nonblocking_payload` creates a new timer for the duration of `Timeout` and spawns a process `TimeConsumer` to complete the potentially *unreliable* function `Fun(Args)`, and send the results back to itself. If `Fun(Args)` completes before `Timer`, then the `Result` is sent back to the main process via `PID` with label `ok`. Otherwise, the message `ko` is returned, signalling it took too long and in the case of Eq. (6.4.2), that `D` should begin waiting to receive `b`. Below is another clause of `nonblocking_payload` which takes a timer, reads the value and re-enters through the function shown above.

```
1 nonblocking_payload(Fun, Args, PID, Timer) when is_pid(Timer) ->
2   Value = erlang:read_timer(Timer),
3   nonblocking_payload(Fun, Args, PID, Value).
```

In practice, our tool utilises `nonblocking_payload` for the case of co-timeouts in the following way:

```
1 send_before(CoParty, {Label, {Fun, Args}}, Timeout)
2 when is_integer(Timeout) ->
3   NonBlocking = nonblocking_payload(Fun, Args, self(), Timeout),
4   receive {NonBlocking, ok, Result} -> send(CoParty, {Label, Result});
5           {NonBlocking, ko} -> ko end.
```

This snippet above outsources the potentially blocking behaviour to another process spawned within the same node, and waits to either receive `ok` and a payload `Result`, which is then sent to `CoParty`, or receive `ko` which indicates the function took too long to complete. The snippet above is utilised by our tool to generate code corresponding to co-timeouts. Alternatively, we also support timers explicitly defined by the user:

```
1 send_before(CoParty, {Label, {Fun, Args}}, Timer) when is_pid(Timer) ->
2   Value = erlang:read_timer(Timer),
3   IsKo = send_before(CoParty,{Label,{Fun,Args}},Value),
4   case IsKo of ko -> receive {timeout, Timer, _Name} -> ko end;
5                _Else -> _Else end.
```

Since the value of the `Timer` is passed through and used by the spawned process to determine if the function finishes on time, in the case that `ko` is returned, the original timer will have also reached 0, and so we remove this message from the mailbox of our main process `D`.

6.4.7 Producer-Consumer Pattern

Next, we illustrate recursion using the Producer-Consumer pattern. For brevity, we only show the behaviour of the *producer*.

$$\begin{aligned} P_S &= \texttt{?start}(\texttt{true}, \{x\}).P'_S \\ P'_S &= \left\{ \texttt{?stop}(x \leq 1).\texttt{end}, \quad \texttt{!data}(x > 1, \{x\}).P'_S \right\} \end{aligned} \tag{6.4.3}$$

Since the *producer* must both send and receive from the same state, this behaviour is clearly mixed-choice. The mixed-choice are inherently *safe*, as illustrated in Eq. (6.4.3). Note, the constraint '`true`' indicates no upper or lower bound.

After receiving `start`, the *producer* first waits to receive a `stop` signal for 1s. If `stop` is received, the *producer* terminates. Otherwise, the *producer* is to send `data`, and then reset x and recursively loop to P'_S. For the first second of each iteration, the *producer* must wait in case a `stop` signal can be received. The *producer* in Eq. (6.4.3) can be written using the Erlang API as follows:

```
Ps() -> {act,r_start, {timer,"x1",1000, {rec,"Ps", {act,r_stop,endP,
          aft,"x1", {act,s_data, {timer,"x1",1000, {rvar,"Ps"}}}}}}}.
```

Above, when `x1` completes the *producer* can send `data`. Since in Eq. (6.4.3) there is no upper-bound constraint for the *producer* sending `data`, the protocol allows this to take potentially forever. It is plausible that the use case for implementing a protocol with a Producer-Consumer pattern, the *producer* is dependent on some other service for obtaining the data to send. Since this is the latest possible action, if this service encountered an error and the *producer* was unable to send `data`, then the *producer* would become stuck, as would the *consumer*. Ideally, the latest action should be a dependency-free timeout signal used to provide a means to stop a program from being stuck when another process has seized (Peralta et al. 2003). For example, we can amend P'_S in Eq. (6.4.3) as follows:

$$P'_S = \left\{ ?\texttt{stop}(x \leq 1).\texttt{end}, \quad !\texttt{data}(1 < x < 99, \{x\}).P'_S, \quad !\texttt{pass}(x > 100, \{x\}).P'_S \right\}$$

Above, instead of a timeout, we have added the option to send `pass` if `data` on time. In practice, our tool would utilise the `nonblocking_payload` function discussed in Section 6.4.6 to allow the payload of `data` to be obtained in a non-blocking manner.

6.5 Runtime Monitor Generation

The runtime monitors in our tool are not themselves generated. Instead, our tool provides a single generic monitoring program, and generates the protocol specification in the form of an Erlang map, which is derived from the internal FSM representation. We choose to use this map representation since it allows us to more easily make use of pattern matching in function guard sequences.

6.5.1 Protocol Specification

Currently, the protocol specification provided for our monitors is derived from TOAST processes rather than TOAST protocols. In some cases it is possible to have a TOAST process that is a *maximal implementation* of a TOAST protocol (recall Section 6.4.3), and in such cases the process encapsulates the full range of behaviour described by the protocol. Therefore, it follows that a protocol specification derived from a maximal TOAST process is also capable of being used to monitor Erlang stubs derived from *non-maximal* TOAST processes. (Discussed further in Section 7.2.)

6.5.2 Transparency

Our monitors act as transparent mediators for communication, and can be inlined at either party within a session. Being transparent, our monitors do not modify the contents of the messages exchanged and therefore, neither party in a session has to be aware of whether the other party, or even if they themselves are being monitored.

6.5.3 Configurability

Our monitors are highly configurable, capable of performing both runtime verification and runtime enforcement. Runtime verification is the default, where they remain transparent, and any violation will result in the supervisor being notified. For runtime enforcement, *Erlang on TOAST* (2024) supports several presets, such as:

Enforce (strong) is a are more flexible preset than *verification* that allows the timing of interactions to be minimally adjusted to be more lenient, whilst still ensuring that the protocol is not violated (i.e., allowing sending and receiving actions to be re-tried at the next available state, in the case they were received a state too early).

Enforce (weak) preset allows the monitor to fully handle the timing of interactions for their monitored process within a session. In effect, this preset causes a monitor to only strictly enforce that interactions occur as prescribed by the protocol, and does not verify when the process actually attempts to perform certain actions.

The configurability of our monitors enable them to be useful outside the application of our toolchain. The 'enforce (weak)' preset allows developers to write completely

time unaware programs that can adhere to *time-sensitive* protocols, since the timing of interactions is handled by the monitor.

6.5.4 Handling Events

Our generic monitoring program uses Erlang `gen_statem` behaviour to traverse the FSM map. By using callback mode `handle_event_function` we are able to describe how each kind of event should be handled, rather than describing each and every event that happens for each state (as it would be using `state_functions` callback mode). Therefore, the programs of the monitors themselves are of a fixed size with minimal code duplication, compared to generated monitors using `state_functions`, whose size grows with the number of states and edges of the FSM.

For example, we only define once how the monitors handles message receptions, which is by checking if the current state has a receiving action matching the label of the message received. In the case where the reception is *not* prescribed, then the monitor will signal to the supervisor that Otherwise, if the reception *is* prescribed then, the monitor will forward the message to the monitored process and proceed to the next state (just as an FSM would behave).

Revisiting our running example (Example 6.2.1) once more, the internal FSM in Eq. (6.3.3) becomes the following Erlang API:

```
1 #{ init => state1a_recv_a,
2    map => #{ state1a_recv_a => #{recv => #{a => stop_state}},
3              state1b_send_b => #{send => #{b => state1a_recv_a}} },
4    timeouts => #{ state1a_recv_a => {x5, state1b_send_b},
5                   state1b_send_b => {?EQ_LIMIT_MS, error} },
6    resets => #{ state1a_recv_a => {x5 => 5000} },
7    errors => #{ state1b_send_b => b_took_too_long} }.
```

In the above FSM map: (i) line 1 specifies the initial state. (ii) lines 2–3 is a map between states, where each state has a set of `send` or `recv` actions, each with their own labels and resulting states. (iii) lines 4–5 is a map between states and a tuple that defines a timeout, with a duration and destination, where the duration can be an integer value for milliseconds, or reference an Erlang timer. (iv) line 6 shows a map for specifying upon entering a state, which Erlang timers to start (if any) and what

value to start them with. (v) line 7 shows a map between states and error messages, which are to be returned if state `error` is reached from any of the states.

The macro `?EQ_LIMIT_MS` is the degree of leniency given to actions that are meant to be performed at an exact time. Since constraints use integers as constants, we allow the user to specify how precise these constraints should be. However, a user should note that these values should be the same in practical scenarios where both participants are monitored. For example, in the case of Eq. (6.2.1), sending `b` must happen when $(x = 5)$. Once state `state1b_send_b` is reached, a timeout is set for the duration of `?EQ_LIMIT_MS` which when triggered will automatically state-transition to `error`, signalling to the supervisor that the protocol has been violated. However, if the monitored process performs the action of sending `b`, then the subsequent state-transition to `state1a_recv_a` cancels the current timeout that leads to `error`.

6.5.5 Extensibility

Our generic monitoring program specifies how events should be handled by the monitor in general, providing a foundation for runtime monitoring of an Erlang program against the specification derived from the Erlang API.

The event handlers found in the generic monitoring program are specified as generically as possible in order to facilitate as many states as possible. Each kind of event handler function is separated and clearly annotated within the generic monitoring program. In the case where the user requires more specific or unique behaviour from an event handler, it is simply a matter of: (1) re-defining the event handler on the lines above the generic behaviour, using the original handler as a basis, and then (2) amending the pattern matching guard sequence of the function to specify the distinct and identifiable circumstances this definition should handle (e.g., specifying the specific state where this should be called). Finally, (3) add the desired functionality to the body of the newly define event handler function. If done correctly, all other instances of the even will be unaffected by the new addition, and be routed through to the same function clauses as before. By default, the generic monitoring program features an example of this unique behaviour with a set of event handler functions that serve to 'reserve' the label '`emergency_signal`' for a special

kind of transition. In the case that a monitor performs an action with the label 'emergency_signal' (initiated by the monitored process) then, regardless of the protocol specification, the monitor will create a file containing all of its logs, and signal to the supervisor that an error has occurred. This is just one example of how the generic event handling behaviour of our monitoring program can be extended to handle specific use cases, even beyond those prescribed by the protocols. Naturally, such behaviour is *also* a violation of protocols, and may interfere with any behavioural guarantees provided by the theory.

6.6 Using the Toolchain

Specific information of how to use the tool can be found in the README.md of the main project,[5] along with some examples in the README.md of the **sample-app** repository.[6]

6.6.1 Session Initiation

Sessions are started in an Erlang/OTP supervision tree, with a supervisor for each party. Each generated stub features the macro ?MONITORED which if set true will cause the stub to automatically start their inline monitor, which by default acts as a transparent mediator.

6.6.2 Session Configuration

Since our monitors act as transparent mediators within a session, the data exchanged is unaffected, and either party does not have to accommodate if the other is or isn't monitored. Therefore, our monitors can be used in:

1. Fully symmetric sessions, where both parties are monitored and verified.
2. Asymmetric sessions, where only one party is monitored and verified.

In the latter case, it would be the sole responsibility of the *unmonitored* party to adhere to all timing constraints of the protocol.

[5] https://github.com/jonahpears/Erlang-on-TOAST/blob/main/README.md
[6] https://github.com/jonahpears/Erlang-on-TOAST-sample-app

6.6.3 Basic Usage

The following is taken from the `README.md` of the **sample-app** repository.

Building the Tool Naturally, the tool requires Erlang to be installed, and `rebar3` to be built. After cloning the repository to your local machine, open a terminal in the directory and run the following commands:

```
1 rebar3 compile; rebar3 shell
```

Mapping TOAST Processes to Erlang API The tool provides a method to automatically map a notion for TOAST processes to the Erlang API needed as input: `toast_process:to_protocol({ProtocolNameAtom, TOASTProcess})`. For example:

```
1 toast_process:to_protocol({send_once,
2                            {'p','<-',{'data','undefined'},'term'}})
```

would return: '`{act, s_data, endP}`.'

Stub Generation from Erlang API The function for generating an Erlang stub from the Erlang API is: '`gen_stub:gen(ProtocolName, Protocol, FileName)`', where '`Protocol`' is the Erlang API and, '`ProtocolName`' and '`FileExt`' determines the file name. E.g.:

```
1 gen_stub:gen(send_once, {act,s_data,endP}, ".erl").
```

would generate a file named '`send_once.erl`' containing the generated Erlang stub. The file will define the macro '`?PROTOCOL_SPEC`' containing the Erlang API used as input (above), along with macro '`?MONITOR_SPEC`' containing the following:

```
1 #{ init => state1_std,
2    map => #{ state1_std => #{send => #{data => stop_state}} },
3    timeouts => #{}, errors => #{}, resets => #{} }
```

The entirety of the generated Erlang stub can be found later in Appendix C.1.1.

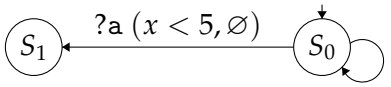
6.7 Use Cases

In this section we will: (i) provide a brief overview of how to use our toolchain using the running example given in Example 6.2.1, and (ii) discuss the design and implementation of a Two Factor Authentication (2FA) protocol.

6.7.1 Running Example

Let us recall our running example (Example 6.2.1) by looking at the TOAST protocol of Eq. (6.2.1) and the corresponding FSM representation of Eq. (6.3.2):

$$S = \left\{ ?\mathtt{a}(x < 5, \varnothing).\mathtt{end}, \quad !\mathtt{b}(x = 5, \{x\}).S \right\} \tag{6.2.1}$$



(6.3.2)

where in Eq. (6.3.2) $S_0 = S$ in Eq. (6.2.1). From here, we can encode the protocol using the Erlang API manually, or obtain it automatically by mapping a TOAST process.

Manual Encoding

Below is a straightforward encoding of Eq. (2.3.3) using the Erlang API:

```
1 {rec,"S", {act, r_a, endP,
2            aft, 5000, {act, s_b, {rvar,"S"}, aft, 0, error}}}
```

Note that the timeout of duration `0` above will be substituted for '`?EQ_LIMIT_MS`' for reasons discussed earlier in Section 6.5.4.

Automatic Mapping from TOAST Process

Alternatively, below is a potential TOAST process implementation of Eq. (6.2.1):

```
1 P() -> { 'def', { 'set', "x", {
2     'p', '->', {'les', 5000}, [ {{'a', 'undefined'}, 'term'} ],
3     'after', { 'if', {"x", 'eq', 5000}, 'then', {
4         'p', '<-', {'b', 'undefined'}, {'call', {"S", {[], []}}}
5       }, 'else', 'error' } } }, 'as', {"S", {[], []}} }
```

Notice the `if-then-else` statement which is used to enforce that message `b` is sent *exactly* 5s after the process began waiting for message `a` to arrive. Following Section 6.6.3, by calling '`toast_process:to_protocol({running_example, P()})`' we obtain the following Erlang API:

```
1 {rec,"S", {timer,"x5000",5000,
2           {act,r_a,endP,aft,5000,
3                {if_timer,"x5000",{act,s_b,{rvar,"S"}},'else',error}}}}
```

The Erlang API above has a similar structure to the manual encoding, except that it uses an `if-then-else` rather than an '`aft, 0`'. The stubs can be generated following Section 6.6.3.

6.7.2 Use Case: Two-Factor Authentication

In this section we discuss the design and implementation of a 2FA protocol. For services that need to be secure, such as online banking, it is crucial to ensure that a given user is who they claim to be. Such services typically employ 2FA to verify the integrity of a user's claimed identity, and it is common for such protocols to enforce timing constraints for the user's response. In Figure 6.2 we present a mock-up 2FA protocol which describes: upon a login request from a User a Bank sending two different codes to the User, one via SMS and the other to their email. Figure 6.2 specifies that the User must send these codes back to the bank within 10 minutes, after which the bank will determine if the login was a success of failure. The symbol (⦶) indicates that the SMS and Email interactions can occur in parallel, and the symbol (⊕) indicates a choice between the bank sending either `Success` or `Failure`. It can be interpreted that these initial messages sent by the bank are assumed to occur instantly. In Eq. (6.7.1), we model Figure 6.2 as a TOAST protocol, where in this example we 'collapse' the parallelism into a single sequence of actions since otherwise, we would need to specify all possible interleavings:

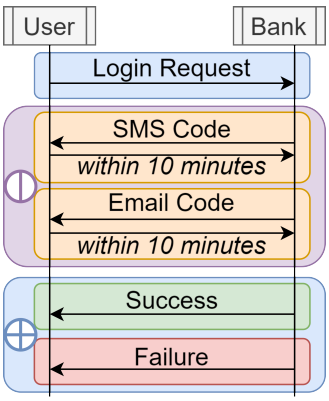


FIGURE 6.2: Bank Application of Two-Factor Authentication

$$
\begin{aligned}
U_0 = {} & !\texttt{login}(\texttt{true}, \{x, y\}) \\
& .?\texttt{email\_code}(\texttt{true}).!\texttt{email\_auth}(\texttt{true}) \\
& .?\texttt{SMS\_code}(\texttt{true}).!\texttt{SMS\_auth}(\texttt{true}) \\
& .\left\{ \begin{array}{l} ?\texttt{success}(x < 10 \wedge y < 10).U_1, \\ ?\texttt{failure}(x \geq 10 \vee y \geq 10).\texttt{end} \end{array} \right\}
\end{aligned}
\tag{6.7.1}
$$

Above, Eq. (6.7.1) specifies that `email_code` will be received before the `SMS_code`, and

`SMS_code` will not be received before the `email_auth` is sent back. While capturing the parallel behaviour in Figure 6.2 using TOAST is possible, it would be cumbersome. Thankfully, the mailboxes of Erlang processes enable Eq. (6.7.1) to be a plausible model of Figure 6.2 since messages may be received slightly out of order, and the end result of sending either `success` or `failure` depends on the parallel interactions both completing. (See fork and join in the work by Deniélou et al. (2012).)

However, to deploy runtime monitoring for *verification* the monitor requires an extended description of Eq. (6.7.1), detailing *all* possible traces, since *verification* does not allow message receptions to be postponed (discussed in Section 7.2).

6.7.3 Misleading Timing Constraints

Notably, in Eq. (6.7.1) there are no timing constraints for how promptly the user is required to respond to the bank. Instead, the timing constraints determine whether a `success` or `failure` are returned to the user, which is misleading since the contents of the codes are also crucial in determining whether the user is authenticated or not. Therefore, Eq. (6.7.1) could be improved by adding a timeout branch at each point of the protocol to determine if the bank should continue to wait for the users codes if its is already guaranteed that they will result in `failure`.

Below we provide an input protocol for our tool corresponding to behaviour of Figure 6.2 *and*: (1) the additional timeouts on the user responding within 10 minutes, (2) the full parallelised behaviour that was missing (or 'collapsed') in Eq. (6.7.1):

```
1 u0() -> {act,s_login, {rec, "retry", {
2   timer,"x600",600000, {
3     branch, [ u0(branch_r_email), u0(branch_r_sms) ]}}}}.
```

where timer `x600` is used to enforce an overarching timeout of 10 minutes for the user to send back `sms` and `email`.

```
4 u0(branch_r_email) -> {act,r_email, {
5  act,s_email, {act,r_sms, {
6    act,s_sms, {
7      branch, [ {act,r_success,u1()}, {act,r_failure,endP} ]},
8    aft,"x600", {act,r_timeout,{rvar,"retry"}}}},
9  aft,"x600", {act,r_timeout,{rvar,"retry"}}}};
```

Note that in the case above, the protocol specifies that the user must respond to `email` before receiving `sms`. In practice, the bank and the user are *not* dual and therefore, the bank will likely send both `email` and `sms` in sequence, before waiting to receive either `success` or `failure`. Below is the other branch:

```
11 u0(branch_r_sms) -> {act,r_sms, {
12  act,s_sms, {act,r_email, {
13    act,s_email, {
14      branch, [ {act,r_success,u1()}, {act,r_failure,endP} ]},
15    aft,"x600", {act,r_timeout,{rvar,"retry"}}}},
16  aft,"x600", {act,r_timeout,{rvar,"retry"}}}}.
```

Note that each user's sending action to the bank, is now a *co-timeout*. If timer `x600` completes at any point before the user has sent both codes, then they must stop and wait to receive a timeout from the bank, and will be able to `retry`.

Chapter 7

Conclusions

In this chapter we summarise the contributions presented in this thesis and discuss them in the context of related work, revisiting some of the background material previously covered in Chapter 2. First we discuss the contributions relating to TOAST presented in Chapters 3 to 5. Then we do the same for *Erlang on TOAST* (2024) presented in Chapter 6. Finally, we conclude this chapter (and this thesis) with a closing statement in Section 7.3.

7.1 Timeout Asynchronous Session Types

In Chapter 3, we have shown how timing constraints provide an intuitive way of integrating mixed-choice into asynchronous (binary) session types. There are many conceivable ways to realise mixed-choice using programming primitives. However, our integration with time, embodied in TOAST, offers new capabilities for modelling timeouts which sit at the heart of protocols and are a widely-used idiom in programming practice. To provide a bridge to programming languages, in Chapter 4 we provide a timed session calculus enriched with a `receive-after` pattern and process timers, providing the means to implement a timeout process and its dual. In Chapter 5 we establish a means of type-checking our processes with TOAST.

Taken altogether, in Chapters 3 to 5 we have lifted a long-standing restriction on asynchronous session types by allowing for safe mixed-choice, through the judicious application of timing constraints. Additionally, we have shown that processes that are well-typed by our type system adhere to subject reduction (Corollary 5.4.5) which means that any step the system can take will maintain the shape of a well-typed

system. Subject reduction guarantees that when an action happens, its sender, receiver, label, payload, and timing all conform to the types.

7.1.1 Related Work

Since the work on TOAST (Pears et al. 2023, 2024c) builds upon the theory of ATST by Bocchi et al. (2019), we begin by highlighting the differences between the two works. We then review the work by Hou et al. (2024), and show the previously relegated discussion from Chapter 2. Later we compare the expressiveness of TOAST with regard to mixed-choice and timeouts to those other works discussed in Section 2.3.

Asynchronous Timed Session Types

Building upon the work on ATST by Bocchi et al. (2019), the extended types of TOAST are streamlined (with only a single interaction type) and more expressive since they allow for mixed-choice. TOAST processes are also extended from ATST. The ATST processes by Bocchi et al. (2019) allow for a simplified variant of a timeout where a branching action is annotated with a deadline, akin to the expression e in the branching process of TOAST (i.e., $p^e \triangleright \left\{ l_i(v_i) : P_i \right\}_{i \in I}$). In Bocchi et al. (2019), if such an ATST process did not receive a corresponding message before the deadline, an `error` would be reached, while in TOAST the typing system requires that such an occurrence cannot occur, unless the branch prefixes an 'after' branch (i.e., `after` Q) which may be taken otherwise. This feature naturally facilitates the use of timeouts in TOAST processes. The added expressiveness of mixed-choice in TOAST allows us to introduce a process to execute upon a timeout to express, for instance, retry strategies, which are not expressible in ATST.

Additionally, TOAST processes feature process timers and time-sensitive conditional statements (i.e., `if-then-else` processes with conditions on process timers) in order to describe the corresponding 'dual' of a timeout. These enable a process to anticipate their own timeliness and know when it is too late to perform a send action, and they should instead do something else. Together, the timeout '`receive-after`'

processes and the time-sensitive '`if-then-else`' statements enable TOAST processes to express the same range of *safe* mixed-choice as TOAST types.

Extending the conjecture of Bocchi et al. (2019), type-checking using our typing system is decidable under the assumption that channel and recursion variables are annotated with typing information. The rules in our typing system with infinitely many premises, such as '$\forall t \in d$' in the premise of rule [`Del-`d] in Eq. (5.2.2), can be accounted for symbolically, such as by exploiting zones and Difference Bound Matrixs (DBM) (Bengtsson et al. 2003) or Satisfiability Modulo Theory (SMT).

Remark 7.1.1 (Errors & Progress)**.** Unlike ATST by Bocchi et al. (2019), TOAST processes do not feature the `error` process.[1] More similarly to the earlier work by Bocchi et al. (2014) of TMPST, instead of letting a process reaching an error state (`error`) when an upper-bound (deadline) is violated, we simply do not allow time to pass (see Definition 4.2.2). We choose to do this since it allows us to manage the complexity added by mixed-choice, and separate the concerns of subject reduction, safety and progress. Normally, in the untimed scenario, safety and progress come as consequence of subject reduction. Conversely, in a timed scenario with error states, subject reduction depends on progress – i.e., if a process gets stuck then it must be that the timing constraints have been violated causing an error state to be reached, hence compromising subject reduction.

In Bocchi et al. (2019) this circular dependency was overcome by requiring a progress property called receive-liveness, which is defined on the untimed counterpart of a timed process, which can be checked by well-established techniques (Bettini et al. 2008; Dezani-Ciancaglini et al. 2007). Unfortunately, not only do techniques such as those presented by Dezani-Ciancaglini et al. (2007) not apply to mixed-choice, but they are hardly attainable in this setting since removing time from a mixed-choice is likely to introduce progress violations that would otherwise not exist in the presence of time. Therefore, removing `error` processes became critical in order to break this circular dependency and establish subject reduction. As a future work, we would like to establish a progress property on the basis of subject reduction. △

[1] Note, while the original work by Pears et al. (2023) on TOAST *did* feature an `error` process, the newly extended version (Pears et al. 2024c) does not, for the reasons outlined in the text.

Timed Semantics The semantics of TOAST is based upon the timed semantics of CTA by Krcál et al. (2006). Specifically, the time-passing semantics of rule [time] in Eq. (3.3.3) is based upon the *urgent semantics* by Bartoletti et al. (2018), to enforce *receive urgency* and ensure that the *latest enabled* action is not missed. These semantics differ from those presented by Bocchi et al. (2019) for ATST, which require that sending actions are *never* missed. In our setting a configuration featuring a mixed-choice, such as $(v_0, \{!a\,(x < 3, \varnothing)\,.S, ?b\,(x \geq 3, \varnothing)\,.S'\})$ would be unable to reach the latest-enabled receiving action b, since the preceding sending action a would *always* happen instead, effectively rendering b redundant. However, in this work we only require that the latest-enabled action is never missed, allowing instances such as a to be passed. This provides implementations of TOAST protocols the affordance of not featuring every sending action, and allows them to draw upon the full range of interactions described by the given TOAST protocol.

Affine Timed Multiparty Session Types

We now discuss the work by Hou et al. (2024) which present ATMP. Both the work on TOAST (Pears et al. 2023, 2024c) and Hou et al. (2024) build upon and extend the work of ATST by Bocchi et al. (2019) for the time oriented aspects of our work. Additionally, the work of Hou et al. (2024) also draws upon the recent pool of works that implement AMPST – namely those by Lagaillardie et al. (2020, 2022), which build upon work by Kokke (2019) and Mostrous et al. (2018). Affine sessions (Mostrous et al. 2018) support exception handling by enabling an end-point to perform a subset of the interactions specified by their type, but there is no consideration of time, hence *untimed* timeouts. Aside from the overlap with Bocchi et al. (2019) this work is particularly relevant since, like TOAST, it does not abstract away the specific timing of a timeout as in other works, but *can* model the behaviour of timeouts and their specific durations. This is one of the most notable similarities between TOAST and ATMP, since no other works in the timed, asynchronous setting do this and naturally, *untimed* works are unequipped to do this. Understandably, there are numerous similarities between the works of TOAST (Pears et al. 2023, 2024c) and ATMP (Hou et al. 2024) since we both build upon ATST by Bocchi et al. (2019).

We shall now discuss some of the *differences* between TOAST and ATMP.

Expressivity One of the most unique capabilities of TOAST processes is the ability to not only model the behaviour of timeouts, but also the behaviour of a timeout's *counterpart* – a mixed-choice where a send action can happen first. In most works that focus on implementing timeouts in process calculi, including ATMP by Hou et al. (2024), there is a focus on solely the timeout process (or '`receive-after`' behaviour) since this is the behaviour we see in real-world protocols. However, this takes for granted that in order for a timeout to trigger, something must have gone wrong, and the timeout is a failure-handling technique.

By contrast, TOAST focus on the relationship between timeouts and safe mixed-choice, and therefore, do not see a timeout triggering as a failure, but as an expressive interaction structure. In this way, TOAST can model time-aware processes that have the capacity to make decisions and react to the timeliness of other actions, where one timeout triggering does not mean another party has crashed, but that it was busy at the time. This is the expressive power granted by process timers and time-sensitive conditional statements, and a key feature that separates TOAST from other works on modelling asynchronous timeouts (timed or untimed).

Additionally, while ATMP does not feature time-sensitive conditional statements or process timers as does TOAST, it does inherit the '`try-catch`' behaviour from AMPST, which allows crashed processes to be 'dropped' and enables more fault-tolerant behaviour. It is this '`try-catch`' behaviour that enables the branch/receive processes with deadlines of ATMP to model the behaviour of a timeout, since once a deadline is reached the '`try`' portion of the process crashes, and the process proceeds as '`catch`'. This builds directly upon the branch/receive processes of ATST by Bocchi et al. (2019) which also had deadlines, which if reached would yield an `error` process.

Progress & Deadlock Freedom Notably, Hou et al. (2024) provide **deadlock freedom** for ATMP in order to guarantee **progress**. Their approach hinges on **session fidelity**, where they prove that each interaction prescribed by the types is expressed by a well-typed process. In this way, it remains for Hou et al. (2024) to show that well-typed ATMP processes are guaranteed to not get stuck, which they can prove

somewhat straightforwardly by leveraging the '`try-catch`' processes inherited from AMPST.

Unfortunately, since TOAST does not have '`try-catch`' or build upon works on affine types, this approach is not applicable for TOAST.

Other Works

Recall the work of MAGπ by Brun et al. (2023), which similar to TOAST, supports a 'timeout branch' on their branching processes (and types). The key difference with TOAST is that MAGπ is *untimed*, and the 'timeout branches' are taken non-deterministically, in order to model unexpected failures, which is the focus of their work. Additionally, TOAST processes feature time-sensitive conditional statements in our `if-then-else` processes to model the counterpart of a timeout, while Brun et al. (2023) instead staggers timeout processes in an alternating pattern. While this is effective in their context, it does limit the descriptive capabilities to mixed-choice with only two distinct regions of sending and receiving actions. In comparison, TOAST allows mixed-choice composed of any number of sending and receiving actions interleaved together in the same state.

Recall the work by Peters et al. (2022, 2023) who present FTMPST, which allow for an external failure detector to inform a process when to take a 'default branch' with a predefined value. While different from a timeout, this does allow a process that would otherwise be stuck waiting forever, to instead make progress. In TOAST, timeout process and timers closer align with programming primitives like timeouts and timers in, for example, Erlang (with its `receive-after` pattern and timers) and Go.

7.1.2 Future Work

There are two primary focuses for future work: (1) a progress property for our typing system and (2) an extension to the multiparty setting.

Progress for Timed Mixed-Choice

First and foremost, the priority of future work will be to provide a progress property for the typing system of TOAST (presented in Pears et al. (2024c)). Progress was included in previous work on asynchronous timed session types by Bocchi et al. (2019, 2014). However, for reasons outlined in Remark 7.1.1 we are unable to follow the same approach. In order to prove progress for TOAST we will have to take a different approach, which allows for explicit reasoning on the interaction structures in the timed, asynchronous setting. Work on this proof is underway.

Overview The key idea behind this approach is that we want to prove liveness *syntactically* rather than *semantically*, and reason on the syntactic structure of a session process (i.e., a process $P = (\nu pq)\ Q$) that is both *well-formed* and *well-typed* (against a *well-formed* TOAST type). The intuition is that when a well-formed process is type-checked and deemed to be well-typed, then the entire process has already been checked against the behaviour of a (well-formed) TOAST type and therefore, it should follow subject reduction (Corollary 5.4.5) that the process is well-behaved, and like the corresponding types, is guaranteed to exhibit *progress* (Theorem 3.4.5).

In our context, the crux of the issue lies in proving that at runtime, a TOAST process will *always* be able to make a step, unless terminated. Since we already know a well-typed TOAST process follows the behaviour of a well-formed TOAST type, then the only possible way for such a process to get stuck is if it were to get stuck waiting to receive a message. If we were to add a reduction rule to reduce such stuck processes to an `error` process, then we would have to provide a proof to show that such `error` processes are not reachable (semantically, via the reduction rules) for a process that is both well-formed and well-typed. Instead, we take another approach and aim to show that within a well-formed and well-typed session process, if there is a process waiting to receive a message, then either:

- The message is already waiting in its queue.
- Or, there is some other process that will send within the time limit.
- Or, the process waiting to receive has some timeout.

For each of the points above, we also require that the 'resulting continuation' session process is also *syntactically* live, i.e., the resulting continuation session process is analogous to the one obtained *semantically* via the corresponding reduction rules, except done so syntactically.

Multiparty TOAST

In future work it would be interesting to extend the theory of TOAST to the multiparty setting. In light of the recent work on ATMP by Hou et al. (2024), it would be especially interesting to be able to compare the expressiveness of a **multiparty** TOAST with ATMP. Specifically, as previously outlined, regarding the expressiveness of the process timers and time-sensitive conditional statements of TOAST processes, as opposed to the affine '`try-catch`' statements of ATMP.

7.2 Erlang on TOAST

Presented in Chapter 6, *Erlang on TOAST* (2024) generates Erlang stub programs from an Erlang API which can be obtained via a mapping from TOAST processes given in Chapter 4, following the work of Pears et al. (2024c). When obtaining an Erlang API from a TOAST process, we rely on the TOAST processes being well-typed against a well-formed TOAST protocol in order to ensure a good interaction structure, and certain guarantees of its behavioural properties. The generated programs can be configured to automatically start their own inline monitor (examples can be found in the `sample_app` repository[2]). (Though, the user still has to extend them with functionality. See the project `README.md`.)[3]

7.2.1 Related Work

The *Erlang on TOAST* (2024) toolchain presented in Chapter 6 (and Pears et al. (2024a)) builds upon the tool '*Protocol Reengineering Implementation* (2023)' that accompanies work by Bocchi et al. (2023) which takes an *untimed* process notation as input and generates Erlang `gen_statem` stubs for API implementation.

[2] https://github.com/jonahpears/Erlang-on-TOAST-sample-app/tree/main
[3] https://github.com/jonahpears/Erlang-on-TOAST/blob/main/README.md

Protocol Reengineering

Naturally, *Erlang on TOAST* (2024) shares some similarities with *Protocol Reengineering Implementation* (2023). Notably, the work by Bocchi et al. (2023) present their own language for specifying protocols which lie outside the field of session types – though, the authors acknowledge it resembles both the Calculus of Communicating Systems (CCS) and π-calculus by Milner (1982). Therefore, the notation that the original tool (*Protocol Reengineering Implementation* 2023) used as input does not align perfectly with the processes of the π-calculus. For this reason, *Erlang on TOAST* (2024) uses an altered definition of the protocol specification (i.e., the Erlang API). For example, the Erlang API features the ability to set process timers and conditional statements on process timers, while other notation have been dropped; namely, 'assert', 'require' and 'consume'. Additionally, like many other works implementing theory of session types (or adjacent), *Protocol Reengineering Implementation* (2023) relies on extracting FSM from the input to generate their code, which is a feature also present in *Erlang on TOAST* (2024).

Beyond the generation of an FSM describing a given protocol, the process of code generation of *Erlang on TOAST* (2024) and *Protocol Reengineering Implementation* (2023) is fundamentally different, since the Erlang program stubs generated by *Erlang on TOAST* (2024) do not follow Erlang/OTP `gen_statem` behaviour. Notably, we only use Erlang/OTP `gen_statem` behaviour in the *generic runtime monitoring program*. While `gen_statem` offers a close correlation with the theory, ordinary Erlang stubs are more intuitive and easier for programmers unfamiliar with `gen_statem` to interpret and extend with functionality. Additionally, *Erlang on TOAST* (2024) produces stub programs capable of being executed from the command-line almost immediately, allowing programmers to begin developing and testing the program.

Runtime Monitoring

Recall Section 6.3.3, which provided an overview of the difference classifications of runtime monitors, drawing from the works of Aceto et al. (2024), Bartocci et al. (2018), and Francalanza et al. (2017, 2018). Broadly, there are three categories of runtime monitors: (a) **verification**, (b) **adaptation** and (c) **enforcement** monitors. The

monitors featured in *Erlang on TOAST* (2024) (presented in Chapter 6) support either runtime verification or enforcement (a & c).

Outside of session types, Aceto et al. (2024) generate runtime monitors which are then inlined into a program via code injection at compile-time. *Erlang on TOAST* (2024) achieves inlining differently, by spawning the monitor within the same node the monitored process interacts with their monitor synchronously.

Multiparty Session Actors Fowler (2016) presents a tool based on work by Neykova et al. (2014) for generating runtime monitors for Erlang from Scribble protocols (by Honda et al. (2011) and Yoshida et al. (2014)). Both *Erlang on TOAST* (2024) and Fowler (2016) derive monitors from the theory of asynchronous session types, and utilise the supervision tree structure of Erlang for session coordination, instantiation and reporting violations. However, *Erlang on TOAST* (2024) only consider a single binary session while Fowler (2016) support for multiple multiparty sessions.

Remark 7.2.1 (A Monitoring Template)**.** Unlike the other works discussed in Section 2.1.2 that generate runtime monitors (such as Aceto et al. (2024), Bocchi et al. (2013), Burlò et al. (2021, 2022), Fowler (2016), Neykova (2013), and Neykova et al. (2017a, 2013)), in *Erlang on TOAST* (2024) monitors are not *generated*. Instead, monitors are spawned from a single generic monitoring program, which either verifies or enforces the behaviour specified by an automata notation derived from session types. △

Code Generation

Recall the work by Neykova et al. (2018) which presents a toolchain for generating F# code from Scribble (Yoshida et al. 2014) protocols which provide 'live', responsive feedback in the form of code snippet hints/suggestions to direct the user on how they should proceed in order to implement the given protocol. By contrast, *Erlang on TOAST* (2024) is far more straightforward since it only takes an input from the user and then generates an Erlang program stub for the user to extend. In future work it may be interesting to experiment with such an approach, where a user is guided with how to construct a correct program rather than having it generated for them.

7.2.2 Future Work

Our primary objectives in future work will be to: (a) aid users in designing TOAST protocols (b) semi-automate the process of extracting TOAST processes from TOAST protocols and (c) derive our monitor specification directly from TOAST protocols. We would also like to extend this tool with multiparty sessions, which would naturally follow the theory (as outlined in Section 7.1.2).

TOAST Type-checker Implementation

As part of this growing toolchain, we aim to include an implementation of our typing system. This is already underway, with a preliminary implementation already complete. This basic version is written in Erlang, since we make use of the existing notations of TOAST protocols and processes. For the checking of timing constraints and their satisfiability, we outsource this checking to the *Z3 Theorem Prover* (2024). In future iterations of this implementation, we would like to move more of the functionality over to be implemented in Z3.

Complex Constraints

As previously outlined in Section 6.4.3, it would be interesting to explore how we might add support for more complex timing-constraints found in TOAST protocols (types). In the current version of *Erlang on TOAST* (2024), the Erlang API does not facilitate the full range of timing constraints featured in the theory (see Table 6.1). In this direction, it may be interesting to explore works such as Bartoletti et al. (2018), in order to determine if we can simplify a timing constraint such that it can be more easily encoded using the Erlang API.

Other Considerations

Exploring recovery strategies for time-sensitive protocols would be interesting future work. In the work by Neykova et al. (2017b), they present a new recovery strategy based on session types, which allows processes affected by a crash to be 'recovered' to some prior point (as opposed to them all crashing and being completely restarted).

Streamlined Toolchain Another consideration would take inspiration from the tool in Neykova et al. (2018), which instead of generating code to a file, provides the user with responsive feedback and suggestions. However, it would be crucial to consider how any functionality added by the programmer outside the scope of the protocol could affect the timings of interactions.

Benchmarks & Evaluation In future work we hope to conduct a more thorough evaluation of our toolchain by comparing the performance of different generated implementations, the effectiveness, accuracy and performance overhead of our monitors and provide a wider range of examples and benchmarks.

Using Scribble Hou et al. (2024) recently presented `MultiCrusty`^{T}, a *timed* extension to the Scribble-based toolchain for Rust named `MultiCrusty` by Lagaillardie et al. (2022). For future work it would be interesting to explore how TOAST theory can be integrated within the Scribble ecosystem.

7.3 Closing Statement

This thesis presents TOAST, which help lift a long-standing limitation on the descriptive capabilities of session types by enabling timeouts to be modelled, and furthermore a subset of time-sensitive *safe* mixed-choice. In recent years, there have been numerous other advances in this area as well (Brun et al. 2023; Casal et al. 2020; Hou et al. 2024; Peters et al. 2024; Vasconcelos et al. 2020) which is a testament to the importance of tackling this issue. Notably, the theoretical work presented in this thesis (Pears et al. 2023, 2024c) are the only ones to do so in the *timed*, *asynchronous* setting (though Hou et al. (2024) emerged shortly after).

To supplement the theory of TOAST, *Erlang on TOAST* (2024) (Pears et al. 2024a) helps bridge the gap with programming practice, by enabling a means of generating correct-by-construction Erlang stub programs, which when used with the accompanying generic runtime monitoring program are resilient to user-introduced violations.

Perhaps the most unique thing about TOAST is the view of timeouts as safe mixed-choice, rather than a fault-tolerance technique or failure-handling strategy. The work on TOAST (both types and processes) focuses on the relationship between safe mixed-choice and timeouts, with the goal of expanding the descriptive capabilities of asynchronous, timed session types. In this way, TOAST take a very straightforward approach to modelling timeouts, with as little abstraction as possible, unlike other works (Brun et al. 2023; Hou et al. 2024; Peters et al. 2023) which broadly treat a timeout as some kind of failure to be dealt with. To this end, TOAST processes have a unique expressive power capable of modelling the behaviour of time-aware processes each able to react to both their own timeliness and the timeliness of others, in the broader spirit of safe mixed-choice.

In all, this thesis has served to explore the relationship between timeouts and safe mixed-choice in the timed, asynchronous setting and presented TOAST, a system capable of expressing more than just timeouts using safe mixed-choice. In addition to this theoretical contribution, we have explored in this thesis a proof-of-concept toolchain implemented in Erlang for generating correct-by-construction program stubs, in addition to a generic runtime monitoring program for performing runtime monitoring using TOAST.

Appendix A

Proof of Type Progress

The result of Theorem 3.4.5 follows similar reasoning to Bocchi et al. (2019). While Theorem 3.4.5 states progress for *initial system configurations*, progress is also guaranteed for systems composed of *well-formed* and compatible configurations by Lemma A.7.3. Our proof shows that our 'well-formedness' rules in Eq. (3.4.1) ensure that the timing-constraints on actions are *feasible*, and structured such that any mixed-choice are *safe*. We show that our LTS in Eqs. (3.3.2 to 3.3.4) preserve both 'well-formedness' and compatibility of our configurations.

A.1 Auxiliary Definitions & Assumptions

Definition A.1.1 (Latest-enabled Configurations). (ν, S) is *le* if, $\exists \square, \square'$ such that $\square \neq \square'$ and $(\nu, S) \overset{\square}{\Rightarrow}$ and $(\nu, S) \overset{\square'}{\not\Rightarrow}$.

Definition A.1.2 (Live Configurations). (ν, S) is live if, $S = \mathtt{end}$ or (ν, S) is *fe*.

Definition A.1.3 (Free Names). We define $\mathtt{fn}(S)$, the *free names* of a type S, as:[1]

$$\mathtt{fn}(S) = \begin{cases} \{\alpha\} & \text{if } S = \alpha \\ \mathtt{fn}(S') \setminus \{\alpha\} & \text{if } S = \mu\alpha.S' \\ \bigcup_{i \in I} \mathtt{fn}(S_i) & \text{if } S = \left\{ \square_i \, l_i \, \langle T_i \rangle \, (\delta_i, \lambda_i).S_i \right\}_{i \in I} \\ \varnothing & \text{if } S = \mathtt{end} \end{cases}$$

[1] As in Pierce (2002).

Definition A.1.4 (Capture Avoiding Substitution)**.** Substitution $[S'/\alpha]$ is *capture avoiding* for S iff:

$$S = \left\{\Box_i\, l_i\, \langle T_i\rangle\, (\delta_i, \lambda_i).S_i\right\}_{i\in I} \implies [S'/\alpha] \text{ is capture avoiding for } S_j, \forall j \in I$$
$$S = \mu\beta.S'' \wedge \alpha \neq \beta \implies \beta \notin \mathtt{fn}(S') \text{ and } [S'/\alpha] \text{ is capture avoiding for} S''$$

With an abuse of notation, we say that $S\,[S'/\alpha]$ is capture avoiding when $[S'/\alpha]$ is capture avoiding for S.

Definition A.1.5 (Unfold Equivalence)**.** We define $(\overset{\text{unfold}}{\equiv})$ as the smallest relation such that:

$$S \overset{\text{unfold}}{\equiv} S \qquad S \overset{\text{unfold}}{\equiv} S' \implies S' \overset{\text{unfold}}{\equiv} S \qquad S \overset{\text{unfold}}{\equiv} S' \wedge S' \overset{\text{unfold}}{\equiv} S'' \implies S \overset{\text{unfold}}{\equiv} S''$$
$$S\,[\mu\alpha.S/\alpha] \text{ is capture avoiding} \implies \mu\alpha.S \overset{\text{unfold}}{\equiv} S\,[\mu\alpha.S/\alpha] \qquad S \overset{\text{unfold}}{\equiv} S' \implies \mu\alpha.S \overset{\text{unfold}}{\equiv} \mu\alpha.S'$$
$$S \overset{\text{unfold}}{\equiv} S' \implies \left\{\Box l\, \langle T\rangle(\delta,\lambda).S,\ \Box'' l''\, \langle T''\rangle(\delta'',\lambda'').S''\right\} \overset{\text{unfold}}{\equiv} \left\{\Box l\, \langle T\rangle(\delta,\lambda).S',\ \Box'' l''\, \langle T''\rangle(\delta'',\lambda'').S''\right\}$$

Assumption A.1.6. *If $A, \alpha : \delta'; \delta \vdash S$, then $\alpha \notin Dom(A)$.*

A.2 Well-formed Types

Lemma A.2.1. *If $\alpha \notin \mathtt{fn}(S)$ and $\alpha \notin Dom(A)$, then $A, \alpha : \delta'; \delta \vdash S \iff A; \delta \vdash S$.*

Proof. Since $\alpha \notin \mathrm{Dom}(A)$ it follows that recursive call α has not yet been defined. Additionally, since $\alpha \notin \mathtt{fn}(S)$, then it follows Definition A.1.3 that if $\mu\alpha.S'$ appears at any point within S, then α *may* appear at some point within S'. For each case of $(\iff)$:

Case 1. $A, \alpha : \delta'; \delta \vdash S \implies A; \delta \vdash S$. If $S = \mu\beta.S'$, then by rule [rec]:

$$\frac{A, \alpha : \delta', \beta : \delta; \delta \vdash S'}{A, \alpha : \delta'; \delta \vdash \mu\beta.S'}\ [\mathtt{rec}]$$

Notice the environment $A, \alpha : \delta', \beta : \delta$ in the premise of rule [rec] (above). If $\beta = \alpha$ then, the environment would contain duplicate variables. Therefore, $\beta \neq \alpha$ or β must be renamed to some $\beta' \notin \mathrm{Dom}(A, \alpha : \delta')$ and $S = \mu\beta'.S'\,[\beta'/\beta]$. Similarly, if $S = \beta$ then, since $\alpha \notin \mathtt{fn}(\beta)$, it must be that $\beta \neq \alpha$. In both of these cases, the presence of $\alpha : \delta$ is immediately inconsequential, and will remain so for

the remainder of the evaluation. Clearly, if $S = \texttt{end}$ then the same holds. If $S = \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$ then by Definition A.1.3, it must also hold for all $i \in I$. In summary, since $\alpha \notin \mathrm{Dom}(A)$ and $\alpha \notin \texttt{fn}(S)$, it follows that α must correspond to a recursive variable in a different unfolding (either outside S, or a future unfolding within S), and that it has been renamed from the corresponding recursive variable in the current unfolding.

Case 2. $A; \delta \vdash S \implies A, \alpha : \delta'; \delta \vdash S$. Following Case 1, the presence of $\alpha : \delta$ is inconsequential since α corresponds to some already unfolded recursive call outside S, or a some future unfolding within S. In both cases, the α has been renamed from the corresponding recursive variable in the current unfolding. □

Lemma A.2.2 (Substitution Lemma)**.** *Let* $A; \delta' \vdash S'$. *If* $\alpha \notin \mathrm{Dom}(A)$ *and* $S\,[S'/\alpha]$ *is* capture avoiding, *then* $A; \delta \vdash S\,[S'/\alpha] \iff A, \alpha : \delta'; \delta \vdash S$ *holds.*

Proof. We proceed by induction on the structure of S:

Case 1. If $S = \mu\beta.S''$ then, for each case of ($\iff$):

i. $A; \delta \vdash \mu\beta.S''\,[S'/\alpha] \implies A, \alpha : \delta'; \delta \vdash \mu\beta.S''$.

a. If $\beta = \alpha$, then $\mu\beta.S''\,[S'/\alpha] = \mu\beta.S''\,[S'/\beta] = \mu\beta.S''$. Therefore:

$$A; \delta \vdash \mu\beta.S''$$

Since $\alpha = \beta$ and $\alpha \notin \mathrm{Dom}(A)$, by Lemma A.2.2 we have the thesis:

$$A, \alpha : \delta'; \delta \vdash \mu\beta.S''$$

b. If $\beta \neq \alpha$, then by rule [rec]:

$$\frac{A, \beta : \delta; \delta \vdash S''\,[S'/\alpha]}{A; \delta \vdash \mu\beta.S''\,[S'/\alpha]}\ [\texttt{rec}]$$

By Definition A.1.4, $\beta \notin \texttt{fn}(S')$ and $S''\,[S'/\alpha]$ is *capture avoiding*. By Lemma A.2.1: $A, \beta : \delta; \delta'; \vdash S' \iff A; \delta' \vdash S'$. By the induction hypothesis:

$$A, \alpha : \delta', \beta : \delta; \delta \vdash S'' \iff A, \beta : \delta; \delta; \vdash S''\,[S'/\alpha]$$

We obtain our thesis following the conclusion of rule [rec] (below), using the former case (above) as our premise:

$$\frac{A, \alpha : \delta', \beta : \delta; \delta \vdash S''}{A, \alpha : \delta'; \delta \vdash \mu\beta.S''} \ [\texttt{rec}]$$

ii. $A, \alpha : \delta'; \delta \vdash \mu\beta.S'' \Longrightarrow A; \delta \vdash \mu\beta.S'' \, [S'/\alpha]$. By rule [rec]:

$$\frac{A, \alpha : \delta', \beta : \delta; \delta \vdash S''}{A, \alpha : \delta'; \delta \vdash \mu\beta.S''} \ [\texttt{rec}]$$

It cannot be that $\beta = \alpha$, as otherwise the environment $A, \alpha : \delta', \beta : \delta$ would contain a duplicate variable (as in the premise of rule [rec], above). Therefore, it must be that $\beta \neq \alpha$. Since $\mu\beta.S'' \, [S'/\alpha]$ is *capture avoiding*, by Definition A.1.4, $\beta \notin \texttt{fn}(S')$ and $S'' \, [S'/\alpha]$ is also *capture avoiding*. By Lemma A.2.1: $A; \delta' \vdash S' \iff A, \beta : \delta; \delta' \vdash S'$. By the induction hypothesis:

$$A, \alpha : \delta', \beta : \delta; \delta \vdash S'' \iff A, \beta : \delta; \delta \vdash S'' \, [S'/\alpha]$$

We obtain our thesis following the conclusion of rule [rec] (below), using the latter case (above) as the premise:

$$\frac{A, \beta : \delta; \delta \vdash S'' \, [S'/\alpha]}{A; \delta \vdash \mu\beta.S'' \, [S'/\alpha]} \ [\texttt{rec}]$$

Case 2. If $S = \beta$ then, for each case of ($\iff$):

i. $A; \delta \vdash \beta \, [S'/\alpha] \Longrightarrow A, \alpha : \delta'; \delta \vdash \beta$.

a. If $\beta = \alpha$, then $\beta \, [S'/\alpha] = S'$ and $\delta' = \delta$. Since $\alpha \notin \text{Dom}(A)$, it follows:

$$\frac{}{A', \beta : \delta; \delta \vdash \beta} \ [\texttt{var}]$$

b. If $\beta \neq \alpha$, then $\beta \, [S'/\alpha] = \beta$ and $A = A', \beta : \delta$ and by rule [var]:

$$\frac{}{A', \beta : \delta; \delta \vdash \beta} \ [\texttt{var}]$$

By rule [var] (and up-to reordering of variables):

$$\frac{}{A', \beta : \delta, \alpha : \delta'; \delta \vdash \beta}\ [\texttt{var}]$$

ii. $A, \alpha : \delta'; \delta \vdash \beta \implies A; \delta \vdash \beta\,[S'/\alpha]$.

a. If $\beta = \alpha$, then $\delta' = \delta$ and $\beta\,[S'/\alpha] = S'$. The thesis coincides with the hypothesis:

$$A; \delta' \vdash S'$$

b. If $\beta \neq \alpha$, then $\beta\,[S'/\alpha] = \beta$ and $A = A', \beta : \delta$ and by rule [var]:

$$\frac{}{A', \alpha : \delta', \beta : \delta; \delta \vdash \beta}\ [\texttt{var}]$$

The thesis follows immediately by rule [var]:

$$\frac{}{A', \beta : \delta; \delta \vdash \beta}\ [\texttt{var}]$$

Case 3. If $S = \left\{ \Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I}$ then, for each case of ($\iff$):

i. $A; \delta \vdash \left\{ \Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I} [S'/\alpha] \implies A, \alpha : \delta'; \delta \vdash \left\{ \Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I}$.

By Definition A.1.4, for all $i \in I$, $S_i\,[S'/\alpha]$ is *capture avoiding*. By rule [choice]:

$$\frac{\begin{array}{rl} \forall i \in I : A;\ \gamma_i \vdash S_i\,[S'/\alpha] \ \wedge\ \delta_i\,[\lambda_i \mapsto 0] \models \gamma_i & \text{(feasibility)} \\ \forall i, j \in I : i \neq j \implies \delta_i \wedge \delta_j \models \texttt{false} \vee \Box_i = \Box_j & \text{(mixed-choice)} \\ \forall i \in I : T_i = (\delta''',\ S''') \implies \varnothing;\ \gamma' \vdash S''' \ \wedge\ \delta''' \models \gamma' & \text{(delegation)} \end{array}}{A;\ \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{ \Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I} [S'/\alpha]}\ [\texttt{choice}] \tag{A.2.1}$$

where $\delta = \downarrow \bigvee_{i \in I} \delta_i$. By the induction hypothesis:

$$A; \gamma_i \vdash S_i\,[S'/\alpha] \iff A, \alpha : \delta'; \gamma_i \vdash S_i \qquad \text{where } \delta_i\,[\lambda_i \mapsto 0] \models \gamma_i$$

Therefore, we obtain our thesis following the conclusion of rule [choice] (below), using the latter case (above) as the (feasibility) premise:

$$\frac{\begin{array}{ll}\forall i \in I : A, \alpha : \delta';\ \gamma_i \vdash S_i \ \wedge\ \delta_i\,[\lambda_i \mapsto 0] \models \gamma_i & \text{(feasibility)}\\ \forall i, j \in I : i \neq j \implies \delta_i \wedge \delta_j \models \texttt{false} \vee \square_i = \square_j & \text{(mixed-choice)}\\ \forall i \in I : T_i = (\delta''',\ S''') \implies \varnothing;\ \gamma' \vdash S''' \ \wedge\ \delta''' \models \gamma' & \text{(delegation)}\end{array}}{A, \alpha : \delta';\ \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}}\ [\texttt{choice}] \tag{A.2.2}$$

ii. $A, \alpha : \delta'; \delta \vdash \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I} \implies A; \delta \vdash \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I} [S'/\alpha]$.

By rule [choice] as in Eq. (A.2.2), $\delta = \downarrow \bigvee_{i \in I} \delta_i$. By the (feasibility) premise of rule [choice], for all $i \in I$, $A, \alpha : \delta'; \gamma_i \vdash S_i$ and $\delta_i\,[\lambda_i \mapsto 0] \models \gamma_i$. Therefore, by the induction hypothesis:

$$A, \alpha : \delta; \gamma_i \vdash S_i \iff A; \gamma_i \vdash S_i\,[S'/\alpha] \qquad \text{where } \delta_i\,[\lambda_i \mapsto 0] \models \gamma_i$$

Our thesis follows the conclusion of rule [choice] in Eq. (A.2.1), where we use the latter case (above) as the (feasibility) premise.

Case 4. If $S = \texttt{end}$ then, by rule [end]:

$$\frac{}{A; \texttt{true} \vdash \texttt{end}}\ [\texttt{end}]$$

where $\delta = \texttt{true}$. By rule [end], end is *well-formed* against any δ and any A. It follows that $\texttt{end}\,[S'/\alpha] = \texttt{end}$. Therefore, the hypothesis holds for $S = \texttt{end}$.

We have shown the hypothesis to hold for any S. The interesting cases are when $S = \mu\beta.S''$ or $S = \beta$, which vary depending on $\alpha = \beta$ or $\alpha \neq \beta$. Since $\alpha \notin \mathrm{Dom}(A)$, we are dealing with either: (1) an unfolding of a yet-to-be defined recursion α, or (2) an future unfolding of a renamed recursion. By our thesis, we prove that unfolding and renaming recursive types does not interfere with the judgements of 'well-formedness' found in the rules in Eq. (3.4.1). □

A.3 Well-formed Configurations

Lemma A.3.1. $(\nu, \texttt{end})$ *is always* well-formed.

Proof. By Definition 3.4.2, $\exists\delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash \texttt{end}$. By rule [end] in Eq. (3.4.1), $\delta = \texttt{true}$. Therefore, our thesis follows $\nu \models \texttt{true}$. □

Lemma A.3.2. *If* (ν, S) *is* well-formed, *then* $S \neq \alpha$.

Proof. Let us consider by contradiction that $S = \alpha$. By Definition 3.4.2, $\exists\delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash \alpha$. The only rule applicable to α is [var]:

$$\frac{}{A, \alpha : \delta; \delta \vdash \alpha}\ [\texttt{var}]$$

Yet, $\varnothing \neq A, \alpha : \delta$, and so $S = \alpha$ contradicts with Definition 3.4.2 and we obtain our thesis. □

Lemma A.3.3. *If* $S \overset{\texttt{unfold}}{\equiv} S'$, *then* $A; \delta \vdash S \implies A; \delta \vdash S'$.

Proof. We proceed by induction on the last equation of Definition A.1.5 used to establish $S \overset{\texttt{unfold}}{\equiv} S'$.

Case 1. If $S = \mu\alpha.S_0$ with $S_0\,[\mu\alpha.S_0/\alpha]$ capture avoiding, then $S' = S_0\,[\mu\alpha.S_0/\alpha]$. The only rule applicable to $A; \delta \vdash \mu\alpha.S_0$ is rule [rec]:

$$\frac{A, \alpha : \delta; \delta \vdash S_0}{A; \delta \vdash \mu\alpha.S_0}\ [\texttt{rec}]$$

Note that α cannot be in $\mathrm{Dom}(A)$. Since $S_0\,[\mu\alpha.S_0/\alpha]$ is capture avoiding, by Lemma A.2.2:

$$A, \alpha : \delta; \delta \vdash S_0 \iff A; \delta \vdash S_0\,[\mu\alpha.S_0/\alpha]$$

The thesis follows the conclusion of rule [rec], using the latter case (above) as premise:

$$\frac{A, \alpha : \delta; \delta \vdash S_0\,[\mu\alpha.S_0/\alpha]}{A; \delta \vdash \mu\alpha.S_0\,[\mu\alpha.S_0/\alpha]}\ [\texttt{rec}]$$

Case 2. If $S = \mu\alpha S_0$, then $S' = \mu\alpha S_0'$ with $S_0 \overset{\texttt{unfold}}{\equiv} S_0'$. The only rule applicable to $A; \delta \vdash \mu\alpha.S_0$ is rule [rec]:

$$\frac{A, \alpha : \delta; \delta \vdash S_0}{A; \delta \vdash \mu\alpha.S_0}\ [\texttt{rec}]$$

By the induction hypothesis applied to the premise of the above: $A, \alpha : \delta; \delta \vdash S_0'$. Therefore, we obtain our thesis following the conclusion of rule [rec]:

$$\frac{A, \alpha : \delta; \delta \vdash S_0'}{A; \delta \vdash \mu\alpha.S_0'}\ [\texttt{rec}]$$

Case 3. If $S = \left\{ \Box' l' \langle T' \rangle (\delta', \lambda').S_0' \right\}_{i \in I}$ then $S' = \left\{ \Box' l' \langle T' \rangle (\delta', \lambda').S_0'' \right\}_{i \in I}$ with $S_0' \overset{\texttt{unfold}}{\equiv} S_0''$. (For brevity and clarity we only show for $|I| = 1$.) The only rule applicable to $A; \delta \vdash \left\{ \Box' l' \langle T' \rangle (\delta', \lambda').S_0' \right\}_{i \in I}$ is rule [choice]:

$$\frac{\begin{array}{rl} A;\ \gamma \vdash S_0' \ \wedge\ \delta' \left[\lambda \mapsto 0\right]' \models \gamma & \text{(feasibility)} \\ T' = (\delta'',\ S'') \implies \varnothing;\ \gamma' \vdash S'' \ \wedge\ \delta'' \models \gamma' & \text{(delegation)} \end{array}}{A;\ \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{ \Box' l' \langle T' \rangle (\delta', \lambda').S_0' \right\}_{i \in I}}\ [\texttt{choice}]$$

where $\delta = \downarrow \bigvee_{i \in I} \delta_i = \downarrow \delta'$. (Since $|I| = 1$, we omit the (mixed-choice) premise of rule [choice].) By the induction hypothesis, applied to the (feasibility) premise above:

$$A;\ \gamma \vdash S_0'' \ \wedge\ \delta' \left[\lambda \mapsto 0\right]' \models \gamma$$

We obtain our thesis following the conclusion of rule [choice], when the above is used as the (feasibility) premise:

$$\frac{\begin{array}{rl} A;\ \gamma \vdash S_0'' \ \wedge\ \delta' \left[\lambda \mapsto 0\right]' \models \gamma & \text{(feasibility)} \\ T' = (\delta'',\ S'') \implies \varnothing;\ \gamma' \vdash S'' \ \wedge\ \delta'' \models \gamma' & \text{(delegation)} \end{array}}{A;\ \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{ \Box' l' \langle T' \rangle (\delta', \lambda').S_0'' \right\}_{i \in I}}\ [\texttt{choice}]$$ □

Lemma A.3.4. *If* $S \overset{\texttt{unfold}}{\equiv} S'$, *then* (ν, S) *is* well-formed $\implies$ (ν, S') *is* well-formed.

Proof. By Definition 3.4.2, $\exists \delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash S$. The thesis follows immediately from Lemma A.3.3, since $\varnothing; \delta \vdash S'$. □

Lemma A.3.5. *If* (ν, S) *is* well-formed, *then* $S \overset{\text{unfold}}{\equiv} \texttt{end}$ *or* $S \overset{\text{unfold}}{\equiv} \left\{ \Box_i \, l_i \, \langle T_i \rangle \, (\delta_i, \lambda_i).S_i \right\}_{i \in I}$.

Proof. Since (ν, S) is *well-formed*, by Definition 3.4.2 $\exists \delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash S$. We proceed by induction on the derivation, analysing the structure of S. By Definition A.1.5, the only interesting case is $S = \mu\alpha.S''$ and $S \overset{\text{unfold}}{\equiv} S' \left[\mu\alpha.S'/\alpha\right]$. By rule [rec]:

$$\frac{\alpha : \delta; \delta \vdash S'}{\varnothing; \delta \vdash \mu\alpha.S'} \, [\texttt{rec}]$$

By Lemma A.2.2, $\alpha : \delta; \delta \vdash S' \iff \varnothing; \delta \vdash S' \left[\mu\alpha.S'/\alpha\right]$. Therefore, it holds that $(\nu, S' \left[\mu\alpha.S'/\alpha\right])$ is *well-formed*, and so, we obtain our thesis by the induction hypothesis:

$$(\nu, S' \left[\mu\alpha.S'/\alpha\right]) \text{ is } \textit{well-formed} \implies (\nu, S'') \text{ is } \textit{well-formed}$$

where $S' \overset{\text{unfold}}{\equiv} S''$. Since types are *contractive*, we rule out the possibility of $S' = \mu\alpha_1 \ldots \mu\alpha_n.\alpha_i$, for some $i \leq n$ where $1 < n \in \mathbb{N}$. □

A.4 Live Configurations

Lemma A.4.1. *If* (ν, S) *is* well-formed, *then* (ν, S) *is live.*

Proof. By Definition 3.4.2, $\exists \delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash S$. We proceed by induction on the derivation of $\varnothing; \delta \vdash S$ by the 'well-formedness' rules in Eq. (3.4.1):

Case 1. If rule [choice], then $S = \left\{ \Box_i \, l_i \, \langle T_i \rangle \, (\delta_i, \lambda_i).S_i \right\}_{i \in I}$ and $\delta = \downarrow \bigvee_{i \in I} \delta_i$. By Definition 3.1.1, $\exists t$ such that $\nu + t \models \bigvee_{i \in I} \delta_i$. It follows Definition 3.3.1 that (ν, S) is *fe*. Following Definition A.1.2 we obtain our thesis.

Case 2. If rule [rec], then $S = \mu\alpha.S'$. By Lemma A.3.4, since $(\nu, \mu\alpha.S')$ is *well-formed*, if $S \overset{\text{unfold}}{\equiv} S''$ then (ν, S'') is *well-formed*. By Definition A.1.5, $\mu\alpha.S' \overset{\text{unfold}}{\equiv} S' \left[\mu\alpha.S'/\alpha\right] = S''$. Therefore, since $(\nu, S' \left[\mu\alpha.S'/\alpha\right])$ is *well-formed*, the thesis follows by induction hypothesis on $(\nu, S' \left[\mu\alpha.S'/\alpha\right])$ being live.

Case 3. If rule [end], then $S = \texttt{end}$. By Definition A.1.2, $(\nu, \texttt{end})$ is live.

Case 4. Rule [var] is not applicable following Lemma A.3.2, as $S \neq \alpha$. □

Lemma A.4.2. *If* $(\nu + t, S)$ *is* well-formed, *then* (ν, S) *is* well-formed.

Proof. By Lemma A.3.5, $S \overset{\text{unfold}}{\equiv} \texttt{end}$ or $S \overset{\text{unfold}}{\equiv} \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$. By Lemma A.3.4, together with rules [end] and [choice] we have that $\varnothing; \delta \vdash S$ where $\delta = \texttt{true}$ or $\delta =\downarrow \bigvee_{i \in I} \delta_i$. Obviously, $\nu \models \texttt{true}$. The fact $\nu \models\downarrow \bigvee_{i \in I} \delta_i$ instead follows from the definition of $\downarrow$. □

Lemma A.4.3. *If* $\varnothing; \delta \vdash S$ *and* (ν, S) *is* fe, *then* (ν, S) *is* well-formed.

Proof. By Definition 3.3.1, $\exists t$ such that $(\nu, S) \xrightarrow{t} (\nu + t, S) \xrightarrow{\Box m} (\nu', S')$. We proceed by induction on the last rule applied for the transition $(\nu + t, S) \xrightarrow{\Box m}$ of those in Figure C.2 Eq. (3.3.2):

Case 1. If rule [act], then $S = \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$ and:

$$\frac{\nu'' \models \delta_j \quad m = l_j\, \langle T_j \rangle \quad j \in I}{(\nu'', \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}) \xrightarrow{\Box_j m} (\nu''\, [\lambda_j \mapsto 0]\,, S_j)}\ [\texttt{act}]$$

where $\Box = \Box_j$ and $\nu'' = \nu + t$. Since $\varnothing; \delta \vdash \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$ by hypothesis, by rule [choice] we conclude:

$$A; \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$$

where $A = \varnothing$ and $\delta =\downarrow \bigvee_{i \in I} \delta_i$. By Definition 3.4.2, it remains to show that $\nu \models\downarrow \bigvee_{i \in I} \delta_i$. First we show that $\nu + t \models\downarrow \bigvee_{i \in I} \delta_i$. By the premise of rule [act], $\nu'' \models \delta_j$ (where $\nu'' = \nu + t$). Since $\delta_j \models\downarrow \bigvee_{i \in I} \delta_i$, it follows that $\nu + t \models\downarrow \bigvee_{i \in I} \delta_i$ and therefore, $(\nu + t, \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I})$ is *well-formed*. By the conclusion of rule [choice], the set of constraints $\delta =\downarrow \bigvee_{i \in I} \delta_i$ only require a minimum of *one weak past* constraint to be satisfied for the entire set constraint to be satisfied. Therefore, given that we know $\nu + t \models\downarrow \bigvee_{i \in I} \delta_i$, it follows that $\nu \models\downarrow \bigvee_{i \in I} \delta_i$ must also hold, and we obtain our thesis.

Case 2. If rule [unfold], then $S = \mu\alpha.S''$ and:

$$\frac{(\nu + t,\, S''\,[\mu\alpha.S''/\alpha]) \xrightarrow{\ell} (\nu',\, S')}{(\nu + t,\, \mu\alpha.S'') \xrightarrow{\ell} (\nu',\, S')}\ [\texttt{unfold}]$$

By the induction hypothesis we have that $(\nu + t,\, S''\,[\mu\alpha.S''/\alpha])$ is *well-formed*. Then by Lemma A.3.4 we have that $(\nu + t,\, \mu\alpha.S'')$ is *well-formed*. Finally, by Lemma A.4.2, $(\nu,\, \mu\alpha.S'')$ is *well-formed*, as required. □

Lemma A.4.4. *If* $(\nu,\, S) \xrightarrow{\square m}$, *then* $(\nu,\, \overline{S}) \xrightarrow{\overline{\square} m}$.

Proof. We proceed by induction on the last rule applied for the transition $(\nu,\, S) \xrightarrow{\square m}$ of those in Figure C.2 Eq. (3.3.2):

Case 1. If rule [act], then $S = \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$ and:

$$\frac{\nu \models \delta_j \quad m = l_j\, \langle T_j \rangle \quad j \in I}{(\nu,\, \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}) \xrightarrow{\square_j m} (\nu\, [\lambda_j \mapsto 0]\,,\, S_j)}\ [\texttt{act}]$$

where $\square = \square_j$. By Definition 3.4.1, $\overline{S} = \left\{\overline{\square}_i\, l_i \langle T_i \rangle (\delta_i, \lambda_i).\overline{S}_i\right\}_{i \in I}$. Therefore, as the preconditions (δ_i) in S and $\overline{S}$ are identical, it follows that the premise of rule [act] shown above is equally applicable to $(\nu,\, \overline{S})$. With the only difference being the interactions having opposite directions (sending or receiving), we obtain our thesis.

Case 2. If rule [unfold], then $S = \mu\alpha.S''$ and:

$$\frac{(\nu,\, S''\,[\mu\alpha.S''/\alpha]) \xrightarrow{\ell} (\nu',\, S')}{(\nu,\, \mu\alpha.S'') \xrightarrow{\ell} (\nu',\, S')}\ [\texttt{unfold}]$$

where $\ell = \square m$. By Definition 3.4.1, $\overline{S} = \mu\alpha.\overline{S''}$. The thesis follows by induction:

$$(\nu,\, S''\,[\mu\alpha.S''/\alpha]) \xrightarrow{\square m} \quad \Longrightarrow \quad (\nu,\, \overline{S''}\,[\mu\alpha.\overline{S''}/\alpha]) \xrightarrow{\overline{\square} m}$$ □

A.5 Configuration Transitions

Lemma A.5.1. *Let* (ν, S) *be* well-formed. *Claims A.5.2 and A.5.3 both hold.*

Claim A.5.2. $(\nu, S) \xrightarrow{t} (\nu', S') \implies \nu' = \nu + t \wedge S \overset{\texttt{unfold}}{\equiv} S'$

Claim A.5.3. $(\nu, S) \xrightarrow{\Box m} (\nu', S') \implies \exists \delta \text{ s.t.} \varnothing; \delta \vdash S \wedge \nu \models \delta$
$\wedge \exists \delta' \text{ s.t.} \delta' = \delta[\lambda \mapsto 0] \wedge \nu' = \nu[\lambda \mapsto 0]$
$\wedge \exists \gamma \text{ s.t.} \nu' \models \delta' \models \gamma \wedge \varnothing; \gamma \vdash S'$

Proof. We proceed by addressing each claim in turn:

Claim A.5.2 We proceed by induction on the derivation of the transition $(\nu, S) \xrightarrow{t} (\nu', S')$ analysing the last rule applied of those given in Figure C.2 Eq. (3.3.2):

Case 1. If rule [tick], then the thesis holds as $\nu' = \nu + t$ and $S = S'$.

Case 2. If rule [unfold], then $S = \mu\alpha.S''$ and:

$$\frac{(\nu, S''[\mu\alpha.S''/\alpha]) \xrightarrow{\ell} (\nu', S')}{(\nu, \mu\alpha.S'') \xrightarrow{\ell} (\nu', S')}\ [\texttt{unfold}]$$

where $\ell = t$. By Definition A.1.5, $\mu\alpha.S'' \overset{\texttt{unfold}}{\equiv} S''[\mu\alpha.S''/\alpha]$. By Lemma A.3.4, $(\nu, S''[\mu\alpha.S''/\alpha])$ is *well-formed*. By the induction hypothesis, $S''[\mu\alpha.S''/\alpha] \overset{\texttt{unfold}}{\equiv} S'$. By the transitivity equation in Definition A.1.5, $\mu\alpha.S'' \overset{\texttt{unfold}}{\equiv} S'$, as required.

Claim A.5.3 We proceed by induction on the derivation of the transition $(\nu, S) \xrightarrow{\Box m} (\nu', S')$, analysing the last rule applied of those given in Figure C.2 Eq. (3.3.2):

Case 1. If rule [act], then $S = \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$ and:

$$\frac{\nu \models \delta_j \quad m = l_j \langle T_j \rangle \quad j \in I}{(\nu, \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}) \xrightarrow{\Box_j m} (\nu\left[\lambda_j \mapsto 0\right], S_j)}\ [\texttt{act}]$$

where $\nu' = \nu\left[\lambda_j \mapsto 0\right]$, $S' = S_j$ and $\delta = \delta_j$. Since (ν, S) is *well-formed*, by Definition 3.4.2, $\exists\delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash S$. By rule [choice]:

$$\frac{\begin{array}{ll}\forall i \in I : A;\ \gamma_i \vdash S_i \ \wedge\ \delta_i\left[\lambda_i \mapsto 0\right] \models \gamma_i & \text{(feasibility)}\\ \forall i, j \in I : i \neq j \implies \delta_i \wedge \delta_j \models \texttt{false} \vee \square_i = \square_j & \text{(mixed-choice)}\\ \forall i \in I : T_i = (\delta'',\ S'') \implies A;\ \gamma' \vdash S'' \ \wedge\ \delta'' \models \gamma' & \text{(delegation)}\end{array}}{A;\downarrow \bigvee_{i\in I} \delta_i \vdash \left\{\square_i\, l_i\, \langle T_i\rangle\, (\delta_i, \lambda_i).S_i\right\}_{i\in I}}\ [\texttt{choice}]$$

where $A = \varnothing$. As in Case 1 of Lemma A.4.3, the set of constraints $\downarrow \bigvee_{i\in I} \delta_i$ requires as minimum the *weak past* of only *one* set of constraints (δ_j) to be satisfied for the entire set of constraints to be satisfied; hence $\delta_j \models\downarrow \bigvee_{i\in I} \delta_i$. It remains to show: $\nu\left[\lambda_j \mapsto 0\right] \models \delta_j\left[\lambda_j \mapsto 0\right] \models \gamma$ and $\varnothing; \gamma \vdash S_j$. Clearly, by the (feasibility) premise of rule [choice] it holds that $\delta_j\left[\lambda_j \mapsto 0\right] \models \gamma$ and $\varnothing; \gamma \vdash S_j$ (as $A = \varnothing$). By the premise of rule [act], it holds that $\nu \models \delta_j$, and therefore it follows that $\nu\left[\lambda_j \mapsto 0\right] \models \delta_j\left[\lambda_j \mapsto 0\right]$.

Case 2. If rule [unfold], then $S = \mu\alpha.S''$ and:

$$\frac{(\nu,\ S''\left[\mu\alpha.S''/\alpha\right]) \xrightarrow{\ell} (\nu',\ S')}{(\nu,\ \mu\alpha.S'') \xrightarrow{\ell} (\nu',\ S')}\ [\texttt{unfold}]$$

where $\ell = \square m$. The rest follows Case 2 of Claim A.5.2 (above). □

Lemma A.5.4. *Let* (ν, S) *be* well-formed. *Claims A.5.5 to A.5.8 all hold.*

Claim A.5.5. $(\nu, S, \mathtt{M}) \xrightarrow{!m} (\nu', S', \mathtt{M}') \implies \mathtt{M}' = \mathtt{M} \wedge (\nu, S) \xrightarrow{!m} (\nu', S')$

Claim A.5.6. $(\nu, S, \mathtt{M}) \xrightarrow{\tau} (\nu', S', \mathtt{M}') \implies \exists m \text{ s.t.} \mathtt{M} = m; \mathtt{M}' \wedge (\nu, S) \xrightarrow{?m} (\nu', S')$

Claim A.5.7. $(\nu, S, \mathtt{M}) \xrightarrow{?m} (\nu', S', \mathtt{M}') \implies S' = S \wedge \mathtt{M}' = \mathtt{M}; m \wedge \nu' = \nu$

Claim A.5.8. $(\nu, S, \mathtt{M}) \xrightarrow{t} (\nu', S', \mathtt{M}') \implies S' \overset{\texttt{unfold}}{\equiv} S \wedge \mathtt{M}' = \mathtt{M} \wedge \nu' = \nu + t$

Proof. We proceed addressing each claim in turn, using the rules in Eq. (3.3.3) in Figure C.2:

Claim A.5.5 By rule [send] the claim holds. By the premise $(\nu, S) \xrightarrow{!m} (\nu', S')$.

Claim A.5.6 By rule [recv] the claim holds. By the premise $(\nu, S) \xrightarrow{?m} (\nu', S')$.

Claim A.5.7 By rule [que] the claim holds.

Claim A.5.8 By rule [time] the claim holds. By the premise $\nu' = \nu + t$ via rule [tick].

(See Claim A.5.2 of Lemma A.5.1.) □

Lemma A.5.9. *Let* (ν, S) *be* well-formed. *If* $(\nu, S) \xrightarrow{\Box m}$ *and* $(\nu, S) \xrightarrow{\Box' m'}$ *then* $\Box = \Box'$.

Proof. We proceed by induction on the derivation of the transition $(\nu, S) \xrightarrow{\Box m}$, analysing the last rule applied of those in Figure C.2 Eq. (3.3.2). We only show the case of rule [act], as the only other applicable case (rule [unfold]) follows by induction hypothesis as in Claims A.5.2 and A.5.3 of Lemma A.5.1. Therefore, if rule [act], then $S = \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$ and:

$$\frac{\nu \models \delta_j \quad m = l_j\, \langle T_j \rangle \quad j \in I}{(\nu, \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}) \xrightarrow{\Box_j m} (\nu\, [\lambda_j \mapsto 0]\,, S_j)}\ [\texttt{act}] \tag{A.5.1}$$

where $\Box = \Box_j$ and $m = l_j\, \langle T_j \rangle$. Since (ν, S) is *well-formed*, by Definition 3.4.2, $\exists \delta$ such that $\nu \models \delta$ and $\varnothing; \delta \vdash \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}$. The only possible rule is [choice]:

$$\frac{\begin{array}{rl} \forall i \in I : A;\ \gamma_i \vdash S_i \ \wedge\ \delta_i\, [\lambda_i \mapsto 0] \models \gamma_i & \text{(feasibility)} \\ \forall i, j \in I : i \neq j \implies \delta_i \wedge \delta_j \models \texttt{false} \vee \Box_i = \Box_j & \text{(mixed-choice)} \\ \forall i \in I : T_i = (\delta'',\ S'') \implies \varnothing;\ \gamma' \vdash S'' \ \wedge\ \delta'' \models \gamma' & \text{(delegation)} \end{array}}{A;\ \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{\Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}}\ [\texttt{choice}]$$

As in Lemma A.4.3, it follows that $\delta_j \models \downarrow \bigvee_{i \in I} \delta_i$. Since $(\nu, S) \xrightarrow{\Box' m'}$, we proceed by induction on the derivation of the transition, analysing the last rule applied. Again, we omit the case by rule [unfold]. By rule [act], it follows similarly to Eq. (A.5.1): for some $k \in I$, $\Box' = \Box_k$, $m = l_k\, \langle T_k \rangle$ and $\nu \models \delta_k$. In the case where $j = k$, then the thesis coincides with the hypothesis. Otherwise, if $j \neq k$, then by the (mixed-choice) premise of rule [choice], $\delta_j \wedge \delta_k \models \texttt{false}$ or $\Box_j = \Box_k$. Since $\nu \models \delta_j$ and $\nu \models \delta_k$, it follows that $\delta_j \wedge \delta_k \not\models \texttt{false}$. Therefore, it must be that $\Box_j = \Box_k$ and we obtain our thesis. □

A.6 System Transition Preservations

Lemma A.6.1. *Let* (ν_1, S_1) *and* (ν_2, S_2) *be* well-formed, *and* $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$. *If* $\mathbf{S}_1 \mid \mathbf{S}_2 \to (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')$, *then* (ν_1', S_1') *and* (ν_2', S_2') *are* well-formed.

Proof. We proceed by cases on the rule used in the derivation of the transition:

$$(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \to (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')$$

analysing the last rule applied of those given in Figure C.2 Eq. (3.3.4):

Case 1. If rule [`wait`], then:

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1') \quad (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu_2', S_2', \mathtt{M}_2')}{(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')}\ [\mathtt{wait}]$$

Hereafter, we only show the case for $(\nu_1, S_1, \mathtt{M}_1)$, as the transition for $(\nu_2, S_2, \mathtt{M}_2)$ is analogous. We proceed by inner induction on the derivation of the transition $(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1')$ analysing the last rule applied. By rule [`time`]:

$$\frac{\begin{array}{rl} (\nu_1, S_1) \xrightarrow{t} (\nu_1', S_1') & \text{(configuration)} \\ (\nu_1, S_1) \text{ is } \mathtt{fe} \implies (\nu_1', S_1') \text{ is } \mathtt{fe} & \text{(persistency)} \\ \forall t' < t : (\nu_1 + t', S_1, \mathtt{M}_1) \not\xrightarrow{\tau} & \text{(urgency)} \end{array}}{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1)}\ [\mathtt{time}]$$

By inner induction on the derivation of the transition $(\nu_1, S_1) \xrightarrow{t} (\nu_1', S_1')$ analysing the last rule applied of those in Figure C.2 Eq. (3.3.2):

i. If rule [`tick`], then by Claim A.5.2 of Lemma A.5.1, $\nu_1' = \nu_1 + t$, $\mathtt{M}_1' = \mathtt{M}_1$ and $S_1' \overset{\text{unfold}}{\equiv} S_1$. If $t = 0$ then $\nu_1 = \nu_1 + t$ and therefore, the thesis coincides with the hypothesis. By the (urgency) premise of rule [`time`], t must be valued such that no message can be received from the queue $\mathtt{M}_1$ via rule [`recv`]. If $t = 0$, then $\nexists t' < t$, and the (urgency) premise of rule [`time`] always holds, even if $(\nu_2, S_2, \mathtt{M}_2) \xrightarrow{\tau}$. However, since $\nu_1' = \nu_1$, this is the same as no transition occurring via rule [`time`]. Otherwise, $t > 0$. Since (ν_1, S_1) is *well-formed*, it

follows Lemma A.7.1, (ν_1, S_1) is also *fe*. Since (ν_1, S_1) is *fe*, it follows the (persistency) premise of rule [time] that (ν_1', S_1') is also *fe*. By Lemma A.4.3, $(\nu_1 + t, S_1')$ is *well-formed*.

ii. If rule [unfold], then $S_1 = \mu\alpha.S_1''$ and:

$$\frac{(\nu_1, S_1''\,[\mu\alpha.S_1''/\alpha]) \xrightarrow{\ell} (\nu_1', S_1')}{(\nu_1, \mu\alpha.S_1'') \xrightarrow{\ell} (\nu_1', S_1')}\ [\texttt{unfold}]$$

where $\ell = t$. Since $(\nu_1, \mu\alpha.S_1'')$ is *well-formed*, $\exists\delta$ such that $\nu_1 \models \delta$ and $\varnothing; \delta \vdash \mu\alpha.S_1''$. By rule [rec]:

$$\frac{\alpha : \delta; \delta \vdash S_1''\,[\mu\alpha.S_1''/\alpha]}{\varnothing; \delta \vdash \mu\alpha.S_1''}\ [\texttt{rec}]$$

By Lemma A.2.2: $\alpha : \delta; \delta \vdash S_1''\,[\mu\alpha.S_1''/\alpha] \iff \varnothing; \delta \vdash S_1''$. By Definition A.1.5, it follows $\mu\alpha.S_1'' \overset{\text{unfold}}{\equiv} S_1''\,[\mu\alpha.S_1''/\alpha]$. By Lemma A.3.3, since $\varnothing; \delta \vdash S_1''$, then $\varnothing; \delta \vdash \mu\alpha.S_1''$. By Lemma A.3.4, if $(\nu_1, S_1''\,[\mu\alpha.S_1''/\alpha])$ is *well-formed*, then $(\nu_1, \mu\alpha.S_1'')$ is *well-formed*. The thesis holds by the induction hypothesis. (As in Case 2 in Lemma A.4.3.)

Case 2. If rule [com-l], then:

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{!m} (\nu_1', S_1', \mathtt{M}_1') \quad (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{?m} (\nu_2', S_2', \mathtt{M}_2')}{(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{\tau} (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')}\ [\texttt{com-l}]$$

First, we proceed by cases on the derivation of the transition $(\nu_2, S_2, \mathtt{M}_2) \xrightarrow{?m} (\nu_2', S_2', \mathtt{M}_2')$ analysing the last rule applied. By rule [que]:

$$(\nu_2, S_2, \mathtt{M}_2) \xrightarrow{?m} (\nu_2, S_2, \mathtt{M}_2; m) \quad [\texttt{que}]$$

where, as in Claim A.5.7 of Lemma A.5.4, $\nu_2' = \nu_2$, $S_2' = S_2$ and $\mathtt{M}_2' = \mathtt{M}_2; m$. Therefore, it follows that (ν_2', S_2') is *well-formed*. Next, by cases on the derivation of the transition $(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{!m} (\nu_1', S_1', \mathtt{M}_1')$ analysing the last rule applied. By rule [send]:

$$\frac{(\nu_1, S_1) \xrightarrow{!m} (\nu_1', S_1')}{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{!m} (\nu_1', S_1', \mathtt{M}_1')}\ [\texttt{send}]$$

where, as in Claim A.5.5 of Lemma A.5.4, $\mathtt{M}'_1 = \mathtt{M}_1$ and $(\nu_1, S_1) \xrightarrow{!m} (\nu'_1, S'_1)$. As in Claims A.5.2 and A.5.3 of Lemma A.5.1, we proceed to only show the case of rule [act] (as the only other applicable rule is rule [unfold], which follows by induction). Therefore, $S_1 = \left\{ \Box_i \, l_i \langle T_i \rangle \, (\delta_i, \lambda_i).S_i \right\}_{i \in I}$ and for some $j \in I$, $\nu_1 \models \delta_j$, $m = l_j \langle T_j \rangle$, $! = \Box_j$, $\nu'_1 = \nu_1 \left[\lambda_j \mapsto 0\right]$ and $S'_1 = S_j$. It remains to show that $(\nu_1 \left[\lambda_j \mapsto 0\right], S_j)$ is *well-formed*. The thesis follows Claim A.5.3 of Lemma A.5.1, since $\nu_1 \left[\lambda_j \mapsto 0\right] \models \delta_j \left[\lambda_j \mapsto 0\right] \models \gamma_j$ and $\varnothing; \gamma_j \vdash S_j$, for some $j \in I$. By Definition 3.4.2, $(\nu_1 \left[\lambda_j \mapsto 0\right], S_j)$ is *well-formed*.

Case 3. If rule [par-l], then:

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{\tau} (\nu'_1, S'_1, \mathtt{M}'_1)}{(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{\tau} (\nu'_1, S'_1, \mathtt{M}'_1) \mid (\nu_2, S_2, \mathtt{M}_2)} \; [\mathtt{par\text{-}l}]$$

where, $\nu'_2 = \nu_2$, $S'_2 = S_2$ and $\mathtt{M}'_2 = \mathtt{M}_2$. It holds that (ν'_2, S'_2) remains *well-formed*. We proceed by inner induction on the derivation of the transition $(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{\tau} (\nu'_1, S'_1, \mathtt{M}'_1)$ analysing the last rule applied of those in Figure C.2 Eq. (3.3.3). By rule [recv]:

$$\frac{(\nu_1, S_1) \xrightarrow{?m} (\nu'_1, S'_1)}{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{\tau} (\nu'_1, S'_1, \mathtt{M}'_1)} \; [\mathtt{recv}]$$

As in Case 2, the thesis follows Claim A.5.3 of Lemma A.5.1 (except $? = \Box_j$).

Case 4. Rules [com-r] and [par-r] are analogous to Cases 2 and 3 respectively. □

Lemma A.6.2. *Let* (ν_1, S_1) *and* (ν_2, S_2) *be* well-formed, *and* $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$. *If* $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu'_1, S'_1, \mathtt{M}'_1) \mid (\nu'_2, S'_2, \mathtt{M}'_2)$ *and* $t > 0$, *then* $\mathtt{M}_1 = \varnothing = \mathtt{M}_2$.

Proof. We proceed by cases on the derivation of the transition $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t}$ analysing the last rule applied of those in Figure C.2 Eq. (3.3.4). By rule [wait]:

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu'_1, S'_1, \mathtt{M}'_1) \quad (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu'_2, S'_2, \mathtt{M}'_2)}{(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu'_1, S'_1, \mathtt{M}'_1) \mid (\nu'_2, S'_2, \mathtt{M}'_2)} \; [\mathtt{wait}]$$

where, as in Claim A.5.8 of Lemma A.5.4, $\mathtt{M}'_1 = \mathtt{M}_1$ and $\mathtt{M}'_2 = \mathtt{M}_2$. Hereafter, we only show $(\nu_1, S_1, \mathtt{M}_1)$ as $(\nu_2, S_2, \mathtt{M}_2)$ is analogous. By cases on the derivation of the

transition $(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1')$ analysing the last rule applied. By rule [`time`]:

$$\frac{(\nu_1, S_1) \xrightarrow{t} (\nu_1', S_1') \qquad \text{(configuration)} \qquad (\nu_1, S_1) \text{ is } \mathtt{fe} \implies (\nu_1', S_1') \text{ is } \mathtt{fe} \qquad \text{(persistency)} \qquad \forall t' < t : (\nu_1 + t', S_1, \mathtt{M}_1) \xrightarrow{\tau} \not\rightarrow \qquad \text{(urgency)}}{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1)}\ [\mathtt{time}]$$

where, as in Claim A.5.8 of Lemma A.5.4, $\nu_1' = \nu_1 + t$ and $S_1' \overset{\text{unfold}}{\equiv} S_1$. Suppose by contradiction that one of the queues $\mathtt{M}_1$ were *non-empty*, and $\mathtt{M}_1 = m; \mathtt{M}_1''$. Since $(\nu_1, S_1, m; \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$, by Condition (2) in Definition 3.4.6:

$$\mathtt{M}_1 = m; \mathtt{M}_1' \implies (\nu_1, S_1) \xrightarrow{?m} (\nu_1', S_1') \wedge (\nu_1', S_1', \mathtt{M}_1') \perp (\nu_2, S_2, \mathtt{M}_2)$$

Such a transition is only viable by rule [`recv`]. By Claim A.5.6 of Lemma A.5.4, it follows that $(\nu_1, S_1) \xrightarrow{?m} (\nu_1', S_1')$ is made by the premise of rule [`recv`]. However, by the (urgency) premise of rule [`time`], $\forall t' < t$ no transitions by rule [`recv`] are viable – which contradicts with Condition (2) in Definition 3.4.6. Therefore, it cannot be that either queue is *non-empty* for a transition t, where $t > 0$. For such a transition, both queues must be *empty*. □

Lemma A.6.3. *Let* (ν_1, S_1) *and* (ν_2, S_2) *be* well-formed*, and* $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$*. If* $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \to (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')$*, then* $(\nu_1', S_1', \mathtt{M}_1') \perp (\nu_2', S_2', \mathtt{M}_2')$*.*

Proof. We proceed by cases on the derivation of the transition:

$$(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \to (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')$$

analysing the last rule applied of those given in Figure C.2 Eq. (3.3.4):

Case 1. If rule [`wait`], then:

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1') \quad (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu_2', S_2', \mathtt{M}_2')}{(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t} (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')}\ [\mathtt{wait}]$$

By Claim A.5.8 of Lemma A.5.4, it follows that $\nu_1' = \nu_1 + t$, $S_1' \overset{\text{unfold}}{\equiv} S_1$ and $\mathtt{M}_1' = \mathtt{M}_1$ (and similarly for ν_2', S_2' and $\mathtt{M}_2'$). If $t = 0$ then $\nu_1 = \nu_1 + t$ (and $\nu_2 = \nu_2 + t$) and therefore, the thesis coincides with the hypothesis. Else $t > 0$, and by Lemma A.6.2, $\mathtt{M}_1 = \varnothing = \mathtt{M}_2$. By Condition (4) of Definition 3.4.6: $\mathtt{M}_1 = \varnothing = \mathtt{M}_2 \implies S_1 = \overline{S_2} \wedge \nu_1 = \nu_2$. Since $\nu_1 + t = \nu_2 + t$, it follows that $(\nu_1 + t, S_1', \mathtt{M}_1) \perp (\nu_2 + t, S_2', \mathtt{M}_2)$.

Case 2. If rule [com-l], then:

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{!m} (\nu_1', S_1', \mathtt{M}_1') \quad (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{?m} (\nu_2', S_2', \mathtt{M}_2')}{(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{\tau} (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')}\ [\mathtt{com\text{-}l}]$$

First, by induction on the derivation of the transition $(\nu_2, S_2, \mathtt{M}_2) \xrightarrow{?m} (\nu_2', S_2', \mathtt{M}_2')$ analysing the last rule applied of those in Figure C.2 Eq. (3.3.3). By rule [que]:

$$(\nu_2, S_2, \mathtt{M}_2) \xrightarrow{?m} (\nu_2, S_2, \mathtt{M}_2; m) \quad [\mathtt{que}]$$

where $\nu_2' = \nu_2$, $S_2' = S_2$ and $\mathtt{M}_2' = \mathtt{M}_2; m$. Next, by induction on the derivation of the transition $(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{!m} (\nu_1', S_1', \mathtt{M}_1')$ analysing the last rile applied. By rule [send]:

$$\frac{(\nu_1, S_1) \xrightarrow{!m} (\nu_1', S_1')}{(\nu_1, S_1, \mathtt{M}_1) \xrightarrow{!m} (\nu_1', S_1', \mathtt{M}_1)}\ [\mathtt{send}]$$

where $\mathtt{M}_1' = \mathtt{M}_1$. By inner induction on the derivation of the transition $(\nu_1, S_1) \xrightarrow{!m} (\nu_1', S_1')$ analysing the last rule applied of those in Figure C.2 Eq. (3.3.2). We omit the case of rule [unfold], since following Lemma A.3.5, $S_1 \overset{\text{unfold}}{\equiv} \left\{ \Box_i\, l_i \left\langle T_i \right\rangle (\delta_i, \lambda_i).S_i \right\}_{i \in I}$, and by Lemma A.3.3, $(\nu_1, \left\{ \Box_i\, l_i \left\langle T_i \right\rangle (\delta_i, \lambda_i).S_i \right\}_{i \in I})$ is *well-formed*. Therefore, by rule [act]:

$$\frac{\nu_1 \models \delta_j \quad m = l_j \left\langle T_j \right\rangle \quad j \in I}{(\nu_1, \left\{ \Box_i\, l_i \left\langle T_i \right\rangle (\delta_i, \lambda_i).S_i \right\}_{i \in I}) \xrightarrow{\Box_j m} (\nu_1 \left[\lambda_j \mapsto 0\right], S_j)}\ [\mathtt{act}]$$

where $\nu_1' = \nu_1 \left[\lambda_j \mapsto 0\right]$, $S_1' = S_j$ and $m = l_j \left\langle T_j \right\rangle$, for some $j \in I$ such that $\nu_1 \models \delta_j$. If $\mathtt{M}_1$ were *non-empty* (i.e.: $\mathtt{M}_1 = m'; \mathtt{M}_1''$), then by Condition (2) of Definition 3.4.6, it must be that $(\nu_1, S_1) \xrightarrow{?m'}$. However, by Lemma A.5.9, this is plainly not the case since $(\nu_1, S_1) \xrightarrow{!m}$. Therefore, $\mathtt{M}_1 = \varnothing$. We proceed by case analysis on the contents

of $\mathtt{M}_2$:

i. If $\mathtt{M}_2 = \varnothing$, then by Condition (4) of Definition 3.4.6, $S_1 = \overline{S_2}$ and $\nu_1 = \nu_2$. Therefore, by the induction hypothesis:

$$(\nu_1, \left\{\Box_i\, l_i\, \langle T_i\rangle\, (\delta_i, \lambda_i).S_i\right\}_{i\in I}, \varnothing) \mid (\nu_2, \left\{\overline{\Box}_i\, l_i\langle T_i\rangle(\delta_i, \lambda_i).\overline{S}_i\right\}_{i\in I}, \varnothing) \xrightarrow{\tau} (\nu_1\left[\lambda_j \mapsto 0\right], S_j, \varnothing) \mid (\nu_2, \left\{\overline{\Box}_i\, l_i\langle T_i\rangle(\delta_i, \lambda_i).\overline{S}_i\right\}_{i\in I}, m)$$

It remains to show that: $(\nu_1\left[\lambda_j \mapsto 0\right], S_j, \varnothing) \perp (\nu_2, \left\{\overline{\Box}_i\, l_i\langle T_i\rangle(\delta_i, \lambda_i).\overline{S}_i\right\}_{i\in I}, m)$. Since $\mathtt{M}_2' = m;\mathtt{M}_2''$, then by Condition (3) of Definition 3.4.6:

$$(\nu_2, \left\{\overline{\Box}_i\, l_i\langle T_i\rangle(\delta_i, \lambda_i).\overline{S}_i\right\}_{i\in I}) \xrightarrow{?m} (\nu_2'', S_2'') \text{ and } (\nu_1\left[\lambda_j \mapsto 0\right], S_j, \varnothing) \perp (\nu_2'', S_2'', \mathtt{M}_2'') \tag{A.6.1}$$

The thesis follows Lemma A.4.4, since $(\nu_1, S_1) \xrightarrow{!m}$ and $\nu_1 = \nu_2$, then $(\nu_2, \overline{S_2}) \xrightarrow{?m}$.

ii. If $\mathtt{M}_2 = m';\mathtt{M}_2''$, then $\mathtt{M}_2' = m';\mathtt{M}_2'';m$ and therefore, by Condition (3) of Definition 3.4.6, it follows Eq. (A.6.1). The thesis follows by the induction hypothesis:

$$(\nu_1, \left\{\Box_i\, l_i\, \langle T_i\rangle\, (\delta_i, \lambda_i).S_i\right\}_{i\in I}, \varnothing) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{\tau} (\nu_1\left[\lambda_j \mapsto 0\right], S_j, \varnothing) \mid (\nu_2, S_2, m';\mathtt{M}_2'';m)$$

If $\mathtt{M}_2'' = \varnothing$, then it follows Case 2.i. Otherwise, it follows this case.

Case 3. If rule [`par-l`], then by Claim A.5.6 of Lemma A.5.4 it follows that $\mathtt{M}_1 = m;\mathtt{M}_1'$, and by Condition (2) in Definition 3.4.6, we obtain our thesis.

Case 4. If rule [`com-r`] or rule [`par-r`], then it follows Cases 2 and 3 respectively. □

A.7 System Progress

Lemma A.7.1. *Let* (ν_1, S_1) *and* (ν_2, S_2) *be* well-formed. *If* $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$, *then* $(\nu_1, S_1, \mathtt{M}_1)$ *and* $(\nu_2, S_2, \mathtt{M}_2)$ *are* final, *or* $\exists t : (\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \xrightarrow{t\ \tau}$.

Proof. Since both (ν_1, S_1) and (ν_2, S_2) are *well-formed*, by Lemma A.4.1, both (ν_1, S_1) and (ν_2, S_2) are live. By Definition 3.4.4, $(\nu_1, S_1, \mathtt{M}_1)$ is *final* if $S_1 \overset{\text{unfold}}{\equiv} \mathtt{end}$ (and similarly for S_2). By Definition A.1.2 if $S_1 \overset{\text{unfold}}{\not\equiv} \mathtt{end}$, then (ν_1, S_1) is *fe* (and similarly for (ν_2, S_2)). By Definition 3.3.1, $\exists t'$ such that $(\nu_1, S_1) \xrightarrow{t'\ \Box m}$ (and similarly for (ν_2, S_2)).

Therefore, we have obtained our thesis as it holds that if, $(\nu_1, S_1, \mathtt{M}_1)$ is *not* final the by Definition A.1.2, it *must* be *fe* by Definition 3.3.1, which coincides with the thesis. □

Lemma A.7.2. *If* (ν_1, S_1) *and* (ν_2, S_2) *are* well-formed, *and* $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ *and* $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \longrightarrow^* (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')$, *then* $(\nu_1', S_1', \mathtt{M}_1') \perp (\nu_2', S_2', \mathtt{M}_2')$ *and* (ν_1', S_1') *and* (ν_2', S_2') *are* well-formed.

Proof. We proceed by induction on the length of $(\longrightarrow^*)$. The base case is trivial, as $(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \longrightarrow^0 (\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2)$ and hence the thesis coincides with the hypothesis. For the induction case, suppose:

$$(\nu_1, S_1, \mathtt{M}_1) \mid (\nu_2, S_2, \mathtt{M}_2) \longrightarrow^n (\nu_1'', S_1'', \mathtt{M}_1'') \mid (\nu_2'', S_2'', \mathtt{M}_2'') \longrightarrow (\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2')$$

By the induction hypothesis: $(\nu_1'', S_1'', \mathtt{M}_1'') \perp (\nu_2'', S_2'', \mathtt{M}_2'')$, and both (ν_1'', S_1'') and (ν_2'', S_2'') are *well-formed*. It remains for us to show that both (ν_1', S_1') and (ν_2', S_2') are *well-formed*, and $(\nu_1', S_1', \mathtt{M}_1') \perp (\nu_2', S_2', \mathtt{M}_2')$. By Lemmas A.6.1 and A.6.3, we obtain our thesis. □

Lemma A.7.3. *The following holds:*

$$\forall \nu, S : (\nu, S) \textit{ is } \text{well-formed} \implies (\nu, S, \varnothing) \mid (\nu, \overline{S}, \varnothing) \textit{ satisfies progress.}$$

Proof. Since (ν, S) is *well-formed*, by Definition 3.4.1 $(\nu, \overline{S})$ is *well-formed*. By Definition 3.4.6, $(\nu, S, \varnothing) \perp (\nu, \overline{S}, \varnothing)$. If $S = \mathtt{end}$ then $\overline{S} = \mathtt{end}$ and therefore, by Definition 3.4.4, the thesis coincides with the hypothesis. Otherwise, $(\nu, S, \varnothing) \mid (\nu, \overline{S}, \varnothing)$ *satisfies progress* if, for all $\mathbf{S}' \mid \mathbf{S}''$ reachable from $(\nu, S, \varnothing) \mid (\nu, \overline{S}, \varnothing)$ then, either $\mathbf{S}'$ and $\mathbf{S}''$ are both *final*, or $\exists t$ such that: $\mathbf{S}' \mid \mathbf{S}'' \xrightarrow{t\ \tau}$. By Lemma A.7.2, any transition made by $(\nu, S, \varnothing) \mid (\nu, \overline{S}, \varnothing)$ will preserve both 'well-formedness' and compatible. By Lemma A.7.1, we obtain our thesis. □

Theorem 3.4.5 (Progress of Systems)**.** *If* S *is* well-formed *against* ν_0, *then:*

$$(\nu_0, S, \varnothing) \mid (\nu_0, \overline{S}, \varnothing) \text{ satisfies progress.}$$

Proof. The thesis follows immediately from Lemma A.7.3. □

As typical for binary systems, our proof of progress builds upon a notion of *duality* between participants. Since communication is asynchronous, each party may break duality with their co-party. Compatibility allows the duality of participants to be broken, only if, doing so does not violate *communication safety*. More specifically, compatibility requires that any message received by a queue is expected (i.e., able to be received), and that the resulting configurations are still compatible. In practice, using compatibility allows each party to behave and progress independently, regardless the state of the other party, while retaining the essence of *duality*, and guaranteeing that all messages will eventually be received.

Appendix B

Proof of Subject Reduction

B.1 Auxiliary Definitions, Figures & Assumptions

Definition B.1.1 (Free-queues of a Process)**.** The function $\mathtt{fq}(P)$ returns the set of *free-queues* in a given process P, defined inductively below:

$$\mathtt{fq}(P) = \begin{cases} \{pq\} & \text{if } P = pq : h \\ \mathtt{fq}(P') \setminus \{pq, qp\} & \text{if } P = (\nu pq)\ P' \\ \mathtt{fq}(P') & \text{if } P \in \left\{ \begin{array}{l} p \lhd l\,(w).P',\ \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ P', \\ \mathtt{set}\,\textcircled{x}.P',\ \mathtt{delay}\,(d)\,.P',\ \mathtt{delay}\,(t)\,.P' \end{array} \right\} \\ \bigcup_{i\in I} \mathtt{fq}(P_i) & \text{if } P = p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I} \\ \mathtt{fq}(P') \cup \mathtt{fq}(Q) & \text{if } P \in \left\{ P' \mid Q,\ \mathtt{if}\ c\ \mathtt{then}\ P'\ \mathtt{else}\ Q \right\} \\ \mathtt{fq}(p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I}) \cup \mathtt{fq}(Q) & \text{if } P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I} \mathtt{after}\ Q \\ \varnothing & \text{if } P = \mathtt{0},\ X\langle\vec{v};\vec{r}\rangle \end{cases}$$

Definition B.1.2 (Free-timers of a Process)**.** The function $\mathtt{ft}(P)$ returns the set of *free timers* in a given process P, defined inductively below:

$$\mathtt{ft}(P) = \begin{cases} \{\textcircled{x}\} \cup \mathtt{ft}(P') & \text{if } P = \mathtt{set}\,\textcircled{x}.P' \\ \{\textcircled{x}\} \cup \mathtt{ft}(P') \cup \mathtt{ft}(Q) & \text{if } P = \mathtt{if}\ c\ \mathtt{then}\ P'\ \mathtt{else}\ Q \text{ and } \textcircled{x} \in c \\ \mathtt{ft}(P') & \text{if } P \in \left\{ \begin{array}{l} p \lhd l\,(w).P',\ \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ P', \\ \mathtt{delay}\,(d)\,.P',\ \mathtt{delay}\,(t)\,.P',\ (\nu pq)\ P' \end{array} \right\} \\ \bigcup_{i\in I} \mathtt{ft}(P_i) & \text{if } P = p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I} \\ \mathtt{ft}(P') \cup \mathtt{ft}(Q) & \text{if } P = P' \mid Q \\ \mathtt{ft}(p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I}) \cup \mathtt{ft}(Q) & \text{if } P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I} \mathtt{after}\ Q \\ \varnothing & \text{if } P = \mathtt{0},\ X\langle\vec{v};\vec{r}\rangle,\ pq : h \end{cases}$$

Definition B.1.3 (Free-names of a Process)**.** The function $\mathtt{fn}(P)$ returns the set of *free-names* in a given process P, defined inductively below:

$$\mathtt{fn}(P) = \begin{cases} \{pq\} \cup \mathtt{fn}(P') & \text{if } P = (\nu pq)\ P' \\ \mathtt{fn}(P') & \text{if } P \in \left\{ \begin{array}{l} p \lhd l\,(w).P',\ \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ P', \\ \mathtt{set}\,\textcircled{x}.P',\ \mathtt{delay}\,(d)\,.P',\ \mathtt{delay}\,(t)\,.P' \end{array} \right\} \\ \bigcup_{i\in I} \mathtt{fn}(P_i) & \text{if } P = p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I} \\ \mathtt{fn}(P') \cup \mathtt{fn}(Q) & \text{if } P \in \left\{ P' \mid Q,\ \mathtt{if}\ c\ \mathtt{then}\ P'\ \mathtt{else}\ Q \right\} \\ \mathtt{fn}(p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I}) \cup \mathtt{fn}(Q) & \text{if } P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i\in I} \mathtt{after}\ Q \\ \varnothing & \text{if } P = \mathtt{0},\ X\langle\vec{v};\vec{r}\rangle,\ pq : h \end{cases}$$

Definition B.1.4 (Free process variables of a Process)**.** The function $\texttt{fpv}(P)$ returns the set of *free process variables* in a given process P, defined inductively below:

$$\texttt{fpv}(P) = \begin{cases} \{X\} & \text{if } P = X\langle \vec{v}; \vec{r} \rangle \\ \{X\} \cup \texttt{fpv}(P') & \text{if } P = \texttt{def } X(\vec{v}; \vec{r}) = P \texttt{ in } P' \\ \texttt{fpv}(P') & \text{if } P \in \left\{ \begin{array}{l} p \lhd l\,(w).P',\ (\nu pq)\ P',\ \texttt{set}\,\textcircled{x}.P', \\ \texttt{delay}\,(d)\,.P',\ \texttt{delay}\,(t)\,.P' \end{array} \right\} \\ \bigcup_{i \in I} \texttt{fpv}(P_i) & \text{if } P = p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \\ \texttt{fpv}(P') \cup \texttt{fpv}(Q) & \text{if } P \in \left\{ P' \mid Q,\ \texttt{if } c \texttt{ then } P' \texttt{ else } Q \right\} \\ \texttt{fpv}(p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}) \cup \texttt{fpv}(Q) & \text{if } P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{ after } Q \\ \varnothing & \text{if } P = \texttt{0},\ pq : h \end{cases}$$

B.1.1 Session Environments

Definition B.1.5 (Live Δ)**.** Δ is live if, for all $p \in \text{Dom}(\Delta)$ such that $\Delta = \Delta', p : (\nu, S) \implies (\nu, S)$ is live by Definition A.1.2.

Definition B.1.6 (Delayable Δ)**.** A session environment Δ is delayable if:

$$\forall qp \in \text{Dom}(\Delta) : \Delta(qp) \neq \varnothing \implies p \notin \text{Dom}(\Delta)$$

B.1.2 Session Environment Reduction

See Figure B.1.

$$\frac{\Delta_1 \longrightarrow \Delta_1'}{\Delta_1,\ \Delta_2 \longrightarrow \Delta_1',\ \Delta_2}\ [\Delta\text{-L}] \qquad \frac{(\nu, S, l\,\langle T \rangle\,;\texttt{M}) \xrightarrow{\tau} (\nu', S', \texttt{M})}{p : (\nu, S),\ qp : l\,\langle T \rangle\,;\texttt{M} \longrightarrow p : (\nu', S'),\ qp : \texttt{M}}\ [\Delta\text{-Recv}]$$

$$\frac{\Delta_2 \longrightarrow \Delta_2'}{\Delta_1,\ \Delta_2 \longrightarrow \Delta_1,\ \Delta_2'}\ [\Delta\text{-R}] \qquad \frac{(\nu, S) \xrightarrow{!m} (\nu', S')}{p : (\nu, S),\ pq : \texttt{M}, \longrightarrow p : (\nu', S'),\ pq : \texttt{M}; m}\ [\Delta\text{-Send}]$$

FIGURE B.1: Reduction for Session Environments

B.2 Delayable Processes

Lemma B.2.1. *If* $\Phi_t(P)$ *is defined, then* $\texttt{Wait}(P) \cap \texttt{NEQ}(P) = \varnothing$.

Proof. The hypothesis holds by the definition of $\Phi_t(P)$ in Definition 4.2.2. □

B.3 Well-formed Δ

Lemma B.3.1. *Let* Δ *be* well-formed. *If* Δ *is balanced and* $\Delta \longrightarrow \Delta'$, *then* Δ' *is balanced. Furthermore, if* Δ *is fully-balanced, then* Δ' *is fully-balanced.*

Proof. We proceed by induction on the derivation of the reduction $\Delta \longrightarrow \Delta'$, analysing the last rule applied of those given in Figure B.1:

Case 1. If rule [Δ-Send], then $\Delta = p : (\nu, S), pq : \texttt{M}$ and:

$$\frac{(\nu, S) \xrightarrow{!l\langle T\rangle} (\nu', S')}{p : (\nu, S), pq : \texttt{M} \longrightarrow p : (\nu', S'), pq : \texttt{M}; l\,\langle T\rangle}\ [\Delta\text{-Send}]$$

We have that $p : (\nu', S'), pq : \texttt{M}; l\,\langle T\rangle$ is balanced, as every condition of Definition 5.4.1 holds trivially. Regarding the furthermore part, it holds trivially as Δ is not fully-balanced.

Case 2. If rule [Δ-Recv], then $\Delta = p : (\nu, S), qp : l\,\langle T\rangle\,;\texttt{M}$ and:

$$\frac{(\nu, S, l\,\langle T\rangle\,;\texttt{M}) \xrightarrow{?l\langle T\rangle} (\nu', S', \texttt{M}')}{p : (\nu, S), qp : l\,\langle T\rangle\,;\texttt{M} \longrightarrow p : (\nu', S'), qp : \texttt{M}}\ [\Delta\text{-Recv}]$$

By rule [recv] of Eq. (3.3.3):

$$\frac{(\nu, S) \xrightarrow{?m} (\nu', S')}{(\nu, S, m;\,\texttt{M}) \xrightarrow{\tau} (\nu', S', \texttt{M})}\ [\texttt{recv}]$$

By the premise of the above, thanks to Case (1) of Definition 5.4.1, it follows that $(\nu', S', \texttt{M})$ is balanced, as required. The furthermore case holds trivially.

Case 3. If rule [Δ-L] then $\Delta = \Delta_1,\ \Delta_2$ and $\Delta' = \Delta_1',\ \Delta_2$ and:

$$\frac{\Delta_1 \longrightarrow \Delta_1'}{\Delta_1,\ \Delta_2 \longrightarrow \Delta_1',\ \Delta_2}\ [\Delta\text{-L}]$$

It is easy to see that Δ_1 and Δ_2 are both balanced. By the induction hypothesis also Δ_1' is balanced. Notice that the composition of balanced environments is not necessarily balanced. So we need to show that all the items in Definition 5.4.1 hold for Δ_1'. We show only for Case (1) of Definition 5.4.1, as the other cases are similar. So, suppose: $\Delta' = \Delta'', p : (\nu, S), qp : m;\mathtt{M}$, then we have to show: $(\nu, S) \xrightarrow{?m} (\nu', S')$ and $\Delta', p : (\nu', S'), qp : \mathtt{M} \in \mathrm{Bal}$. The interesting cases are:

- $\Delta_1'(p) = (\nu, S)$, $\Delta_2(qp) = m;\mathtt{M}$ and $\Delta_1(p) \neq \Delta_1'(p)$. We argue that this case is impossible. So suppose, by contradiction, it is not the case. Then, since queue qp did not change during the transition, it must be $\Delta_1(p) = (\nu_0, S_0)$ for some ν_0, S_0 such that: $(\nu_0, S_0) \xrightarrow{!m'} (\nu, S)$ and $\Delta_1(pq) = \mathtt{M}'$ and $\Delta_1'(pq) = \mathtt{M}'; m'$. By Lemma A.5.9, $(\nu_0, S_0) \xrightarrow{?m} \not\rightarrow$ and hence Δ was not balanced in the first place: contradiction.

- $\Delta_2(p) = (\nu, S)$, $\Delta_1'(qp) = m;\mathtt{M}$ and $\Delta_1(qp) \neq \Delta_1'(qp)$. Then $\mathtt{M} = \mathtt{M}'; m'$ and $\Delta_1(q) = (\nu_0, S_0)$, $\Delta_1'(q) = (\nu_0', S_0')$ and $\Delta_1(qp) = m;\mathtt{M}'$ with: $(\nu_0, S_0) \xrightarrow{!m'} (\nu_0', S_0')$. By Case (2) of Definition 5.4.1, for some $\mathtt{M}_0$: $(\nu, S, \mathtt{M}) \perp (\nu_0, S_0, \mathtt{M}_0)$. by rule [com-r] of Figure C.2 Eq. (3.3.4):

  $$(\nu, S, m;\mathtt{M}') \mid (\nu_0, S_0, \mathtt{M}_0) \xrightarrow{\tau} (\nu, S, m;\mathtt{M}) \mid (\nu_0', S_0', \mathtt{M}_0)$$

  By Lemma A.6.3:

  $$(\nu, S, m;\mathtt{M}) \perp (\nu_0, S_0, \mathtt{M}_0)$$

  Hence $(\nu, S) \xrightarrow{?m} (\nu', S')$ for some (ν', S') such that $(\nu', S', \mathtt{M}) \perp (\nu_0', S_0', \mathtt{M}_0)$. It remains to show that $\Delta', p : (\nu', S'), qp : \mathtt{M}$ is balanced by Definition 5.4.1. For Cases (2) and (3) of Definition 5.4.1 it surely holds. Regarding Case (1) of Definition 5.4.1, it follows by a simple induction on the length of $\mathtt{M}$. □

B.4 Well-typed Processes (Inversion Lemma)

Lemma B.4.1 (Inversion Lemma). *Let* Δ *be* well-formed. *Claims B.4.2 to B.4.16 each hold.*

Claim B.4.2. If $\Gamma, \theta \vdash P \mid Q \blacktriangleright \Delta$, then $\Delta = (\Delta_1, \Delta_2)$ and $\theta = (\theta_1, \theta_2)$ and $\Gamma, \theta_1 \vdash P \blacktriangleright \Delta_1$ and $\Gamma, \theta_2 \vdash Q \blacktriangleright \Delta_2$.

Claim B.4.3. If $\Gamma, \theta \vdash (\nu pq)\, P \blacktriangleright \Delta$, then $\Gamma, \theta \vdash P \blacktriangleright \Delta, p : (\nu_1, S_1), qp : \mathtt{M}_2, q : (\nu_2, S_2), pq : \mathtt{M}_2$ and $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ and both (ν_1, S_1) and (ν_2, S_2) are *well-formed*.

Claim B.4.4. If $\Gamma, \theta \vdash \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q \blacktriangleright \Delta$, then $\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta \vdash Q \blacktriangleright \Delta$ and:

$$\forall (\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta} : \quad \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta' \vdash P \blacktriangleright \vec{r} : (\vec{v},\ \vec{S})$$

Claim B.4.5. If $\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta \vdash \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ P' \mid Q \blacktriangleright \Delta$, then $\Delta = (\Delta_1, \Delta_2)$, $\theta = (\theta_1, \theta_2)$ and:

- $\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta_1 \vdash \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ P' \blacktriangleright \Delta_1$.
- $\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta_2 \vdash \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q \blacktriangleright \Delta_2$.

Claim B.4.6. If $\Gamma, \theta \vdash X\langle \vec{v};\vec{r}\rangle \blacktriangleright \Delta$, then $\Gamma = \Gamma',\ X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta})$ and:

$$\forall i : \Gamma' \vdash \vec{v}_i : \vec{T}_i \quad \text{and} \quad \theta \in \boldsymbol{\theta} \quad \text{and} \quad \Delta = \vec{r} : (\vec{v}, \vec{S}) \in \boldsymbol{\Delta}$$

Claim B.4.7. If $\Gamma, \theta \vdash p \lhd l\,(w).P \blacktriangleright \Delta$, then $\Delta = \Delta', p : (\nu, \{\mathtt{C}_i\}_{i\in I})$ and $\mathtt{NEQ}(\Delta) = \varnothing$ and $\exists j \in I$ such that $\nu \models \delta$ and $\square_j = !$:

- T_j base-type $\implies \Gamma; \theta \vdash P \blacktriangleright \Delta', p : (\nu\left[\lambda_j \mapsto 0\right], S_j)$ and $\Gamma \vdash w : T_j$.
- $T_j = (\delta', S') \implies \Delta' = \Delta'', w : (\nu', S')$ and $\nu' \models \delta'$ and $\Gamma; \theta \vdash P \blacktriangleright \Delta'', p : (\nu\left[\lambda_j \mapsto 0\right], S_j)$.

Claim B.4.8. If $\Gamma, \theta \vdash p^e \rhd l(v) : P \blacktriangleright \Delta$, then $\Delta = \Delta', p : (\nu, ?l\langle T\rangle\,(\delta, \{\lambda\})\,.S)$ and Δ' not e-reading and:

- $\forall t$ such that $\nu + t \models \delta \iff t \in e$
- $\forall t \in e$:

- $T = (\delta',\ S') \implies \Gamma;\theta + t \vdash P \blacktriangleright \Delta' + t, p : (\nu + t\,[\lambda \mapsto 0]\,, S), v : (\nu', S')$ and $\nu' \models \delta'$.
- T base-type $\implies \Gamma, v : T;\theta + t \vdash P \blacktriangleright \Delta' + t, p : (\nu + t\,[\lambda \mapsto 0]\,, S)$.

Claim B.4.9. If $\Gamma, \theta \vdash p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \blacktriangleright \Delta$, then $\Delta = \Delta', p : (\nu, \left\{\square_j\, l_j \left\langle T_j \right\rangle (\delta_j, \lambda_j).S_j\right\}_{j\in J})$ and $\neg(|J| = |I| = 1)$ and, $\forall j \in J$ such that $\nu \models \delta_j$ implies:

$$\square_j = ? \quad \text{and} \quad \left(\exists i \in I : \Gamma, \theta \vdash p^e \rhd l_i(v_i) : P_i \blacktriangleright \Delta', p : (\nu, \square_j\, l_j \left\langle T_j \right\rangle (\delta_j, \lambda_j).S_j) \text{ and } l_i = l_j\right)$$

Claim B.4.10. If $\Gamma, \theta \vdash p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \texttt{after}\ Q \blacktriangleright \Delta$, then $\Delta = \Delta', p : (\nu, \left\{\texttt{C}_j\right\}_{j\in J})$ and:

- $\Gamma, \theta \vdash p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \blacktriangleright \Delta',\ p : (\nu, \left\{\texttt{C}_j\right\}_{j\in J})$.
- $\Gamma, \theta + n \vdash Q \blacktriangleright \Delta' + n,\ p : (\nu + n, \left\{\texttt{C}_j\right\}_{j\in J})$.

Claim B.4.11. If $\Gamma, \theta \vdash \texttt{set}\,\textcircled{x}.P \blacktriangleright \Delta$, then $\theta = \theta', x : t$ and $\Gamma, \theta', x : 0 \vdash P \blacktriangleright \Delta$.

Claim B.4.12. If $\Gamma, \theta \vdash \texttt{delay}\,(\delta)\,.P \blacktriangleright \Delta$, then $\forall t \in \delta : \Gamma, \theta \vdash \texttt{delay}\,(t)\,.P \blacktriangleright \Delta$.

Claim B.4.13. If $\Gamma, \theta \vdash \texttt{delay}\,(t)\,.P \blacktriangleright \Delta$, then Δ not t-reading and $\Gamma, \theta + t \vdash P \blacktriangleright \Delta + t$.

Claim B.4.14. If $\Gamma, \theta \vdash \texttt{if}\left(\delta\right)\texttt{then}\ P\ \texttt{else}\ Q \blacktriangleright \Delta$, then:

- $\theta \models \delta \implies \Gamma, \theta \vdash P \blacktriangleright \Delta$,
- $\theta \not\models \delta \implies \Gamma, \theta \vdash Q \blacktriangleright \Delta$.

Claim B.4.15. If $\Gamma, \theta \vdash qp : \varnothing \blacktriangleright \Delta$, then $\Delta = \Delta', qp : \varnothing$.

Claim B.4.16. If $\Gamma, \theta \vdash qp : l\texttt{v} \cdot h \blacktriangleright \Delta$, then $\Delta = \Delta', qp : l\left\langle T \right\rangle;\texttt{M}$ and $\texttt{NEQ}\left(\Delta'\right) = \varnothing$ and:

- $T = (\delta',\ S') \implies \Delta' = \Delta'', \texttt{v} : (\nu', S')$ and $\Gamma, \theta \vdash qp : h \blacktriangleright \Delta'', qp : \texttt{M}$ and $\nu' \models \delta'$.
- T base-type $\implies \Gamma \vdash \texttt{v} : T$ and $\Gamma, \theta \vdash qp : h \blacktriangleright \Delta', qp : \texttt{M}$.

Proof. As standard, it follows by induction on each derivation, using the *type-checking* rules given in Eqs. (5.2.1 to 5.2.6). □

B.4.1 Well-typed Enabled Actions

Lemma B.4.17. *Let* $\Delta, p : (\nu, S)$ *be* well-formed. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta, p : (\nu, S)$ *and* $(\nu, S) \xrightarrow{?m}$ *then* $p \in \texttt{Wait}(P)$.

Proof. By induction on the derivation of the transition $(\nu, S) \xrightarrow{?m}$, analysing the last rule applied of those given in Figure C.2. The only applicable rules are [act] and [unfold]. As in Lemma A.5.9, we only show rule [act]:

$$\frac{\nu \models \delta_j \quad m = l_j\langle T_j\rangle \quad j \in I}{\left(\nu, \left\{\Box_i\, l_i\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I}\right) \xrightarrow{\Box_j m} \left(\nu\left[\lambda_j \mapsto 0\right], S_j\right)}\ [\texttt{act}]$$

where $S = \left\{\Box_i\, l_i\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I}$. By inspecting Figure 4.1 for when $p \in \texttt{Wait}(P)$, it remains for us to show that P is structured as follows:

$$P \in \left\{\ p^e \rhd \left\{l_j(v_j) : P_j\right\}_{j\in J},\ p^{\diamond n} \rhd \left\{l_j(v_j) : P_j\right\}_{j\in J} \texttt{after}\ Q\ \right\}$$

We proceed by induction on the derivation of $\Gamma, \theta \vdash P \blacktriangleright \Delta, p : (\nu, S)$, analysing the last rule applied of those given in Eqs. (5.2.1 to 5.2.6):

Case 1. If rule [Branch], then $P = p^e \rhd \left\{l_j(v_j) : P_j\right\}_{j\in J}$ and we obtain our thesis.

Case 2. If rule [Timeout], then $P = p^{\diamond n} \rhd \left\{l_j(v_j) : P_j\right\}_{j\in J}$ after Q.

Case 3. If rule [VRecv], then $P = p^e \rhd l(v) : P'$. (The same holds for rule [DRecv].)

Case 4. If rule [VSend], then $P = p \lhd l\,(w).P'$. Clearly $p \notin \texttt{Wait}(P)$, and it remains for us to prove that this case is not applicable. By rule [VSend]:

$$\frac{\begin{array}{c}\exists j \in I : \texttt{C}_j = !l\langle T\rangle\,(\delta, \{\lambda\})\,.S \ \wedge\ \nu \models \delta \ \wedge\ \Gamma \vdash w : T \ \wedge \\ \Gamma, \theta \vdash P \blacktriangleright \Delta,\ p : (\nu\left[\lambda \mapsto 0\right], S)\end{array}}{\Gamma, \theta \vdash p \lhd l\,(w).P' \blacktriangleright \Delta, p : (\nu, \{\texttt{C}_i\}_{i\in I})}\ [\texttt{VSend}]$$

By the premise of rule [VSend], $\exists j \in I$ such that $\texttt{C}_j = !l\langle T\rangle\,(\delta, \{\lambda\})\,.S$ and $\nu \models \delta$ and $\Box$ $=!$. By Lemma A.5.9, since (ν, S) is *well-formed*, it cannot be that both $(\nu, S) \xrightarrow{?m}$ and $(\nu, S) \xrightarrow{!m'}$, where $m' = l\langle T\rangle$. Therefore, this case is not applicable

since it cannot be that a *sending* process P is *well-typed* against p, which has an enabled *receiving* action. (The same holds for rule [DSend].) □

B.4.2 Type-checking Processes

Lemma B.4.18. *Let* Δ *be* well-formed. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta$ *and* $P \equiv Q$, *then* $\Gamma, \theta \vdash Q \blacktriangleright \Delta$.

Proof. As standard Bocchi et al. 2019; Honda et al. 2008; Yoshida et al. 2007, with the additions in Section 4.2.1. □

B.4.3 Well-typed Substitutions

Lemma B.4.19 (Substitution)**.** *Let* Δ *be* well-formed. *Claims B.4.20 and B.4.21 both hold.*

Claim B.4.20. If $\Gamma, \theta \vdash P \blacktriangleright \Delta, p : (\nu, S)$ and $b \notin \mathrm{Dom}(\Delta)$, then $\Gamma, \theta \vdash P\,[^{b}/_{p}] \blacktriangleright \Delta, b : (\nu, S)$.

Claim B.4.21. If $\Gamma_1, m : T,\ \theta \vdash P \blacktriangleright \Delta$ and $\Gamma_2 \vdash v : T$ and $\mathrm{Dom}(\Gamma_1) \cap \mathrm{Dom}(\Gamma_2) = \varnothing$, then $\Gamma_1, \Gamma_2, \theta \vdash P\,[^{v}/_{m}] \blacktriangleright \Delta$.

Proof. Claims B.4.20 and B.4.21 both hold as standard (Bettini et al. 2008; Honda et al. 2008, 2016; Scalas et al. 2019; Vasconcelos 2012; Yoshida et al. 2007). □

B.4.4 Type-checking Messages

Lemma B.4.22 (Well-typed Messages)**.** *Let* $\Gamma, \theta \vdash qp : h \blacktriangleright \Delta, qp : \mathtt{M}$ *and* $\Delta, qp : \mathtt{M}$ *be* well-formed. *Claims B.4.23 and B.4.24 both hold, for all* l.

Claim B.4.23. If $\Gamma \vdash \mathtt{v} : T$, then $\Gamma, \theta \vdash qp : h \cdot l\mathtt{v} \blacktriangleright \Delta, qp : \mathtt{M}; l\,\langle T \rangle$.

Claim B.4.24. If $T = (\delta,\ S)$ and $b \neq \mathrm{Dom}(\Delta)$ and $\exists \nu$ such that $\nu \models \delta$, then:

$$\Gamma, \theta \vdash qp : h \cdot lb \blacktriangleright \Delta, qp : \mathtt{M}; l\,\langle T \rangle, b : (\nu, S)$$

Proof. We proceed by addressing each claim in turn:

Claim B.4.23 Holds following the conclusion of rule [VQue] in Figure C.5 Eq. (5.2.6), where we use the given assumptions as premise:

$$\frac{T \text{ base-type} \quad \Gamma \vdash \mathtt{v} : T \quad \Gamma, \theta \vdash qp : h \blacktriangleright \Delta,\ qp : \mathtt{M}}{\Gamma, \theta \vdash qp : l\mathtt{v} \cdot h \blacktriangleright \Delta,\ qp : l\,\langle T \rangle; \mathtt{M}}\ [\mathtt{VQue}]$$

Claim B.4.24 Holds following the conclusion of rule [DQue] in Figure C.5 Eq. (5.2.6), similar to the case of Claim B.4.23 (above). □

B.4.5 Type-checking Message Handling

Lemma B.4.25. *Let* $\Delta, qp : l\,\langle T\rangle\,;\mathtt{M}$ *be* well-formed. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta, qp : l\,\langle T\rangle\,;\mathtt{M}$, *then* $p \in \mathtt{NEQ}(P)$.

Proof. By induction on the typing derivation, the key rules are [VQue] are [DQue], which are structured such that: $\Gamma, \theta \vdash qp : l\mathtt{v} \cdot h \blacktriangleright \Delta,\ qp : m;\mathtt{M}$ where $m = l\,\langle T\rangle$. The thesis follows Figure 4.1, as $p \in \mathtt{NEQ}(qp : l\mathtt{v} \cdot h)$ holds. □

Lemma B.4.26. *Let* Δ *be* well-formed. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta$ *and* $p \in \mathtt{NEQ}(P)$, *then* $\exists \Delta'$ *such that* $\Delta = \Delta',\ qp : l\,\langle T\rangle\,;\mathtt{M}$.

Proof. By the definition of $\mathtt{NEQ}(P)$ in Figure 4.1, it must be that $P \equiv qp : l\mathtt{v} \cdot h \mid Q$, and by Claim B.4.16 of Lemma B.4.1, we obtain our thesis. □

Lemma B.4.27. *Let* Δ *be* well-formed. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta$ *and* $\mathtt{NEQ}(P) = \varnothing$, *then* $\mathtt{NEQ}(\Delta) = \varnothing$.

Proof. Since $\mathtt{NEQ}(P) = \varnothing$, it follows that $\forall qp : h$ such that $P \equiv qp : h \mid Q$, then $h = \varnothing$. By Claim B.4.2 in Lemma B.4.1, $\Delta = \Delta_1, \Delta_2$ and, $\Gamma, \theta \vdash qp : h \blacktriangleright \Delta_1$ and $\Gamma, \theta \vdash Q \blacktriangleright \Delta_2$. By Claim B.4.15 in Lemma B.4.1, it follows that $\Delta_1 = \Delta_1', qp : \mathtt{M}$ and $\mathtt{M} = \varnothing$. We obtain our thesis: given that all message queues in P are empty, if P is *well-typed* against Δ then, there exists corresponding queues in Δ that are all also empty. □

B.4.6 Delayable Session Environments

Lemma B.4.28. *Let* Δ *be* well-formed. *If* Δ *is balanced and delayable and* $\Delta + t$ *is* well-formed, *then* $\Delta + t$ *is balanced.*

Proof. We show that $\Delta + t$ satisfies all conditions of Definition 5.4.1:

Case 1. Since Δ is delayable, by Definition B.1.6, all queues in the domain of Δ are *empty*. Therefore, the case where $\Delta = \Delta', p : (\nu, S), qp : m;\mathtt{M}$, conflicts with the hypothesis, and is not applicable.

Case 2. If $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2), pq : \mathtt{M}_2$, then $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$. Since (ν_1, S_1) and (ν_2, S_2) are *well-formed*, then by Lemmas A.6.1 and A.6.3, $(\nu_1 + t, S_1)$ and $(\nu_2 + t, S_2)$ are *well-formed*, and $(\nu_1 + t, S_1, \mathtt{M}) \perp (\nu_2 + t, S_2, \mathtt{M}_2)$. Therefore, $\Delta + t$ is balanced.

Case 3. If $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2)$, then $\exists \mathtt{M}_2 : (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ and the thesis follows Case 2. □

Lemma B.4.29. *If Δ is balanced,* well-formed *and not t-reading, then Δ is delayable.*

Proof. If we suppose by contradiction that Δ is *not* delayable, then by Definition B.1.6, $\exists qp, m, \mathtt{M}$ such that $qp \in \mathrm{Dom}(\Delta)$ and $\Delta(qp) = m;\mathtt{M}$ and $\exists p \in \mathrm{Dom}(\Delta)$. Since Δ is balanced, for some $\Delta(p) = (\nu', S')$ it follows Case (1) of Definition 5.4.1:

$$\Delta = \Delta', p : (\nu', S'), qp : m;\mathtt{M} \implies (\nu', S') \xrightarrow{?m} (\nu'', S'') \quad \text{and} \quad \Delta', p : (\nu'', S''), qp : \mathtt{M} \in \mathrm{Bal}$$

However, this contradicts with Δ not t-reading: by Definition 5.1.1, there are no roles $q \in \mathrm{Dom}(\Delta)$ such that $\Delta(q) = (\nu''', S''')$ and $(\nu''' + t', S''') \xrightarrow{?m'}$, for all $t' < t$. Therefore, Δ must be delayable. □

Lemma B.4.30. *Let Δ be* well-formed. *If $\Gamma, \theta \vdash P \blacktriangleright \Delta$ and $\Phi_t(P)$ is* defined *and Δ is balanced, then Δ is delayable.*

Proof. Since $\Phi_t(P)$ is *defined*, we assume that $t > 0$. We proceed by induction on the typing derivation of P:

Case 1. If $P = p^e \rhd \left\{ l_i(v_i) \ : \ P_i \right\}_{i \in I} \mid qp \ : \ h$ then by Claim B.4.2 of Lemma B.4.1, $\theta = (\theta_1, \theta_2)$, $\Delta = (\Delta_1, \Delta_2)$ and:

$$\Gamma, \theta_1 \vdash p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \blacktriangleright \Delta_1 \qquad \Gamma, \theta_2 \vdash qp : h \blacktriangleright \Delta_2$$

By Claim B.4.9 of Lemma B.4.1, $\Delta_1 = \Delta_1', p : (\nu_1, S_1)$ where $S_1 = \left\{ \Box_j \, l_j \left\langle T_j \right\rangle \left(\delta_j, \lambda_j \right).S_j \right\}_{j \in J}$ and $\neg(|J| = |I| = 1)$. It also follows that $\forall j \in J$ such that $\nu_1 \models \delta_j$, then $\Box_j = ?$ and:

$$\exists i \in I : l_i = l_j \quad \text{and} \quad \Gamma, \theta_1 \vdash p^e \rhd l_i(v_i) : P_i \blacktriangleright \Delta_1', p : (\nu_1, \Box_j \, l_j \left\langle T_j \right\rangle \left(\delta_j, \lambda_j \right).S_j)$$

By Claim B.4.8 of Lemma B.4.1, Δ_1' is not e-reading, and $\forall t : \nu_1 + t \models \delta_j \iff t \in e$. Since $\nu_1 \models \delta_j$ and $\Box_j = ?$, by Claim A.5.3 of Lemma A.5.1, by rule [act] in Figure C.2 Eq. (3.3.2):

$$(\nu_1, \Box_j\, l_j\, \langle T_j \rangle\, (\delta_j, \lambda_j).S_j) \xrightarrow{?l_j\langle T_j \rangle} (\nu_1\, [\lambda_j \mapsto 0]\,, S_j)$$

By Lemma B.4.17, $p \in \mathtt{Wait}(P)$. By Definition 4.2.2, $\mathtt{Wait}(P) \cap \mathtt{NEQ}(P) = \varnothing$. Therefore, $p \notin \mathtt{NEQ}(P)$ and, since $p \notin \mathtt{NEQ}(P)$, then $qp : h = \varnothing$. By Claim B.4.15 of Lemma B.4.1, $\Delta_2 = \Delta_2', qp : \mathtt{M}_1$ and since $h = \varnothing$ then by Lemma B.4.27, $\mathtt{M}_1 = \varnothing$. It follows that Δ_2 is delayable, since there cannot be any other queues present in Δ_2' given that $\Gamma, \theta_2 \vdash qp : h \blacktriangleright \Delta_2', qp : \mathtt{M}_1$. It remains to show that Δ_1 is delayable in order to obtain our thesis and prove that Δ is delayable. Since Δ is balanced, it follows Δ_1' is also balanced, Since Δ_1' is balanced, *well-formed* and not e-reading, the thesis follows Lemma B.4.29.

Case 2. If $P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}$ `after` $Q \mid qp : h$ then by Claim B.4.2 of Lemma B.4.1, $\theta = (\theta_1, \theta_2)$, $\Delta = (\Delta_1, \Delta_2)$ and:

$$\Gamma, \theta_1 \vdash p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{after}\ Q \blacktriangleright \Delta_1 \qquad \Gamma, \theta_2 \vdash qp : h \blacktriangleright \Delta_2$$

By Claim B.4.10 of Lemma B.4.1, $\Delta_1 = \Delta_1', p : (\nu_1, S_1)$ where $S_1 = \left\{ \mathtt{C}_j \right\}_{j \in J}$ and:

$$\Gamma, \theta_1 \vdash p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \blacktriangleright \Delta_1', p : (\nu_1, \left\{ \mathtt{C}_j \right\}_{j \in J}) \tag{B.4.1}$$

$$\Gamma, \theta_1 + n \vdash Q \blacktriangleright \Delta_1 + n', p : (\nu_1 + n, \left\{ \mathtt{C}_j \right\}_{j \in J}) \tag{B.4.2}$$

By Definition 4.2.2, if $t \diamond n$ then the thesis follows by the induction hypothesis on Eq. (B.4.1). (See Case 1.) Otherwise, $\neg(t \diamond n)$ and $\Phi_t(p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{after}\ Q) = \Phi_{t-n}(Q)$, and the thesis follows by induction on Eq. (B.4.2).

Case 3. If $P = (\nu pq)\ P'$ then by Claim B.4.3 of Lemma B.4.1:

$$\Gamma, \theta \vdash P' \blacktriangleright \Delta,\ p : (\nu_1, S_1),\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2 \tag{B.4.3}$$

where $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ and both (ν_1, S_1) and (ν_2, S_2) are *well-formed*. By Definition 5.4.2, $\Delta, p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2), pq : \mathtt{M}_2$ is fully-balanced, and therefore the thesis follows by the induction hypothesis of Eq. (B.4.3).

Case 4. If $P = \mathtt{delay}\,(t')\,.P'$ then by Claim B.4.13 of Lemma B.4.1, Δ is not t'-reading and $\Gamma, \theta + t' \vdash P'\Delta + t'$. By Definition 4.2.2:

i. If $t \leq t'$, then $\Phi_t\big(\mathtt{delay}\,(t')\,.P'\big) = \mathtt{delay}\,(t' - t)\,.P'$. Since Δ is not t'-reading, it follows Δ is not t-reading also. The thesis follows Lemma B.4.29.

ii. If $t > t'$, then $\Phi_t\big(\mathtt{delay}\,(t')\,.P'\big) = \Phi_{t-t'}\big(P'\big)$ and the thesis follows by the induction hypothesis on: $\Gamma, \theta + t' \vdash \Phi_{t-t'}\big(P'\big) \blacktriangleright \Delta + t'$.

We only show the key cases of P. The other cases of P follow Definition 4.2.2. □

Lemma B.4.31. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta$ *then* Δ *is* fe.

Proof. A simple induction on the typing derivation. □

Lemma B.4.32. *If* Δ *is* well-formed *and* $\Delta + t$ *is live, then* $\Delta + t$ *is* well-formed.

Proof. Suppose $\Delta + t(p) = (\nu,\ S)$ and $S \neq \mathtt{end}$. It must be $\nu = \nu' + t$ with $\Delta(p) = (\nu',\ S)$ and hence $(\nu',\ S)$ is *well-formed*. By Lemma A.4.2 we have that $(\nu' + t,\ S)$ is *well-formed*, and by Lemma A.4.1 it is live, as required. □

B.5 Reduction Steps

B.5.1 Time-passing Steps

Lemma B.5.1. *Let* Δ *be* well-formed. *If* $\Gamma, \theta \vdash P \blacktriangleright \Delta, \Phi_t\big(P\big)$ *is defined and* $\mathtt{NEQ}\big(P\big) = \varnothing$, *then* $\Delta + t$ *is* well-formed *and* $\Gamma, \theta + t \vdash \Phi_t\big(P\big) \blacktriangleright \Delta + t$.

Proof. We proceed by induction on the derivation of $\Gamma, \theta \vdash P \blacktriangleright \Delta$ analysing the last rule applied of the *type-checking* rules given in Eqs. (5.2.1 to 5.2.6):

Case 1. If rule [VRecv], then $P = p^e \rhd l(v) : P'$ and following Figure C.5 Eq. (5.2.5):

$$\frac{\begin{array}{c} T \text{ base-type} \quad \Delta' \text{ not } e\text{-reading} \quad \forall t' : \nu + t' \models \delta \Longleftrightarrow t' \in e \\ \forall t' \in e : \Gamma, v : T,\ \theta + t' \vdash P \blacktriangleright \Delta' + t',\ p : (\nu + t'\,[\lambda \mapsto 0]\,, S) \end{array}}{\Gamma, \theta \vdash p^e \rhd l(v) : P \blacktriangleright \Delta',\ p : (\nu, ?l\langle T\rangle\,(\delta, \{\lambda\})\,.S)}\ [\mathtt{VRecv}]$$

where by Claim B.4.8 of Lemma B.4.1, $\Delta = \Delta', p : (\nu, ?l\langle T\rangle\,(\delta, \{\lambda\})\,.S)$. Since $\Phi_t(P)$ is defined, by Definition 4.2.2, it must be that $t \in e$. (If $e = \infty$ then this always holds.) Coinciding with the first premise of rule [VRecv], it follows that $\nu + t \models \delta$; additionally, by the second premise it follows that the time-step preserves 'well-typedness':

$$\forall t' \in e : \Gamma, v : T,\ \theta + t' \vdash P' \blacktriangleright \Delta' + t',\ p : (\nu + t'\,[\lambda \mapsto 0]\,, S)$$

It remains to show that $\Delta + t$ is *well-formed*. By Lemma B.4.31 $\Delta + t$ is live. By Lemma B.4.32 it is *well-formed*.

Case 2. If rule [DRecv], then we obtain our thesis analogously to Case 1.

Case 3. If rule [Branch], then $P = p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I}$ and following Figure C.5 Eq. (5.2.4):

$$\frac{\neg(|J| = |I| = 1) \quad \forall j \in J : \nu \models \delta_j \implies \square_j = ? \ \wedge\ \exists i \in I : l_i = l_j \ \wedge \quad \Gamma, \theta \vdash p^e \rhd l_i(v_i) : P_i \blacktriangleright \Delta',\ p : (\nu, \square_j\, l_j\, \langle T_j\rangle\, (\delta_j, \lambda_j).S_j)}{\Gamma, \theta \vdash p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \blacktriangleright \Delta',\ p : (\nu, \left\{\square_j\, l_j\, \langle T_j\rangle\, (\delta_j, \lambda_j).S_j\right\}_{j\in J})}\ [\texttt{Branch}]$$

where by Claim B.4.9 of Lemma B.4.1, $\Delta = \Delta', p : (\nu, \left\{\square_j\, l_j\, \langle T_j\rangle\, (\delta_j, \lambda_j).S_j\right\}_{j\in J})$. Since $\Phi_t(P)$ is defined, by Definition 4.2.2, it must be that $t \in e$. (If $e = \infty$ then this always holds.) By the premise of rule [Branch], each enabled action j in S is receiving, and has a corresponding branch i in P where $l_i = l_j$ and:

$$\Gamma, \theta \vdash p^e \rhd l_i(v_i) : P_i \blacktriangleright \Delta', p : (\nu, \square_j\, l_j\, \langle T_j\rangle\, (\delta_j, \lambda_j).S_j)$$

Therefore, the hypothesis holds by induction. (See Cases 1 and 2.)

Case 4. If rule [Timeout], then $p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I}$ `after` Q and following Figure C.5 Eq. (5.2.4):

$$\frac{\Gamma, \theta \vdash p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \blacktriangleright \Delta',\ p : (\nu, \left\{\mathtt{C}_j\right\}_{j\in J}) \qquad \Gamma, \theta + n \vdash Q \blacktriangleright \Delta' + n,\ p : (\nu + n, \left\{\mathtt{C}_j\right\}_{j\in J})}{\Gamma, \theta \vdash p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \texttt{after}\ Q \blacktriangleright \Delta',\ p : (\nu, \left\{\mathtt{C}_j\right\}_{j\in J})}\ [\texttt{Timeout}]$$

where by Claim B.4.10 of Lemma B.4.1, $\Delta = \Delta', p : (\nu, \{\mathtt{C}_j\}_{j\in J})$. Since $\Phi_t(P)$ is defined by Definition 4.2.2, we show for each case of t:

i. If $t \diamond n$, then $\Phi_t(P) = p^{\diamond n-t} \rhd \{l_i(v_i) : P_i\}_{i\in I}$ `after` Q. Therefore, by the induction hypothesis:

$$\frac{\begin{array}{c}\Gamma, \theta + t \vdash p^{\diamond n+t} \rhd \{l_i(v_i) : P_i\}_{i\in I} \blacktriangleright \Delta' + t,\ p : (\nu + t, \{\mathtt{C}_j\}_{j\in J}) \\ \Gamma, \theta + n \vdash Q \blacktriangleright \Delta' + t + n - t,\ p : (\nu + t + n - t, \{\mathtt{C}_j\}_{j\in J})\end{array}}{\Gamma, \theta + t \vdash p^{\diamond n-t} \rhd \{l_i(v_i) : P_i\}_{i\in I} \ \mathtt{after}\ Q \blacktriangleright \Delta' + t,\ p : (\nu + t, \{\mathtt{C}_j\}_{j\in J})}\ [\mathtt{Timeout}]$$

The first premise of rule [`Timeout`] holds by induction on the derivation. (See Case 3.) The second premise of rule [`Timeout`] coincides with the hypothesis, as $t + n - t = n$.

ii. If $\neg(t \diamond n)$, then $\Phi_t(P) = \Phi_{t-n}(Q)$. The thesis follows the induction hypothesis:

$$\Gamma, \theta + t \vdash \Phi_{t-n}(Q) \blacktriangleright \Delta + t$$

Case 5. If rule [`Del-t`], then $P = \mathtt{delay}\,(t').P'$ and:

i. If $t \leq t'$, then $\Phi_t(P) = \mathtt{delay}\,(t' - t).P'$ and following Figure C.4 Eq. (5.2.2):

$$\frac{\Gamma, \theta + t' \vdash P' \blacktriangleright \Delta + t' \quad \Delta \text{ not } t'\text{-reading}}{\Gamma, \theta \vdash \mathtt{delay}\,(t').P' \blacktriangleright \Delta}\ [\mathtt{Del}\text{-}t]$$

Since Δ is not t'-reading, by Definition 5.1.1, it follows that Δ is also not t-reading. Therefore, we obtain our thesis: $\Gamma, \theta + t \vdash \mathtt{delay}\,(t' - t).P' \blacktriangleright \Delta + t$.

ii. If $t > t'$, then $\Phi_t(P) = \Phi_{t-t'}(P')$. The thesis follows the induction hypothesis:

$$\Gamma, \theta + t \vdash \Phi_{t-t'}(P') \blacktriangleright \Delta + t$$

Case 6. If rule [`Par`], then $P = P' \mid Q$ and following Figure C.4 Eq. (5.2.1):

$$\frac{\Gamma, \theta_1 \vdash P' \blacktriangleright \Delta_1 \quad \Gamma, \theta_2 \vdash Q \blacktriangleright \Delta_2}{\Gamma, \theta \vdash P' \mid Q \blacktriangleright (\Delta_1, \Delta_2)}\ [\mathtt{Par}]$$

where $\theta = (\theta_1, \theta_2)$ and $\Delta = (\Delta_1, \Delta_2)$. Since Δ is *well-formed*, it follows both Δ_1 and Δ_2 are also *well-formed*. By the induction hypothesis:

$$\frac{\Gamma, \theta_1 + t \vdash \Phi_t(P') \blacktriangleright \Delta_1 + t \quad \Gamma, \theta_2 + t \vdash \Phi_t(Q) \blacktriangleright \Delta_2 + t}{\Gamma, (\theta_1 + t, \theta_2 + t) \vdash \Phi_t(P' \mid Q) \blacktriangleright (\Delta_1, \Delta_2) + t} \ [\texttt{Par}]$$

where $\Delta + t = (\Delta_1, \Delta_2) + t = (\Delta_1 + t, \Delta_2 + t)$. (See other cases for P' and Q.)

Case 7. If rule [Res], then $P = (\nu pq)\ P'$ and it follows Figure C.4 Eq. (5.2.1):

$$\frac{\begin{array}{c}\Gamma, \theta \vdash P' \blacktriangleright \Delta,\ p : (\nu_1, S_1),\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2 \\ (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2) \qquad \forall i \in \{1, 2\}.\, S_i \text{ well-formed against } \nu_i\end{array}}{\Gamma, \theta \vdash (\nu pq)\ P' \blacktriangleright \Delta} \ [\texttt{Res}]$$

By the second premise of rule [Res], $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ and both (ν_1, S_1) and (ν_2, S_2) are *well-formed*. Since $\mathtt{NEQ}(P) = \varnothing$, by Lemma B.4.27, it follows that $\forall qp$ such that $qp \in \mathrm{Dom}(\Delta) \implies \Delta(qp) = \varnothing$, and $\mathtt{M}_1 = \varnothing = \mathtt{M}_2$. By Condition (4) of Definition 3.4.6, since $\mathtt{M}_1 = \varnothing = \mathtt{M}_2$, then $S_1 = \overline{S_2}$ and $\nu_1 = \nu_2$. Hereafter, we only show for p, as the proof is similar for q, and . By the premise of rule [Res], p is *well-formed*. By Lemma A.4.1, p is live. By Definition A.1.2, either $S_1 = \mathtt{end}$, or p is *fe*.

i. If $S_1 = \mathtt{end}$, then by Lemma A.3.1 we obtain our thesis as `end` is always *well-formed*.

ii. Otherwise, p is *fe*. By Definition 3.3.1, $\exists t'$ such that $(\nu_1, S_1) \xrightarrow{t'\ \square m}$. It follows that $\forall t'' \leq t'$ $(\nu + t'', S)$ is also *fe*, since $(\nu + t'', S) \xrightarrow{t' - t''\ \square m}$. Therefore, $(\nu_1 + t', S_1)$ is *well-formed*. (The same holds for (ν_2, S_2).) It follows that $(\nu_1 + t', S_1, \mathtt{M}_1) \perp (\nu_2 + t', S_2, \mathtt{M}_2)$. It remains for us to show that $t \leq t'$ for the hypothesis to hold. We proceed by case analysis on the structure of $\square$:

a. If $\square = ?$, then by Lemma B.4.17, $p \in \mathtt{Wait}(P)$. By Figure 4.1, P must be structured $P = P' \mid Q$ where $P' \equiv p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}$ `after` Q. (We only show this case as it can be applied for the other cases, as discussed in Section 4.2.1.) By Claim B.4.10 of Lemma B.4.1, $S_1 = \left\{ \square_i\, l_i \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I}$. The thesis follows Case 4.

b. If $\square = !$, then since $S_1 = \overline{S_2}$, by Lemma A.4.4, $(\nu_2 + t', \overline{S_2}) \xrightarrow{?m}$. (See Case 7.ii.a.) In the case that P is structured $P = P' \mid Q$ where $P' \equiv p \lhd l\,(w).P''$, then by Definition 4.2.2, $\Phi_t(P)$ is not defined, as sending actions cannot be delayed.

Case 8. If rule [Rec], then $P = \texttt{def}\ X(\vec{v};\vec{r}) = P' \ \texttt{in}\ Q$ and $\Phi_t(P) = \texttt{def}\ X(\vec{v};\vec{r}) = P' \ \texttt{in}\ \Phi_t(Q)$ and it follows Figure C.4 Eq. (5.2.1):

$$\frac{\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta} : \Gamma, \vec{v} : \vec{T}, X : (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta' \vdash P' \blacktriangleright \vec{r} : (\vec{v},\ \vec{S}) \qquad \Gamma, X : (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta \vdash Q \blacktriangleright \Delta}{\Gamma, \theta \vdash \texttt{def}\ X(\vec{v};\vec{r}) = P' \ \texttt{in}\ Q \blacktriangleright \Delta}\ [\texttt{Rec}]$$

By the induction hypothesis: $\Gamma, X : (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta + t \vdash \Phi_t(Q) \blacktriangleright \Delta + t$ where $\Delta + t$ is *well-formed*. The thesis then follows by rule [Rec]:

$$\frac{\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta} : \Gamma, \vec{v} : \vec{T}, X : (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta' \vdash P' \blacktriangleright \vec{r} : (\vec{v},\ \vec{S}) \qquad \Gamma, X : (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta + t \vdash \Phi_t(Q) \blacktriangleright \Delta + t}{\Gamma, \theta + t \vdash \Phi_t(P) \blacktriangleright \Delta + t}\ [\texttt{Rec}]$$

□

Theorem 5.4.3 (Time Step). *Let Δ be fully-balanced and* well-formed. *If $\Gamma; \theta \vdash P \blacktriangleright \Delta$ and $\Phi_t(P)$ is defined, then $\Gamma; \theta + t \vdash \Phi_t(P) \blacktriangleright \Delta + t$ and $\Delta + t$ is fully-balanced and* well-formed.

Proof. We proceed by induction on the derivation of $\Gamma, \theta \vdash P \blacktriangleright \Delta$, analysing the last rule applied of the *type-checking* rules given in Eqs. (5.2.1 to 5.2.6). Since Δ is fully-balanced, only rules [Res] and [Par] are applicable:

Case 1. If rule [Res], then $P = (\nu pq)\ P'$ and following Figure C.4 Eq. (5.2.1):

$$\frac{\Gamma, \theta \vdash P' \blacktriangleright \Delta,\ p : (\nu_1, S_1),\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2 \qquad (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2) \qquad \forall i \in \{1,2\}.\, S_i \text{ well-formed against } \nu_i}{\Gamma, \theta \vdash (\nu pq)\ P' \blacktriangleright \Delta}\ [\texttt{Res}]$$

Let $\Delta' = \Delta, p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2), pq : \mathtt{M}_2$. By the second premise of rule

[Res], it follows that Δ' is *well-formed*. By Definition 5.4.2, it follows that Δ' is fully-balanced. By Definition 4.2.2, $\Phi_t(P) = (\nu pq)\ \Phi_t(P')$. Therefore, by the induction hypothesis:

$$\frac{\Gamma, \theta + t \vdash \Phi_t(P') \blacktriangleright \Delta + t,\ p : (\nu_1 + t, S_1),\ qp : \mathtt{M}_1,\ q : (\nu_2 + t, S_2),\ pq : \mathtt{M}_2 \qquad (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2) \qquad \forall i \in \{1,2\}.\, S_i \text{ well-formed against } \nu_i}{\Gamma, \theta + t \vdash \Phi_t((\nu pq)\ P') \blacktriangleright \Delta + t}\ [\mathtt{Res}]$$

where $\Delta' + t$ is *well-formed* and fully-balanced. The second premise of rule [Res] holds:

- Since $\Delta' + t$ is fully-balanced, by Definition 5.4.2, $(\nu_1 + t, S_1, \mathtt{M}_1) \perp (\nu_2 + t, S_2, \mathtt{M}_2)$.
- The latter coincides with Definition 3.4.2 and holds since $\Delta' + t$ is *well-formed*.

The first premise holds by induction. It remains for us to show that $\Delta + t$ is *well-formed* and fully-balanced. Since $\mathrm{Dom}(\Delta + t) \subseteq \mathrm{Dom}(\Delta' + t)$, it follows that $\Delta + t$ is *well-formed* and fully-balanced, and we have obtained our thesis.

Case 2. If rule [Par], then $P = P' \mid Q$ and it follows Figure C.4 Eq. (5.2.1):

$$\frac{\Gamma, \theta_1 \vdash P' \blacktriangleright \Delta_1 \qquad \Gamma, \theta_2 \vdash Q \blacktriangleright \Delta_2}{\Gamma, \theta \vdash P' \mid Q \blacktriangleright (\Delta_1, \Delta_2)}\ [\mathtt{Par}]$$

where $\theta = (\theta_1, \theta_2)$ and $\Delta = (\Delta_1, \Delta_2)$. It follows that both Δ_1 and Δ_2 are *well-formed*. By Definition 5.4.1, both Δ_1 and Δ_2 are balanced. (Their 'fully-balancedness' does not matter.) By Lemma B.4.30, both Δ_1 and Δ_2 are delayable. By Lemma B.4.28, both Δ_1 and Δ_2 are balanced. By Definition 4.2.2, $\Phi_t(P) = \Phi_t(P') \mid \Phi_t(Q)$. By the induction hypothesis:

$$\frac{\Gamma, \theta_1 + t \vdash \Phi_t(P') \blacktriangleright \Delta_1 + t \qquad \Gamma, \theta_2 + t \vdash \Phi_t(Q) \blacktriangleright \Delta_2 + t}{\Gamma, (\theta_1 + t, \theta_2 + t) \vdash \Phi_t(P' \mid Q) \blacktriangleright (\Delta_1, \Delta_2) + t}\ [\mathtt{Par}]$$

where $\theta + t = (\theta_1 + t, \theta_2 + t)$ and $\Delta + t = (\Delta_1, \Delta_2) + t = (\Delta_1 + t, \Delta_2 + t)$. By Case 6 in Lemma B.5.1, $\Delta + t$ is *well-formed*. It remains for us to prove

that $\Delta + t$ is fully-balanced. Since Δ is fully-balanced, by Case (2) of Definition 5.4.2, it follows that $\forall p$ such that $p \in \mathrm{Dom}(\Delta)$ and $\Delta(p) = (\nu_1, S_1)$, then $\exists \Delta', q, \nu_2, S_2, \mathtt{M}_1, \mathtt{M}_2$ such that $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2), pq : \mathtt{M}_2$ is balanced. (By Case (2) of Definition 5.4.2, the same analogously holds $\forall qp$.) By Definition 5.4.1, $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$. By Condition (4) of Definition 3.4.6, if $\mathtt{M}_1 = \varnothing = \mathtt{M}_2$, then $S_1 = \overline{S_2}$ and $\nu_1 = \nu_2$. In such a case, it follows that $\Delta + t = \Delta' + t, p : (\nu_1 + t, S_1), qp : \mathtt{M}_1, q : (\nu_2 + t, S_2), pq : \mathtt{M}_2$ and $\Delta + t$ is fully-balanced. We now show that it must be that $\mathtt{M}_1 = \varnothing = \mathtt{M}_2$. Suppose by contradiction, that $\mathtt{M}_1 = m; \mathtt{M}_1'$. By Definition 5.4.1, if $\Delta, p : (\nu_1, S_1), qp : m; \mathtt{M}_1'$ then $(\nu_1, S_1) \xrightarrow{?m} (\nu_1', S_1')$ and $\Delta, p : (\nu_1', S_1'), qp : \mathtt{M}_1'$ is balanced. Since $(\nu_1, S_1) \xrightarrow{?m}$, by Lemma B.4.17, $p \in \mathtt{Wait}(P)$. By Lemma B.4.25, $p \in \mathtt{NEQ}(P)$. However, since $\Phi_t(P)$ is defined, by Lemma B.2.1, $\mathtt{Wait}(P) \cap \mathtt{NEQ}(P) = \varnothing$, which is a contradiction. Therefore, all queues must be empty and $\Delta + t$ is fully-balanced. □

B.5.2 Action Steps

Theorem 5.4.4 (Action Step). *Let Δ be balanced and* well-formed. *If $\Gamma; \theta \vdash P \blacktriangleright \Delta$ and $(\theta, P) \rightharpoonup (\theta', P')$ then $\exists \Delta' : \Delta \longrightarrow^* \Delta'$ and $\Gamma; \theta' \vdash P' \blacktriangleright \Delta'$ and Δ' is balanced and* well-formed.

Proof. We proceed by induction on the derivation of the reduction $(\theta, P) \rightharpoonup (\theta', P')$ analysing the last rule applied of those given in Figure C.3:

Case 1. If rule [Send], then $P = p \triangleleft l\,(w).P'' \mid pq : h$ and it follows Figure C.3 Eq. (4.2.2):

$$(\theta,\ p \triangleleft l\,(\mathrm{v}).P'' \mid pq : h) \rightharpoonup (\theta,\ P'' \mid pq : h \cdot l\mathrm{v}) \quad [\mathtt{Send}]$$

where $w = \mathrm{v}$ $P' = P'' \mid pq : h \cdot l\mathrm{v}$ and $\theta' = \theta$. By Claim B.4.2 of Lemma B.4.1, $\theta = (\theta_1, \theta_2)$. $\Delta = (\Delta_1, \Delta_2)$ and:

$$\Gamma, \theta_1 \vdash p \triangleleft l\,(\mathrm{v}).P'' \blacktriangleright \Delta_1 \qquad \Gamma, \theta_2 \vdash pq : h \blacktriangleright \Delta_2 \tag{B.5.1}$$

By Claim B.4.15 of Lemma B.4.1, $\Delta_2 = \Delta_2', pq : \mathtt{M}$ and if $h = \varnothing$ then $\mathtt{M} = \varnothing$. Conversely, if $h \neq \varnothing$ then, by Claim B.4.16 of Lemma B.4.1, $\mathtt{M} \neq \varnothing$. By Claim B.4.7

of Lemma B.4.1, $\Delta_1 = \Delta_1', p : (\nu, \{\mathtt{C}_i\}_{i\in I})$ and $\exists j \in I : \mathtt{C}_j = !l\langle T\rangle\,(\delta, \{\lambda\})\,.S$ and, $l = l_j, ! = \square_j, \nu \models \delta_j$ and:

i. If T_j is base-type, then $\Gamma \vdash \mathtt{v} : T_j$ and $\Gamma, \theta_1 \vdash P'' \blacktriangleright \Delta_1', p : (\nu\,[\lambda_j \mapsto 0]\,, S_j)$. To obtain our thesis we must prove $\exists \Delta' : \Delta \longrightarrow^* \Delta'$, and Δ' is *well-formed*, balanced, and: $\Gamma, (\theta_1, \theta_2) \vdash P'' \mid pq : h \cdot l\mathtt{v} \blacktriangleright \Delta'$. By Claim B.4.2 of Lemma B.4.1, $\Delta' = (\Delta_1'', \Delta_2'')$ and:

$$\Gamma, \theta_1 \vdash P'' \blacktriangleright \Delta_1'' \qquad \Gamma, \theta_2 \vdash pq : h \cdot l\mathtt{v} \blacktriangleright \Delta_2'' \tag{B.5.2}$$

We proceed to show the structure of Δ' necessary to obtain our thesis. By inner induction on the derivation of each in Eq. (B.5.1), analysing the last rule applied. Firstly:

$$\frac{\begin{array}{c}\exists j \in I : \mathtt{C}_j = !l\langle T\rangle\,(\delta, \{\lambda\})\,.S \;\wedge\; \nu \models \delta \;\wedge\; \Gamma \vdash \mathtt{v} : T \;\wedge \\ \Gamma, \theta \vdash P'' \blacktriangleright \Delta_1',\; p : (\nu\,[\lambda \mapsto 0]\,, S)\end{array}}{\Gamma, \theta_1 \vdash p \lhd l\,(\mathtt{v}).P'' \blacktriangleright \Delta_1',\; p : (\nu, \{\mathtt{C}_i\}_{i\in I})}\;[\mathtt{VSend}]$$

Therefore, it must be that $\Delta_1'' = \Delta_1', p : (\nu\,[\lambda \mapsto 0]\,, S)$. By the premise of rule [Send], $\Gamma \vdash \mathtt{v} : T$. By Claim B.4.23 of Lemma B.4.22: $\Gamma, \theta_2 \vdash pq : h \cdot l\mathtt{v} \blacktriangleright \Delta_2',\; pq : \mathtt{M}; l\,\langle T\rangle$ which coincides with the latter of Eq. (B.5.2). Therefore, $\Delta_2'' = \Delta_2', pq : \mathtt{M}; l\,\langle T\rangle$. To summarise:

- $\Delta = \Delta_1',\; p : (\nu, \{\mathtt{C}_i\}_{i\in I}),\; \Delta_2',\; pq : \mathtt{M}$
- $\Delta' = \Delta_1',\; p : (\nu\,[\lambda \mapsto 0]\,, S),\; \Delta_2',\; pq : \mathtt{M}; l\,\langle T\rangle$

Such a reduction is possible via rule [Δ-Send] in Figure B.1:

$$\frac{(\nu, \left\{\square_i\, l_i\,\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I}) \xrightarrow{!m} (\nu', S')}{\Delta'',\; p : (\nu, \left\{\square_i\, l_i\,\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I})\; pq : \mathtt{M} \longrightarrow \Delta'',\; p : (\nu', S')\; pq : \mathtt{M}; m}\;[\Delta\text{-}\mathtt{Send}]$$

where $\{\mathtt{C}_i\}_{i\in I} = \left\{\square_i\, l_i\,\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I}$ and $m = l\,\langle T\rangle$ and $\Delta'' = (\Delta_1', \Delta_2')$. By Case 2 in Lemma A.6.1, it follows that, for some $j \in I$, $\nu \models \delta_j$, $\nu' = \nu\,[\lambda_j \mapsto 0]$ and $S' = S_j$. (Since $\{\mathtt{C}_i\}_{i\in I} = \left\{\square_i\, l_i\,\langle T_i\rangle\,(\delta_i, \lambda_i).S_i\right\}_{i\in I}$, then it follows

the premise of rule [VSend] that $!l\langle T\rangle\,(\delta, \{\lambda\})\,.S = \Box_j l_j\langle T_j\rangle\,\left(\delta_j, \{\lambda\}_j\right).S_j$.) Additionally, it follows that p being *well-formed* indicates that the constraints on actions are structured to preserve 'well-formedness' across such transitions. Therefore, it follows that $\Delta' = \Delta'',\ p : (\nu\,[\lambda \mapsto 0]\,, S)\ pq : \mathtt{M}; m$ is *well-formed*. By Lemma B.3.1, Δ' is balanced. Finally, by rule [Par]:

$$\frac{\Gamma, \theta_1 \vdash P'' \blacktriangleright \Delta_1',\ p : (\nu\,[\lambda \mapsto 0]\,, S) \quad \Gamma, \theta_2 \vdash pq : h \cdot l\mathtt{v} \blacktriangleright \Delta_2',\ pq : \mathtt{M}; l\,\langle T\rangle}{\Gamma, (\theta_1, \theta_2) \vdash P'' \mid pq : h \cdot l\mathtt{v} \blacktriangleright \Delta_1',\ p : (\nu\,[\lambda \mapsto 0]\,, S),\ \Delta_2',\ pq : \mathtt{M}; l\,\langle T\rangle}\ [\mathtt{Par}]$$

ii. If $T_j = (\delta',\ S')$, then $\Delta_1'' = \Delta_1''', b : (\nu', S')$ and, $\nu' \models \delta', \mathtt{v} = b$ and:

$$\Gamma, \theta_1 \vdash P'' \blacktriangleright \Delta_1''',\ p : (\nu\,[\lambda \mapsto 0]\,, S)$$

The rest follows Case 1.i.

In Case 1.i, we have shown that the process of a configuration yielded by a reduction via rule [Send] of Figure C.3 Eq. (4.2.2), is *well-typed* against a session environment Δ', which itself is reachable via a reduction via the rules in Figure B.1, and that the 'well-formedness' and 'balancedness' of Δ' is preserved. Case 1.ii follows similarly, for delegations.

Case 2. If rule [Recv], then $P = p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \mid qp : l\mathtt{v} \cdot h$ and by Figure C.3 Eq. (4.2.2):

$$\frac{j \in I \quad l = l_j}{(\theta,\ p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \mid qp : l\mathtt{v} \cdot h) \rightharpoonup (\theta,\ P_j\,[\mathtt{v}/v_j] \mid qp : h)}\ [\mathtt{Recv}]$$

where $P' = P_j\,[\mathtt{v}/v_j] \mid qp : h$ and $\theta' = \theta$. By Claim B.4.2 of Lemma B.4.1, $\theta = (\theta_1, \theta_2)$ and $\Delta = (\Delta_1, \Delta_2)$ and:

$$\Gamma, \theta_1 \vdash p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i\in I} \blacktriangleright \Delta_1 \qquad \Gamma, \theta_2 \vdash qp : l\mathtt{v} \cdot h \blacktriangleright \Delta_2 \tag{B.5.3}$$

By Claim B.4.16 of Lemma B.4.1, $\Delta_2 = \Delta_2', pq : l\,\langle T\rangle\,;\mathtt{M}$. By Claim B.4.8 of Lemma B.4.1, $\Delta_1 = \Delta_1', p : (\nu, \left\{\Box_k\, l_k\,\langle T_k\rangle\,(\delta_k, \lambda_k).S_k\right\}_{k\in K})$ and $\forall k \in K$ such that $\nu \models \delta_k$, then

$\Box_k = ?$ and $\exists i \in I$ such that:

$$\Gamma, \theta_1 \vdash p^e \triangleright l_i(v_i) : P_i \blacktriangleright \Delta_1',\ p : (\nu, ?l\langle T\rangle\,(\delta, \{\lambda\})\,.S)$$

where $?l\langle T\rangle\,(\delta, \{\lambda\})\,.S = \Box_k l_k\langle T_k\rangle\,(\delta_k, \{\lambda\}_k)\,.S_k$. Hereafter we write $\Box_k l_k\langle T_k\rangle\,(\delta_k, \{\lambda\}_k)\,.S_k$. It follows that in Eq. (B.5.3), $l\mathtt{v} = l_i v_i$. We proceed by inner induction on the derivation, analysing the last rule applied of those in Figure C.5 Eq. (5.2.5):

i. If rule [VRecv], then:

$$\frac{\begin{array}{c} T_k \text{ base-type} \quad \Delta_1' \text{ not } e\text{-reading} \quad \forall t : \nu + t \models \delta_k \Longleftrightarrow t \in e \\ \forall t \in e : \Gamma, v_i : T_k,\ \theta_1 + t \vdash P_i \blacktriangleright \Delta_1' + t,\ p : (\nu + t\,[\lambda_k \mapsto 0]\,, S_k) \end{array}}{\Gamma, \theta_1 \vdash p^e \triangleright l_i(v_i) : P_i \blacktriangleright \Delta_1',\ p : (\nu, \Box_k l_k\langle T_k\rangle\,(\delta_k, \{\lambda\}_k)\,.S_k)}\ [\mathtt{VRecv}] \tag{B.5.4}$$

where $l_k = l_i$ (implicitly). It remains to show that $\exists \Delta'$ such that $\Delta \longrightarrow^* \Delta'$, and Δ' is *well-formed*, balanced, and: $\Gamma, (\theta_1, \theta_2) \vdash P_j\,[\mathtt{v}/v_j] \mid qp : h \blacktriangleright \Delta'$. By Claim B.4.2 of Lemma B.4.1, $\theta' = (\theta_1', \theta_2')$ and $\Delta' = (\Delta_1'', \Delta_2'')$ and:

$$\Gamma, \theta_1' \vdash P_j\,[\mathtt{v}/v_j] \blacktriangleright \Delta_1'' \qquad \Gamma, \theta_2 \vdash qp : h \blacktriangleright \Delta_2''$$

Clearly, following Eq. (B.5.4), $\Delta_1'' = \Delta_1', p : (\nu\,[\lambda_k \mapsto 0]\,, S_k)$. By Claims B.4.15 and B.4.16 of Lemma B.4.1, $\Delta_2'' = \Delta_2', qp : \mathtt{M}$ and, if $h = \varnothing$ then $\mathtt{M} = \varnothing$. (Simiarly, if $h \neq \varnothing$ then $\mathtt{M} \neq \varnothing$.) Therefore, it must be that $\Delta' = \Delta_1',\ p : (\nu\,[\lambda_k \mapsto 0]\,, S_k),\ \Delta_2',\ qp : \mathtt{M}$ and, it follows $\Delta \longrightarrow^* \Delta'$ is possible via rule [Δ-Recv] in Figure B.1:

$$\frac{(\nu, \Box_k\, l_k\, \langle T_k\rangle\, (\delta_k, \lambda_k).S_k, l\,\langle T\rangle\,;\mathtt{M}) \xrightarrow{\tau} (\nu', S', \mathtt{M})}{\Delta'',\ p : (\nu, \Box_k l_k\langle T_k\rangle\,(\delta_k, \{\lambda\}_k)\,.S_k),\ qp : l\,\langle T\rangle\,;\mathtt{M} \longrightarrow \Delta'',\ p : (\nu', S'),\ qp : \mathtt{M}}\ [\Delta\text{-}\mathtt{Recv}]$$

where $\Delta'' = (\Delta_1', \Delta_2')$. By Case 3 in Lemma A.6.1, $l\,\langle T\rangle = l_k\,\langle T_k\rangle$ and, the 'well-formedness' rules in Eq. (3.4.1) are enough to guarantee that 'well-formedness' is preserved across transitions via rules in Figure C.2. Therefore, Δ' is *well-formed*. By Lemma B.3.1, Δ' is also balanced. By induction on the latter derivation in Eq. (B.5.3), analysing the last rule applied. Since T_k is base-type, by rule

[VQue]:

$$\frac{T_k \text{ base-type} \quad \Gamma \vdash \mathtt{v}_i : T_k \quad \Gamma, \theta_2 \vdash qp : h \blacktriangleright \Delta_2',\ qp : \mathtt{M}}{\Gamma, \theta_2 \vdash qp : l_i \mathtt{v}_i \cdot h \blacktriangleright \Delta_2',\ qp : l_k \langle T_k \rangle ; \mathtt{M}}\ [\mathtt{VQue}] \tag{B.5.5}$$

It remains to show that $\Gamma, (\theta_1, \theta_2) \vdash P_j [\mathtt{v}/v_j] \mid qp : h \blacktriangleright \Delta'$. Since $l_k = l_i$ and $l\mathtt{v} = l_i v_i$, then by reformulation of Eq. (B.5.3):

$$\Gamma, \theta_1 \vdash p^e \rhd l_i(v_i) : P_i \blacktriangleright \Delta_1',\ p : (\nu, \square_k l_k \langle T_k \rangle (\delta_k, \{\lambda\}_k) . S_k)$$

$$\Gamma, \theta_2 \vdash qp : l_i \mathtt{v}_i \cdot h \blacktriangleright \Delta_2',\ qp : l_k \langle T_k \rangle ; \mathtt{M}$$

The induction of the reformulated derivations (above) follow the same as Eq. (B.5.4) and Eq. (B.5.5) respectively. Therefore, we conclude by rule [Par]:

$$\frac{\Gamma, \theta_1 \vdash P_j [\mathtt{v}/v_j] \blacktriangleright \Delta_1',\ p : (\nu, p : (\nu [\lambda_k \mapsto 0], S_k)), \qquad \Gamma, \theta_2 \vdash qp : h \blacktriangleright \Delta_2',\ qp : h}{\Gamma, (\theta_1, \theta_2) \vdash P_j [\mathtt{v}/v_j] \mid qp : h \blacktriangleright \Delta_1',\ p : (\nu, p : (\nu [\lambda_k \mapsto 0], S_k)),\ \Delta_2',\ qp : h}\ [\mathtt{Par}]$$

ii. If rule [DRecv], then it follows similarly to Case 2.i.

By Case 2.i, a *well-typed* receiving process may reduce and remain *well-typed* against a session environment resulting from a corresponding reduction via the rules in Figure B.1.

Case 3. If rule [Recv-T], then $P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \mathtt{after}\ Q \mid qp : l\mathtt{v} \cdot h$ and it follows Figure C.3 Eq. (4.2.2):

$$\frac{j \in I \quad l = l_j}{(\theta,\ p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \mathtt{after}\ Q \mid qp : l\mathtt{v} \cdot h) \rightharpoonup (\theta,\ P_j [\mathtt{v}/v_j] \mid qp : h)}\ [\mathtt{Recv\text{-}T}]$$

where $P' = P_j [\mathtt{v}/v_j] \mid qp : h$. The rest follows Case 2.

Case 4. If rule [Set], then $P = \mathtt{set}\, \textcircled{x} . P'$ and following Figure C.3 Eq. (4.2.4):

$$(\theta,\ \mathtt{set}\, \textcircled{x} . P') \rightharpoonup (\theta [x \mapsto 0],\ P')\ \ [\mathtt{Set}]$$

where $\theta' = \theta\,[x \mapsto 0]$. By Claim B.4.11 in Lemma B.4.1, it follows that $\Delta = \Delta'$. Therefore, the thesis coincides with the hypothesis.

Case 5. If rule [Det], then $P = \texttt{delay}\,(\delta)\,.P''$ and following Figure C.3 Eq. (4.2.4):

$$\frac{t \models \delta\,[t/x]}{(\theta,\, \texttt{delay}\,(\delta)\,.P'') \;\rightharpoonup\; (\theta,\, \texttt{delay}\,(t)\,.P'')}\;[\texttt{Det}]$$

where $P' = \texttt{delay}\,(t)\,.P''$. By Claim B.4.12 in Lemma B.4.1, it follows that $\Delta = \Delta'$. Therefore, the thesis coincides with the hypothesis.

Case 6. If rule [If-T], then $P = \texttt{if}\;\delta\;\texttt{then}\;P''\;\texttt{else}\;Q$ and following Figure C.3 Eq. (4.2.4):

$$\frac{\theta \models \delta}{(\theta,\, \texttt{if}\;\delta\;\texttt{then}\;P''\;\texttt{else}\;Q) \;\rightharpoonup\; (\theta,\, P'')}\;[\texttt{If-T}]$$

where $P' = P''$. By Claim B.4.14 in Lemma B.4.1, it follows that $\Delta = \Delta'$. Therefore, the thesis coincides with the hypothesis.

Case 7. If rule [If-F], then it follows simiarly to Case 6, except $\theta \not\models \delta$ and $P' = Q$.

Case 8. If rule [Par-L], then $P = P'' \mid Q$ and following Figure C.3 Eq. (4.2.1):

$$\frac{(\theta_1,\, P'') \;\rightharpoonup\; (\theta_1',\, P''')}{(\theta_1, \theta_2,\, P'' \mid Q) \;\rightharpoonup\; (\theta_1', \theta_2,\, P''' \mid Q)}\;[\texttt{Par-L}]$$

where $\theta' = (\theta_1', \theta_2)$ and $P' = P''' \mid Q$. By Claim B.4.2 of Lemma B.4.1, $\theta = (\theta_1, \theta_2)$ and $\Delta = (\Delta_1, \Delta_2)$ and:

$$\Gamma, \theta_1 \vdash P'' \blacktriangleright \Delta_1 \qquad \Gamma, \theta_2 \vdash Q \blacktriangleright \Delta_2$$

We must show $\exists \Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and $\Gamma, \theta' \vdash P''' \mid Q \blacktriangleright \Delta'$. Again, by Claim B.4.2 of Lemma B.4.1, $\theta' = (\theta_1', \theta_2)$ and $\Delta' = (\Delta_1', \Delta_2')$ and:

$$\Gamma, \theta_1' \vdash P''' \blacktriangleright \Delta_1' \qquad \Gamma, \theta_2' \vdash Q \blacktriangleright \Delta_2'$$

Clearly, $\theta_2' = \theta_2$ and $\Delta_2' = \Delta_2$. By rule [Δ-L] in Figure B.1:

$$\frac{\Delta_1 \longrightarrow \Delta_1'}{\Delta_1,\ \Delta_2 \longrightarrow \Delta_1',\ \Delta_2}\ [\Delta\text{-L}]$$

The thesis follows by the induction hypothesis: since $\Gamma, \theta \vdash P'' \blacktriangleright \Delta_1$, if $(\theta_1,\ P'') \rightharpoonup (\theta_1',\ P''')$ then $\exists \Delta_1'$ such that $\Delta_1 \longrightarrow^* \Delta_1'$ and $\Gamma, \theta_1' \vdash P''' \blacktriangleright \Delta_1'$ and Δ_1' is balanced and *well-formed*.

Case 9. If rule [Par-R], then it follows similarly to Case 8.

Case 10. If rule [Scope], then $P = (\nu pq)\ P''$ and following Figure C.3 Eq. (4.2.1):

$$\frac{(\theta,\ P'') \rightharpoonup (\theta',\ P''')}{(\theta,\ (\nu pq)\ P'') \rightharpoonup (\theta',\ (\nu pq)\ P''')}\ [\texttt{Scope}]$$

where $P' = (\nu pq)\ P'''$. By Claim B.4.3 in Lemma B.4.1, it follows that $\Delta = \Delta'$, as the scope restriction ensures that the inner session (corresponding to pq) is independent of the rest of the session. Therefore, the thesis coincides with the hypothesis.

Case 11. If rule [Def], then $P = \texttt{def}\ X(\vec{v};\vec{r}) = P''\ \texttt{in}\ Q$ and following Figure C.3 Eq. (4.2.3):

$$\frac{(\theta,\ Q) \rightharpoonup (\theta',\ Q')}{(\theta,\ \texttt{def}\ X(\vec{v};\vec{r}) = P''\ \texttt{in}\ Q) \rightharpoonup (\theta',\ \texttt{def}\ X(\vec{v};\vec{r}) = P''\ \texttt{in}\ Q')}\ [\texttt{Def}]$$

where $P' = \texttt{def}\ X(\vec{v};\vec{r}) = P''\ \texttt{in}\ Q'$. By Claim B.4.4 in Lemma B.4.1, it follows that:

$$\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta \vdash Q \blacktriangleright \Delta$$

The thesis follows by the induction hypothesis: since $\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta \vdash Q \blacktriangleright \Delta$, if $(\theta,\ Q) \rightharpoonup (\theta',\ Q')$ then $\exists \Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and $\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta' \vdash Q' \blacktriangleright \Delta'$ and Δ' is balanced and *well-formed*.

Case 12. If rule [Call], then $P = \texttt{def}\ X(\vec{v}';\vec{r}') = P''\ \texttt{in}\ X\langle\vec{v};\vec{r}\rangle \mid Q$ and following Figure C.3 Eq. (4.2.3):

$$(\theta,\ \texttt{def}\ X(\vec{v}';\vec{r}') = P''\ \texttt{in}\ X\langle\vec{v};\vec{r}\rangle \mid Q) \rightharpoonup (\theta,\ \texttt{def}\ X(\vec{v}';\vec{r}') = P''\ \texttt{in}\ P''\,[\vec{v};\vec{r}/\vec{v}';\vec{r}'] \mid Q)\quad [\texttt{Call}]$$

where $P' = \texttt{def}\ X(\vec{v'};\vec{r'}) = P''\ \texttt{in}\ P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q$ and $\theta' = \theta$. By Claims B.4.4 and B.4.5 of Lemma B.4.1: $\theta = (\theta_1, \theta_2)$, $\Delta = (\Delta_1, \Delta_2)$ and:

$$\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta}: \quad \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta'' \vdash P'' \blacktriangleright \vec{r'} : (\vec{v},\ \vec{S}) \tag{B.5.6}$$

$$\Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta_1 \vdash X\langle\vec{v};\vec{r}\rangle \blacktriangleright \Delta_1 \qquad \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta_2 \vdash Q \blacktriangleright \Delta_2 \tag{B.5.7}$$

By Claim B.4.6 of Lemma B.4.1 on Eq. (B.5.7) we have that $\Delta_1 = \vec{r} : (\vec{v}, \vec{S})$ and $(\vec{v}, \vec{S}) \in \boldsymbol{\Delta}$. We must show $\exists\Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and:

$$\Gamma,\ \theta' \vdash \texttt{def}\ X(\vec{v'};\vec{r'}) = P''\ \texttt{in}\ P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q \blacktriangleright \Delta'$$

By Lemma B.4.19 on Eq. (B.5.6): By Lemma B.4.19 on Eq. (B.5.6):

$$\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta}: \quad \Gamma, \vec{v'} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta'' \vdash P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \blacktriangleright \vec{r} : (\vec{v},\ \vec{S})$$

By rule [Par]:

$$\frac{\Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta_1 \vdash P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \blacktriangleright \Delta_1 \quad \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}),\ \theta_2 \vdash Q \blacktriangleright \Delta_2}{\Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta_1, \theta_2 \vdash P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q \blacktriangleright \Delta_1, \Delta_2}\ [\texttt{Par}]$$

By rule [Rec]:

$$\frac{\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta} : \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta'' \vdash P'' \blacktriangleright \vec{r} : (\vec{v},\ \vec{S}) \quad \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta}), \theta \vdash P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q \blacktriangleright \Delta}{\Gamma, \theta \vdash \texttt{def}\ X(\vec{v'};\vec{r'}) = P''\ \texttt{in}\ P''\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q \blacktriangleright \Delta}\ [\texttt{Rec}]$$

The thesis then holds with $\Delta = \Delta'$.

In the cases above we have shown, given a process P that is *well-typed* against a session environment Δ that is balanced and *well-formed*, any *instantaneous* reduction made by the process yields a process P' that is *well-typed* against some session environment Δ' reachable from Δ using the rules in Figure B.1. □

B.5.3 Subject Reduction

Corollary 5.4.5 (Subject Reduction)**.**

$$\textit{If } \varnothing;\theta \vdash P \blacktriangleright \varnothing \textit{ and } (\theta, P) \longrightarrow (\theta', P') \textit{ then } \varnothing;\theta' \vdash P' \blacktriangleright \varnothing.$$

Proof. Since $\Delta = \varnothing$, then by Definition 5.4.2, Δ is fully-balanced. We proceed by induction on the nature of the reduction ($\longrightarrow$):

Case 1. If $\longrightarrow = \rightsquigarrow$ then the thesis follows Theorem 5.4.3.

Case 2. If $\longrightarrow = \rightharpoonup$ then the thesis follows Theorem 5.4.4. □

Appendix C

Supplementary Material

C.1 Generated Erlang Stubs

C.1.1 Example: send_once

```
1  -module(send_once).
2
3  -file("send_once", 1).
4
5  -define(MONITORED, false).
6
7  -define(SHOW_MONITORED,
8    case ?MONITORED of
9      true -> "(monitored)␣";
10     _ -> ""
11   end
12 ).
13
14 -define(SHOW_ENABLED, true).
15
16 -define(SHOW(Str, Args, Data),
17   case ?SHOW_ENABLED of
18     true ->
19       printout(
20         Data,
21         ?SHOW_MONITORED ++ "~p,␣" ++ Str,
22         [?FUNCTION_NAME] ++ Args
23       );
24     _ ->
```

```
25        ok
26    end
27 ).
28
29 -define(SHOW_VERBOSE, ?SHOW_ENABLED and true).
30
31 -define(VSHOW(Str, Args, Data),
32    case ?SHOW_VERBOSE of
33      true ->
34        printout(
35          Data,
36          ?SHOW_MONITORED ++ "(verbose) ~p, " ++ Str,
37          [?FUNCTION_NAME] ++ Args
38        );
39      _ ->
40        ok
41    end
42 ).
43
44 -define(DO_SHOW(Str, Args, Data),
45    printout(
46      Data,
47      ?SHOW_MONITORED ++ "(verbose) ~p, " ++ Str,
48      [?FUNCTION_NAME] ++ Args
49    )
50 ).
51
52 -define(EQ_LIMIT_MS, 10).
53
54 -define(MONITOR_SPEC, #{
55    init => state1_std,
56    map => #{state1_std => #{send => #{data => stop_state}}},
57    timeouts => #{},
58    errors => #{},
59    resets => #{}
60 }).
61
62 -define(PROTOCOL_SPEC, {act, s_data, endP}).
63
```

```
64 -export([init/1, run/1, run/2, start_link/0, start_link/1, stopping/2]).
65
66 -include("stub.hrl").
67
68 %% @doc
69 start_link() -> start_link([]).
70
71 %% @doc
72 start_link(Args) -> stub_start_link(Args).
73
74 %% @doc Called to finish initialising process.
75 %% @see stub_init/1 in ‘stub.hrl‘.
76 %% If this process is set to be monitored (i.e., ?MONITORED) then, in the
       space indicated below setup options for the monitor may be specified,
       before the session actually commences.
77 %% Processes wait for a signal from the session coordinator (SessionID)
       before beginning.
78 init(Args) ->
79   printout("args:\n\t\t~p.", [Args]),
80   {ok, Data} = stub_init(Args),
81   printout("data:\n\t\t~p.", [Data]),
82   CoParty = maps:get(coparty_id, Data),
83   SessionID = maps:get(session_id, Data),
84   case ?MONITORED of
85     true ->
86       CoParty !
87         {
88           self(),
89           setup_options,
90           {printout, #{
91             enabled => true,
92             verbose => true,
93             termination => true
94           }}
95         },
96       CoParty ! {self(), ready, finished_setup},
97       ?VSHOW("finished␣setting␣options␣for␣monitor.", [], Data);
98     _ ->
99       ok
```

```
100   end,
101   ?VSHOW(
102     "waiting␣to␣received␣start␣signal␣from␣session␣(~p).",
103     [SessionID],
104     Data
105   ),
106   receive
107     {SessionID, start} ->
108       ?SHOW(
109         "received␣start␣signal␣from␣session,␣starting.", [], Data
110       ),
111       run(CoParty, Data)
112   end.
113
114 %% @doc Adds default empty list for Data.
115 %% @see run/2.
116 run(CoParty) ->
117   Data = default_stub_data(),
118   ?VSHOW("using␣default␣Data.", [], Data),
119   run(CoParty, Data).
120
121 %% @doc Called immediately after a successful initialisation.
122 %% Add any setup functionality here, such as for the contents of Data.
123 %% @param CoParty is the process ID of the other party in this binary
      session.
124 %% @param Data is a map that accumulates data as the program runs, and is
      used by a lot of functions in ‘stub.hrl‘.
125 %% @see stub.hrl for helper functions and more.
126 run(CoParty, Data) ->
127   ?DO_SHOW("running...\nData:\t~p.\n", [Data], Data),
128   main(CoParty, Data).
129
130 %%% @doc the main loop of the stub implementation.
131 %%% CoParty is the process ID corresponding to either:
132 %%%   (1) the other party in the session;
133 %%%   (2) if this process is monitored, then the transparent monitor.
134 %%% Data is a map that accumulates messages received, sent and keeps track
       of timers as they are set.
```

```
135 %%% Any loops are implemented recursively, and have been moved to their
      own function scope.
136 %%% @see stub.hrl for further details and the functions themselves.
137 main(CoParty, Data) ->
138   Payload1 = get_payload1({data, Data}),
139   CoParty ! {self(), data, Payload1},
140   Data1 = save_msg(send, data, Payload1, Data),
141   ?SHOW("sent data.", [], Data1),
142   stopping(normal, CoParty, Data1).
143
144 %%% @doc Adds default reason 'normal' for stopping.
145 %%% @see stopping/3.
146 stopping(CoParty, Data) ->
147   ?VSHOW("\n\t\tData:\t~p.\n", [Data], Data),
148   stopping(normal, CoParty, Data).
149
150 %%% @doc Writes logs to file before stopping if configured to do so.
151 %%% (enabled by default)
152 %%% @param Reason is either atom like 'normal' or tuple like {error,
      more_details_or_data}.
153 stopping(
154   Reason,
155   CoParty,
156   #{
157     role := #{name := Name, module := Module},
158     options := #{
159       default_log_output_path := Path,
160       output_logs_to_file := true,
161       logs_written_to_file := false
162     } = Options
163   } =
164     Data
165 ) ->
166   {{Year, Month, Day}, {Hour, Min, Sec}} = calendar:now_to_datetime(
167     erlang:timestamp()
168   ),
169   LogFilePath = io_lib:fwrite("~p/dump_~p_~p_~p_~p_~p_~p_~p.log", [
170     Path, Module, Name, Year, Month, Day, Hour, Min, Sec
171   ]),
```

```
172   ?SHOW("writing␣logs␣to␣\"~p\".", [LogFilePath], Data),
173   file:write_file(LogFilePath, io_lib:fwrite("~p.\n", [Data])),
174   stopping(
175     Reason,
176     CoParty,
177     maps:put(
178       options, maps:put(logs_written_to_file, true, Options), Data
179     )
180   );
181 %%% @doc Catches 'normal' reason for stopping.
182 %%% @param Reason is either atom like 'normal' or tuple like {error,
      more_details_or_data}.
183 stopping(
184   normal = Reason,
185   _CoParty,
186   #{role := #{name := Name, module := Module}, session_id := SessionID} =
187     Data
188 ) ->
189   ?SHOW("stopping␣normally.", [], Data),
190   SessionID ! {{Name, Module, self()}, stopping, Reason, Data},
191   exit(normal);
192 %%% @doc stopping with error.
193 %%% @param Reason is either atom like 'normal' or tuple like {error,
      Reason, Details}.
194 %%% @param CoParty is the process ID of the other party in this binary
      session.
195 %%% @param Data is a list to store data inside to be used throughout the
      program.
196 stopping(
197   {error, Reason, Details} = Info,
198   _CoParty,
199   #{role := #{name := Name, module := Module}, session_id := SessionID} =
200     Data
201 ) when is_atom(Reason) ->
202   ?SHOW(
203     "error,␣stopping...\n\t\tReason:\t~p,\n\t\tDetails:\t~p,\n\t\tCoParty
      :\t~p,\n\t\tData:\t~p.\n",
204     [Reason, Details, _CoParty, Data],
205     Data
```

```
206   ),
207   SessionID ! {{Name, Module, self()}, stopping, Info, Data},
208   erlang:error(Reason, Details);
209 %%% @doc Adds default Details to error.
210 stopping({error, Reason}, CoParty, Data) when is_atom(Reason) ->
211   ?VSHOW(
212     "error, stopping...\n\t\tReason:\t~p,\n\t\tCoParty:\t~p,\n\t\tData:\t~
      p.\n",
213     [Reason, CoParty, Data],
214     Data
215   ),
216   stopping({error, Reason, []}, CoParty, Data);
217 %%% @doc stopping with Unexpected Reason.
218 stopping(
219   Reason,
220   _CoParty,
221   #{role := #{name := Name, module := Module}, session_id := SessionID} =
222     Data
223 ) when is_atom(Reason) ->
224   ?SHOW(
225     "unexpected stop...\n\t\tReason:\t~p,\n\t\tCoParty:\t~p,\n\t\tData:\t~
      p.\n",
226     [Reason, _CoParty, Data],
227     Data
228   ),
229   SessionID ! {{Name, Module, self()}, stopping, Reason, Data},
230   exit(Reason).
231
232 get_payload1({_Args, _Data}) ->
233   extend_with_functionality_for_obtaining_payload.
```

C.2 Restated Theory

This chapter contains restated definitions and figures of the theory of TOAST.

C.2.1 TOAST Types

$$\delta ::= \texttt{true} \mid x > n \mid x = n \mid x - y > n \mid x - y = n \mid \neg\delta \mid \delta_1 \wedge \delta_2 \qquad (\text{where } n \in \mathbb{N}) \tag{3.1.1}$$

$$\begin{array}{rcl@{\qquad}rcl} S & ::= & \{\mathtt{C}_i\}_{i\in I} \mid \mu\alpha.S \mid \alpha \mid \mathtt{end} & \mathtt{C} & ::= & \Box\, l\,\langle T\rangle\,(\delta,\lambda).S \\ T & ::= & (\delta,\, S) \mid \mathtt{Unit} \mid \mathtt{Nat} \mid \mathtt{Bool} \mid \mathtt{String} \mid \ldots & \Box & ::= & ! \mid ? \end{array} \tag{3.2.1}$$

$$\ell ::= \Box m \mid t \mid \tau \qquad \Box ::= ! \mid ? \qquad m ::= l\,\langle T\rangle \qquad \mathtt{M} ::= \varnothing \mid m;\mathtt{M} \tag{3.3.1}$$

Definition 3.3.1 (Future-enabled Configurations)**.** Configuration (ν, S) is *fe*, written either as '$(\nu, S) \overset{\Box}{\Rightarrow}$' or '$(\nu, S)$ is $\mathtt{fe}$', iff $\exists t, m$ s.t. $(\nu, S) \xrightarrow{t\ \Box m}$ via the rules in Eq. (3.3.2).

Definition 3.4.1 (Type Duality)**.** Given a type S, we define its *dual* type $\overline{S}$ as follows:

$$\overline{\mathtt{end}} = \mathtt{end} \qquad \overline{\alpha} = \alpha \qquad \overline{\mu\alpha.S} = \mu\alpha.\overline{S} \qquad \overline{?} = ! \qquad \overline{!} = ?$$
$$\overline{\left\{\Box_i\, l_i\,\langle T_i\rangle\,(\delta_i,\lambda_i).S_i\right\}_{i\in I}} = \left\{\overline{\Box}_i\, l_i\langle T_i\rangle(\delta_i,\lambda_i).\overline{S}_i\right\}_{i\in I}$$

Definition 3.4.2 (Well-formed Configurations)**.** Given (ν, S), S is *well-formed* against ν if, $\exists\delta$ s.t. $\varnothing;\delta \vdash S$ and $\nu \models \delta$. A type S is *well-formed* if it is *well-formed* against ν_0.

Definition 3.4.4 (Type Progress)**.** A configuration with queues $(\nu, S, \mathtt{M})$ is *final* if both $S \overset{\mathtt{unfold}}{\equiv} \mathtt{end}$ and $\mathtt{M} = \varnothing$. A system $\mathbf{S}_1 \mid \mathbf{S}_2$ *satisfies progress* if, for all $\mathbf{S}_1' \mid \mathbf{S}_2'$ reachable from $\mathbf{S}_1 \mid \mathbf{S}_2$ either:

1. $(\nu_1', S_1', \mathtt{M}_1')$ and $(\nu_2', S_2', \mathtt{M}_2')$ are *final*
2. or, $\exists t \in \mathbb{R}_{\geq 0}$ such that $(\nu_1', S_1', \mathtt{M}_1') \mid (\nu_2', S_2', \mathtt{M}_2') \xrightarrow{t\ \tau}$.

Definition 3.4.6 (Compatibility)**.** We write $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$ iff $(\nu_1, S_1, \mathtt{M}_1)$ and $(\nu_2, S_2, \mathtt{M}_2)$ are *compatible*. Given $(\nu_1, S_1, \mathtt{M}_1)$ and $(\nu_2, S_2, \mathtt{M}_2)$, we define *compatibility* as the largest relation satisfying each of the following:

1. $(\mathtt{M}_1 = \varnothing)$ or $(\mathtt{M}_2 = \varnothing)$
2. $(\mathtt{M}_1 = m;\mathtt{M}_1') \implies ((\nu_1, S_1) \xrightarrow{?m} (\nu_1', S_1')$ and $(\nu_1', S_1', \mathtt{M}_1') \perp (\nu_2, S_2, \mathtt{M}_2))$
3. $(\mathtt{M}_2 = m;\mathtt{M}_2') \implies ((\nu_2, S_2) \xrightarrow{?m} (\nu_2', S_2')$ and $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2', S_2', \mathtt{M}_2'))$
4. $(\mathtt{M}_1 = \varnothing = \mathtt{M}_2) \implies (S_1 = \overline{S_2}$ and $\nu_1 = \nu_2)$

$$\frac{\begin{array}{rl} \forall i \in I : A;\ \gamma_i \vdash S_i \ \wedge\ \delta_i\,[\lambda_i \mapsto 0] \models \gamma_i & \text{(feasibility)} \\ \forall i,j \in I : i \neq j \implies \delta_i \wedge \delta_j \models \texttt{false} \ \vee \Box_i = \Box_j & \text{(mixed-choice)} \\ \forall i \in I : T_i = (\delta',\ S') \implies \varnothing;\ \gamma' \vdash S' \ \wedge\ \delta' \models \gamma' & \text{(delegation)} \end{array}}{A;\ \downarrow \bigvee_{i \in I} \delta_i \vdash \left\{ \Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I}}\ [\texttt{choice}]$$

$$\frac{}{A;\ \texttt{true} \vdash \texttt{end}}\ [\texttt{end}] \qquad \frac{A, \alpha : \delta;\ \delta \vdash S}{A;\ \delta \vdash \mu\alpha.S}\ [\texttt{rec}] \qquad \frac{}{A, \alpha : \delta;\ \delta \vdash \alpha}\ [\texttt{var}] \tag{3.4.1}$$

FIGURE C.1: (Restated) TOAST Well-formedness Rules

$$\frac{\nu \models \delta_j \quad m = l_j\,\langle T_j \rangle \quad j \in I}{(\nu, \left\{ \Box_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i \right\}_{i \in I}) \xrightarrow{\Box_j m} (\nu\,[\lambda_j \mapsto 0],\ S_j)}\ [\texttt{act}] \tag{3.3.2}$$

$$\frac{(\nu,\ S\,[\mu\alpha.S/\alpha]) \xrightarrow{\ell} (\nu',\ S')}{(\nu,\ \mu\alpha.S) \xrightarrow{\ell} (\nu',\ S')}\ [\texttt{unfold}] \qquad (\nu,\ S) \xrightarrow{t} (\nu + t,\ S)\ \ [\texttt{tick}]$$

$$\frac{(\nu,\ S) \xrightarrow{!m} (\nu',\ S')}{(\nu,\ S,\ \texttt{M}) \xrightarrow{!m} (\nu',\ S',\ \texttt{M})}\ [\texttt{send}] \qquad \frac{(\nu,\ S) \xrightarrow{?m} (\nu',\ S')}{(\nu,\ S,\ m;\ \texttt{M}) \xrightarrow{\tau} (\nu',\ S',\ \texttt{M})}\ [\texttt{recv}]$$

$$(\nu,\ S,\ \texttt{M}) \xrightarrow{?m} (\nu,\ S,\ \texttt{M};\ m)\ \ [\texttt{que}] \tag{3.3.3}$$

$$\frac{\begin{array}{rl} (\nu,\ S) \xrightarrow{t} (\nu',\ S') & \text{(configuration)} \\ ((\nu,\ S)\ \text{is}\ \texttt{fe}) \implies ((\nu',\ S')\ \text{is}\ \texttt{fe}) & \text{(persistency)} \\ \forall t' < t : (\nu + t',\ S,\ \texttt{M}) \not\xrightarrow{\tau} & \text{(urgency)} \end{array}}{(\nu,\ S,\ \texttt{M}) \xrightarrow{t} (\nu',\ S',\ \texttt{M})}\ [\texttt{time}]$$

$$\frac{\mathbf{S}_1 \xrightarrow{!m} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{?m} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2'}\ [\texttt{com-l}] \qquad \frac{\mathbf{S}_1 \xrightarrow{\tau} \mathbf{S}_1'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2}\ [\texttt{par-l}]$$

$$\frac{\mathbf{S}_1 \xrightarrow{t} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{t} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{t} \mathbf{S}_1' \mid \mathbf{S}_2'}\ [\texttt{wait}]$$

$$\frac{\mathbf{S}_1 \xrightarrow{?m} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{!m} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2'}\ [\texttt{com-r}] \qquad \frac{\mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1 \mid \mathbf{S}_2'}\ [\texttt{par-r}] \tag{3.3.4}$$

FIGURE C.2: (Restated) An LTS for TOAST

C.2.2 TOAST Processes

$$
\begin{array}{lll}
P, Q ::= \texttt{set}\,\textcircled{x}.P & \mid & (\nu pq)\ P \\
\quad \mid\ p \lhd l\,(w).P & \mid & P \mid Q \\
\quad \mid\ p^{e} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} & \mid & \texttt{0} \\
\quad \mid\ p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}\ \texttt{after}\ Q & \mid & pq : h \\
\quad \mid\ \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q & w ::= v \mid \texttt{v} & \\
\quad \mid\ \texttt{delay}\,(d)\,.P & e ::= \diamond n \mid \infty \mid \leq 0 \qquad (n \in \mathbb{R}_{>0}) & \\
\quad \mid\ \texttt{delay}\,(t)\,.P & \diamond ::= \ < \mid \leq & \\
\quad \mid\ \texttt{def}\ X(\vec{v};\vec{r}) = P\ \texttt{in}\ Q & c ::= \textcircled{x} \diamond n \mid n \diamond \textcircled{x} \mid \textcircled{x} = n & \\
\quad \mid\ X\langle \vec{v};\vec{r} \rangle & d ::= \texttt{t} \diamond n \mid n \diamond \texttt{t} \mid \texttt{true} & \\
 & h ::= \ \varnothing \mid h \cdot l\texttt{v} &
\end{array}
\tag{4.1.1}
$$

Definition 4.1.1 (Well-formed Process)**.** The function $\texttt{wf}\big(P\big)$ is defined inductively as:

$$
\texttt{wf}\big(P\big) = \begin{cases}
\texttt{true} & \text{if } P \in \left\{ \texttt{0},\ X\langle \vec{v};\vec{r} \rangle,\ qp : h \right\} \\
\texttt{wf}\big(P'\big) \wedge \big(\texttt{fq}\big(P'\big) = \{pq, qp\}\big) & \text{if } P = (\nu pq)\ P' \\
\texttt{wf}\big(P'\big) \wedge \big(\texttt{fq}\big(P'\big) = \varnothing\big) & \text{if } P \in \left\{ \begin{array}{l} p \lhd l\,(w).P',\ \texttt{set}\,\textcircled{x}.P', \\ \texttt{delay}\,(d)\,.P',\ \texttt{delay}\,(t)\,.P' \end{array} \right\} \\
\texttt{wf}\big(P'\big) \wedge \texttt{wf}\big(Q\big) & \text{if } P \in \left\{ \begin{array}{l} \texttt{if}\ c\ \texttt{then}\ P'\ \texttt{else}\ Q, \\ \texttt{def}\ X(\vec{v};\vec{r}) = P'\ \texttt{in}\ Q \end{array} \right\} \\
\texttt{wf}\big(P'\big) \wedge \texttt{wf}\big(Q\big) \wedge \big(\texttt{ft}\big(P'\big) \cap \texttt{ft}\big(Q\big) = \varnothing\big) & \text{if } P = P' \mid Q \\
\bigwedge_{i \in I} \texttt{wf}\big(P_i\big) \wedge \big(\texttt{fq}\big(P_i\big) = \varnothing\big) & \text{if } P = p^{e} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \\
\texttt{wf}\big(p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}\big) \wedge \texttt{wf}\big(Q\big) & \text{if } P = p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I}\ \texttt{after}\ Q
\end{cases}
$$

Definition 4.1.3 (Structural Congruence)**.** We define $P \equiv Q$ as the smallest relation on processes:

$$
\begin{array}{l}
P \mid \texttt{0} \equiv P \qquad (P \mid Q) \equiv (Q \mid P) \qquad (P_1 \mid P_2) \mid Q \equiv (P_1 \mid Q) \mid P_2 \qquad P \mid Q \equiv Q \mid P \\
(\nu pq)\ \texttt{0} \equiv \texttt{0} \qquad \texttt{delay}\,(0)\,.P \equiv P \qquad (\nu pq)\ (\nu ab)\ P \equiv (\nu ab)\ (\nu pq)\ P \\
(\nu pq)\ P \equiv (\nu qp)\ P \qquad (\nu pq)\ P \mid Q \equiv (\nu pq)\ (P \mid Q) \quad \text{if } pq \notin \texttt{fn}(Q) \\
(\texttt{def}\ X(\vec{v};\vec{r}) = P\ \texttt{in}\ P') \mid Q \equiv \texttt{def}\ X(\vec{v};\vec{r}) = P\ \texttt{in}\ (P' \mid Q) \quad \text{if } X\langle \vec{v'};\vec{r'} \rangle \notin \texttt{fpv}\big(Q\big)
\end{array}
$$

where functions $\texttt{fn}(Q)$ and $\texttt{fpv}(Q)$ return the set of free names and process variables in Q, respectively. (Formally defined in Definitions B.1.3 and B.1.4.)

C.2.3 TOAST Reduction Rules

Definition 4.2.1 (Timer Environment)**.** Recall $\mathbb{T}$ is the set of timers, ranged over by $\textcircled{x}$, $\textcircled{y}$ and $\textcircled{z}$. A *timer environment* θ is a linear map $\theta : \mathbb{T} \mapsto \mathbb{R}_{\geq 0}$, from timers to valuations, defined: $\theta ::= \varnothing \mid \theta,\ \textcircled{x} : n$ where $n \in \mathbb{R}_{\geq 0}$.

We define $\theta + t = \{\textcircled{x} \mapsto \theta(\textcircled{x}) + t \mid \textcircled{x} \in \mathbb{T}\}$ so that time may pass over θ and the timers contained within. We define $\theta\,[\textcircled{x} \mapsto 0]$ to be the map $\theta\,[\textcircled{x} \mapsto 0]\,(\textcircled{y}) =$ if $(\textcircled{x} = \textcircled{y})$ 0 else $\theta(\textcircled{y})$. We write $\mathrm{Dom}(\theta)$ for the domain of θ, and θ_1, θ_2 for $\theta_1 \cup \theta_2$ when $\mathrm{Dom}(\theta_1) \cap \mathrm{Dom}(\theta_2) = \varnothing$. We write $\theta \models c$ to denote that θ satisfies c, formally: $(\theta \models c) = (\forall \textcircled{x} \in \texttt{fn}(c).c\,[^{\theta(\textcircled{x})}/_{\textcircled{x}}])$. (This is similar to the definition of $\nu \models \delta$.)

Definition 4.2.2 (Time Passing Function)**.** The time-passing function $\Phi_t(P)$ is a partial function only defined for the cases below:

$$\Phi_t\left(p^{\diamond n} \rhd \{\mathtt{B}_i\}_{i\in I}\ \texttt{after}\ Q\right) = \begin{cases} p^{\diamond n-t} \rhd \{\mathtt{B}_i\}_{i\in I}\ \texttt{after}\ Q & \text{if } (t \diamond n) \\ \Phi_{t-n}(Q) & \text{otherwise} \end{cases}$$

$$\Phi_t\left(p^e \rhd \{\mathtt{B}_i\}_{i\in I}\right) = \begin{cases} p^e \rhd \{\mathtt{B}_i\}_{i\in I} & \text{if } e = \infty \\ p^{\diamond n-t} \rhd \{\mathtt{B}_i\}_{i\in I} & \text{if } e = \diamond n \text{ and } (t \diamond n) \end{cases}$$

$$\Phi_t\left(\texttt{delay}\,(t')\,.P\right) = \begin{cases} \texttt{delay}\,(t'-t)\,.P & \text{if } t' \geq t \\ \Phi_{t-t'}(P) & \text{otherwise} \end{cases}$$

$$\Phi_t(P_1 \mid P_2) = \Phi_t(P_1) \mid \Phi_t(P_2) \quad \text{if } (\texttt{Wait}(P_i) \cap \texttt{NEQ}(P_j) = \varnothing) \text{ and } i \neq j \in \{1,2\}$$

$$\Phi_t\left(\texttt{def}\ X(\vec{v};\vec{r}) = P\ \texttt{in}\ Q\right) = \texttt{def}\ X(\vec{v};\vec{r}) = P\ \texttt{in}\ \Phi_t(Q) \qquad \Phi_0(P) = P$$

$$\Phi_t\left((\nu pq)\ P\right) = (\nu pq)\ \Phi_t(P) \qquad \Phi_t(pq : h) = pq : h \qquad \Phi_t(\mathtt{0}) = \mathtt{0}$$

See the rules in Figure C.3.

$$\frac{P \equiv P' \quad (\theta,\, P') \longrightarrow (\theta',\, Q') \quad Q \equiv Q'}{(\theta, P) \longrightarrow (\theta',\, Q)}\ [\texttt{Str}]$$

$$\frac{(\theta,\, P) \rightharpoonup (\theta',\, P')}{(\theta,\, (\nu pq)\ P) \rightharpoonup (\theta',\, (\nu pq)\ P')}\ [\texttt{Scope}]$$

$$\frac{(\theta_1,\, P) \rightharpoonup (\theta_1',\, P')}{(\theta_1, \theta_2,\, P \mid Q) \rightharpoonup (\theta_1', \theta_2,\, P' \mid Q)}\ [\texttt{Par-L}]$$

$$\frac{(\theta_2,\, Q) \rightharpoonup (\theta_2',\, Q')}{(\theta_1, \theta_2,\, P \mid Q) \rightharpoonup (\theta_1, \theta_2',\, P \mid Q')}\ [\texttt{Par-R}] \tag{4.2.1}$$

$$(\theta,\, p \lhd l\,(\mathtt{v}).P \mid pq : h) \rightharpoonup (\theta,\, P \mid pq : h \cdot l\mathtt{v}) \quad [\texttt{Send}]$$

$$\frac{j \in I \quad l = l_j}{(\theta,\, p^e \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \mid qp : l\mathtt{v} \cdot h) \rightharpoonup (\theta,\, P_j\,[\mathtt{v}/v_j] \mid qp : h)}\ [\texttt{Recv}]$$

$$\frac{j \in I \quad l = l_j}{(\theta,\, p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{ after } Q \mid qp : l\mathtt{v} \cdot h) \rightharpoonup (\theta,\, P_j\,[\mathtt{v}/v_j] \mid qp : h)}\ [\texttt{Recv-T}] \tag{4.2.2}$$

$$\frac{(\theta,\, Q) \rightharpoonup (\theta',\, Q')}{(\theta,\, \texttt{def } X(\vec{v};\vec{r}) = P \texttt{ in } Q) \rightharpoonup (\theta',\, \texttt{def } X(\vec{v};\vec{r}) = P \texttt{ in } Q')}\ [\texttt{Def}]$$

$$(\theta,\, \texttt{def } X(\vec{v'};\vec{r'}) = P \texttt{ in } X\langle \vec{v};\vec{r} \rangle \mid Q) \rightharpoonup (\theta,\, \texttt{def } X(\vec{v'};\vec{r'}) = P \texttt{ in } P\,[\vec{v};\vec{r}/\vec{v'};\vec{r'}] \mid Q)\ [\texttt{Call}] \tag{4.2.3}$$

$$(\theta, \textcircled{x} : n,\, \texttt{set}\,\textcircled{x}.P) \rightharpoonup (\theta, \textcircled{x} : 0,\, P)\ [\texttt{Reset}]$$

$$\frac{\theta \models c}{(\theta,\, \texttt{if } c \texttt{ then } P \texttt{ else } Q) \rightharpoonup (\theta,\, P)}\ [\texttt{If-T}]$$

$$\frac{\theta \not\models c}{(\theta,\, \texttt{if } c \texttt{ then } P \texttt{ else } Q) \rightharpoonup (\theta,\, Q)}\ [\texttt{If-F}] \tag{4.2.4}$$

$$\frac{t \models d\,[t/\mathtt{t}]}{(\theta,\, \texttt{delay}\,(d)\,.P) \rightharpoonup (\theta,\, \texttt{delay}\,(t)\,.P)}\ [\texttt{Det}] \qquad (\theta, P) \rightsquigarrow (\theta + t,\, \Phi_t(P))\ [\texttt{Delay}]$$

FIGURE C.3: (Restated) Reduction Rules for TOAST Processes

$$\texttt{Wait}(P) = \begin{cases} \{p\} & \text{if } P \in \left\{ \begin{array}{l} p^{\diamond n} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \texttt{ after } Q, \\ p^{e} \rhd \left\{ l_i(v_i) : P_i \right\}_{i \in I} \end{array} \right\} \\ \texttt{Wait}(Q) \setminus \{p, q\} & \text{if } P = (\nu pq)\, Q \\ \texttt{Wait}(Q) & \text{if } P = \texttt{def } X(\vec{v}; \vec{r}) = P' \texttt{ in } Q \\ \texttt{Wait}(P') \cup \texttt{Wait}(Q) & \text{if } P = P' \mid Q \\ \varnothing & \text{otherwise} \end{cases}$$

(A) Definition of $\texttt{Wait}(P)$.

$$\texttt{NEQ}(P) = \begin{cases} \{p\} & \text{if } P = qp : h \wedge h \neq \varnothing \\ \texttt{NEQ}(Q) \setminus \{p, q\} & \text{if } P = (\nu pq)\, Q \\ \texttt{NEQ}(Q) & \text{if } P = \texttt{def } X(\vec{v}; \vec{r}) = P' \texttt{ in } Q \\ \texttt{NEQ}(P') \cup \texttt{NEQ}(Q) & \text{if } P = P' \mid Q \\ \varnothing & \text{otherwise} \end{cases}$$

(B) Definition of $\texttt{NEQ}(P)$.

FIGURE 4.1: (Restated) Definition of $\texttt{Wait}(P)$ and $\texttt{NEQ}(P)$.

C.2.4 TOAST Typing System

$$\begin{aligned}
\Gamma &::= \varnothing \mid \Gamma,\ v : T \mid \Gamma,\ X\colon (\vec{T}; \boldsymbol{\theta}; \boldsymbol{\Delta}) \\
\theta &::= \varnothing \mid \theta,\ \textcircled{x} : n \qquad \text{(recall Definition 4.2.1, } n \in \mathbb{R}_{\geq 0}) \\
\Delta &::= \varnothing \mid \Delta,\ p : (\nu, S) \mid \Delta,\ qp : \mathtt{M}
\end{aligned} \tag{5.1.1}$$

Definition 5.1.1 (t-reading Δ)**.** Δ is t-reading if, $\exists t' < t, p \in \mathrm{Dom}(\Delta)$ such that $\Delta(p) = (\nu, S) \implies (\nu + t', S) \xrightarrow{?m}$. We write Δ is $\diamond t$-reading if, $\exists t' \diamond t, p \in \mathrm{Dom}(\Delta)$ such that $\Delta(p) = (\nu, S) \implies (\nu + t', S) \xrightarrow{?m}$.

Definition 5.1.2 (Well-formed Δ)**.** Δ is *well-formed* if, for all $p \in \mathrm{Dom}(\Delta)$ such that $\Delta = \Delta', p : (\nu, S) \implies (\nu, S)$ is *well-formed* by Definition 3.4.2.

Definition 5.4.1 (Balanced Δ)**.** Let Bal be the set containing all session environments Δ, such that if $\Delta \in$ Bal, then Δ adheres to the following:

1. $\Delta = \Delta', p : (\nu, S), qp : m;\mathtt{M} \implies (\nu, S) \xrightarrow{?m} (\nu', S')$ and $\Delta', p : (\nu', S'), qp : \mathtt{M} \in$ Bal.
2. $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2) \implies \exists \mathtt{M}_2 : (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$.
3. $\Delta = \Delta', p : (\nu_1, S_1), qp : \mathtt{M}_1, q : (\nu_2, S_2), pq : \mathtt{M}_2 \implies (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$.

Definition 5.4.2 (Fully-Balanced Δ)**.** A balanced Δ is said to be fully-balanced if:

1. $\Delta = \Delta',\ p : (\nu_1, S_1) \implies$
$$\exists \Delta'',\ q,\ \nu_2,\ S_2,\ \mathtt{M}_1,\ \mathtt{M}_2 \quad \text{s.t.} \quad \Delta' = \Delta'',\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2$$
2. $\Delta = \Delta',\ qp : \mathtt{M}_1 \implies$
$$\exists \Delta'',\ \nu_1,\ S_1,\ \nu_2,\ S_2,\ \mathtt{M}_2 \quad \text{s.t.} \quad \Delta' = \Delta'',\ p : (\nu_1, S_1),\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2$$

$$\Gamma;\theta \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}] \qquad \Gamma;\theta \vdash pq : \varnothing \blacktriangleright pq : \varnothing \quad [\mathtt{Empty}]$$

$$\frac{\Gamma;\theta \vdash P \blacktriangleright \Delta}{\Gamma;\theta \vdash P \blacktriangleright \Delta,\ p : (\nu, \mathtt{end})}\ [\mathtt{Weak}] \qquad \frac{\Gamma;\theta_1 \vdash P \blacktriangleright \Delta_1 \quad \Gamma;\theta_2 \vdash Q \blacktriangleright \Delta_2}{\Gamma;\theta_1,\theta_2 \vdash P \mid Q \blacktriangleright \Delta_1,\Delta_2}\ [\mathtt{Par}]$$

$$\frac{\begin{array}{c}\Gamma;\theta \vdash P \blacktriangleright \Delta,\ p : (\nu_1, S_1),\ qp : \mathtt{M}_1,\ q : (\nu_2, S_2),\ pq : \mathtt{M}_2 \\ (\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2) \qquad \forall i \in \{1,2\}.\, S_i \text{ well-formed against } \nu_i\end{array}}{\Gamma;\theta \vdash (\nu pq)\ P \blacktriangleright \Delta}\ [\mathtt{Res}]$$

$$\frac{\begin{array}{c}\forall(\vec{v},\ \vec{S}) \in \boldsymbol{\Delta}, \theta' \in \boldsymbol{\theta} : \Gamma, \vec{v} : \vec{T}, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta});\ \theta' \vdash P \blacktriangleright \vec{r} : (\vec{v},\ \vec{S}) \\ \Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta});\ \theta \vdash Q \blacktriangleright \Delta\end{array}}{\Gamma;\theta \vdash \mathtt{def}\ X(\vec{v};\vec{r}) = P\ \mathtt{in}\ Q \blacktriangleright \Delta}\ [\mathtt{Rec}]$$

$$\frac{(\vec{v},\vec{S}) \in \boldsymbol{\Delta} \quad \theta \in \boldsymbol{\theta} \quad \forall i : \Gamma \vdash \vec{v}_i : \vec{T}_i}{\Gamma, X\colon (\vec{T};\boldsymbol{\theta};\boldsymbol{\Delta});\ \theta \vdash X\langle\vec{v};\vec{r}\rangle \blacktriangleright \vec{r} : (\vec{v},\vec{S})}\ [\mathtt{Var}] \tag{5.2.1}$$

$$\frac{\Gamma;\theta, \textcircled{x} : 0 \vdash P \blacktriangleright \Delta}{\Gamma;\theta, \textcircled{x} : n \vdash \mathtt{set}\ \textcircled{x}.P \blacktriangleright \Delta}\ [\mathtt{Timer}]$$

$$\frac{\forall t \in d : \Gamma;\theta \vdash \mathtt{delay}\,(t)\,.P \blacktriangleright \Delta}{\Gamma;\theta \vdash \mathtt{delay}\,(d)\,.P \blacktriangleright \Delta}\ [\mathtt{Del}\text{-}d]$$

$$\frac{\Gamma;\theta + t \vdash P \blacktriangleright \Delta + t \quad \Delta \text{ not } t\text{-reading}}{\Gamma;\theta \vdash \mathtt{delay}\,(t)\,.P \blacktriangleright \Delta}\ [\mathtt{Del}\text{-}t] \tag{5.2.2}$$

$$\frac{\theta \models c \quad \Gamma;\theta \vdash P \blacktriangleright \Delta}{\Gamma;\theta \vdash \mathtt{if}\ c\ \mathtt{then}\ P\ \mathtt{else}\ Q \blacktriangleright \Delta}\ [\mathtt{IfTrue}]$$

$$\frac{\theta \not\models c \quad \Gamma;\theta \vdash Q \blacktriangleright \Delta}{\Gamma;\theta \vdash \mathtt{if}\ c\ \mathtt{then}\ P\ \mathtt{else}\ Q \blacktriangleright \Delta}\ [\mathtt{IfFalse}]$$

FIGURE C.4: (Restated) A Typing System for TOAST (standard & time-sensitive)

$$\frac{\begin{array}{c}\exists i \in I : (l = l_i) \wedge (\nu \models \delta_i) \wedge (T_i \text{ base-type}) \wedge (\Gamma \vdash w : T_i) \wedge \\ (\square_i = !) \wedge \Gamma; \theta \vdash P \blacktriangleright \Delta, p : (\nu\,[\lambda_i \mapsto 0]\,, S_i)\end{array}}{\Gamma; \theta \vdash p \lhd l\,(w).P \blacktriangleright \Delta,\ p : (\nu, \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I})}\ [\texttt{VSend}]$$

$$\frac{\begin{array}{c}\exists i \in I : (l = l_i) \wedge (\nu \models \delta_i) \wedge (T_i = (\delta', S')) \wedge (\nu' \models \delta') \wedge \\ (\square_i = !) \wedge \Gamma; \theta \vdash P \blacktriangleright \Delta, p : (\nu\,[\lambda_i \mapsto 0]\,, S_i)\end{array}}{\Gamma; \theta \vdash p \lhd l\,(b).P \blacktriangleright \Delta,\ p : (\nu, \left\{\square_i\, l_i\, \langle T_i \rangle\, (\delta_i, \lambda_i).S_i\right\}_{i \in I}),\ b : (\nu', S')}\ [\texttt{DSend}] \tag{5.2.3}$$

$$\frac{\begin{array}{c}\neg(|J| = |I| = 1) \\ \forall j \in J : (\nu \models \delta_j) \implies (\square_j = ?) \wedge \exists i \in I : (l_i = l_j) \wedge \\ \Gamma; \theta \vdash p^e \rhd l_i(v_i) : P_i \blacktriangleright \Delta,\ p : (\nu, \square_j\, l_j\, \langle T_j \rangle\, (\delta_j, \lambda_j).S_j)\end{array}}{\Gamma; \theta \vdash p^e \rhd \left\{l_i(v_i) : P_i\right\}_{i \in I} \blacktriangleright \Delta,\ p : (\nu, \left\{\square_j\, l_j\, \langle T_j \rangle\, (\delta_j, \lambda_j).S_j\right\}_{j \in J})}\ [\texttt{Branch}]$$

$$\frac{\begin{array}{c}\Gamma; \theta \vdash p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i \in I} \blacktriangleright \Delta,\ p : (\nu, \left\{\texttt{C}_j\right\}_{j \in J}) \\ \Gamma; \theta + n \vdash Q \blacktriangleright \Delta + n,\ p : (\nu + n, \left\{\texttt{C}_j\right\}_{j \in J})\end{array}}{\Gamma; \theta \vdash p^{\diamond n} \rhd \left\{l_i(v_i) : P_i\right\}_{i \in I} \texttt{ after } Q \blacktriangleright \Delta,\ p : (\nu, \left\{\texttt{C}_j\right\}_{j \in J})}\ [\texttt{Timeout}] \tag{5.2.4}$$

$$\frac{\begin{array}{c}T \text{ base-type} \quad \Delta \text{ not } e\text{-reading} \quad \forall t : (\nu + t \models \delta) \implies (t \in e) \\ \forall t \in e : \Gamma, v : T;\ \theta + t \vdash P \blacktriangleright \Delta + t,\ p : (\nu + t\,[\lambda \mapsto 0]\,, S)\end{array}}{\Gamma; \theta \vdash p^e \rhd l(v) : P \blacktriangleright \Delta,\ p : (\nu, ?l\langle T \rangle\, (\delta, \{\lambda\})\, .S)}\ [\texttt{VRecv}]$$

$$\frac{\begin{array}{c}T = (\delta', S') \quad \nu' \models \delta' \quad \Delta \text{ not } e\text{-reading} \quad \forall t : (\nu + t \models \delta) \implies (t \in e) \\ \forall t \in e : \Gamma; \theta + t \vdash P \blacktriangleright \Delta + t,\ p : (\nu + t\,[\lambda \mapsto 0]\,, S),\ q : (\nu', S')\end{array}}{\Gamma; \theta \vdash p^e \rhd l(q) : P \blacktriangleright \Delta,\ p : (\nu, ?l\langle T \rangle\, (\delta, \{\lambda\})\, .S)}\ [\texttt{DRecv}] \tag{5.2.5}$$

$$\frac{T \text{ base-type} \quad \Gamma \vdash \texttt{v} : T \quad \Gamma; \theta \vdash qp : h \blacktriangleright \Delta,\ qp : \texttt{M}}{\Gamma; \theta \vdash qp : l\texttt{v} \cdot h \blacktriangleright \Delta,\ qp : l\,\langle T \rangle; \texttt{M}}\ [\texttt{VQue}]$$

$$\frac{T = (\delta, S) \quad \nu \models \delta \quad \Gamma; \theta \vdash qp : h \blacktriangleright \Delta,\ qp : \texttt{M}}{\Gamma; \theta \vdash qp : lq \cdot h \blacktriangleright \Delta,\ qp : l\,\langle T \rangle; \texttt{M},\ q : (\nu, S)}\ [\texttt{DQue}] \tag{5.2.6}$$

FIGURE C.5: (Restated) A Typing System for TOAST (communication)

C.3 Example Derivation: Ping-pong, Example 5.3.1

Recall Eqs. (5.3.1) and (5.3.2) from Example 5.3.1, below is the initial typing judgement:

$$
\begin{aligned}
&\varnothing;\ \theta \vdash\ p^{\leq 3} \triangleright \mathit{ping} : \\
&\qquad \texttt{set}\ \textcircled{z}.\texttt{delay}\ (t \leq 4)\,. \\
&\qquad \texttt{if}\ \big(\textcircled{z} \leq 3\big) \\
&\qquad\ \texttt{then}\ p \triangleleft \mathit{pong}.\mathtt{0} \\
&\qquad\ \texttt{else}\ p^{\infty} \triangleright \mathit{timeout} : \mathtt{0} \\
&\qquad \texttt{after}\ \texttt{delay}\ (0.1)\,. \\
&\qquad p \triangleleft \mathit{pong}. \\
&\qquad p^{\leq 3} \triangleright \mathit{ping} : \mathtt{0} \\
&\qquad \texttt{after}\ \texttt{delay}\ (0.1)\,. \\
&\qquad\ p \triangleleft \mathit{timeout}.\mathtt{0}
\end{aligned}
\quad \blacktriangleright \quad p : \left(v_0,\ \left\{ \begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}). \\ \quad \left\{ \begin{array}{l} !\mathit{pong}(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end} \end{array} \right\}, \\ !\mathit{pong}(x > 3,\ \{x\}). \\ \quad \left\{ \begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end} \end{array} \right\} \end{array} \right\} \right)
$$

The evaluation begins in Eq. (C.3.1) on the next page.

$$\dfrac{\dfrac{\dfrac{}{\varnothing;\ \theta_0 \vdash\ p^{\leq 3} \triangleright \mathit{ping} :\ \texttt{set}\ \textcircled{z}.\texttt{delay}\ (t \leq 4)\,.\ \texttt{if}\ (\textcircled{z} \leq 3)\ \texttt{then}\ p \triangleleft \mathit{pong}.\mathbf{0}\ \texttt{else}\ p^{\infty} \triangleright \mathit{timeout} : \mathbf{0} \blacktriangleright p : \left(\nu_0,\ \left\{\begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\left\{\begin{array}{l} !\mathit{pong}(x \leq 3,\ \{x\}).\texttt{end},\\ ?\mathit{timeout}(x > 3,\ \varnothing).\texttt{end}\end{array}\right\},\\ !\mathit{pong}(x > 3,\ \{x\}).\left\{\begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\texttt{end},\\ !\mathit{timeout}(x > 3,\ \varnothing).\texttt{end}\end{array}\right\}\end{array}\right\}\right)}\ \text{(See Eq. (C.3.2))}}{\varnothing;\ \theta_0 + 3 \vdash\ p \triangleleft \mathit{pong}.\ p^{\leq 3} \triangleright \mathit{ping} : \mathbf{0}\ \texttt{after}\ p \triangleleft \mathit{timeout}.\mathbf{0} \blacktriangleright p : \left(\nu_0 + 3,\ \left\{\begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\left\{\begin{array}{l} !\mathit{pong}(x \leq 3,\ \{x\}).\texttt{end},\\ ?\mathit{timeout}(x > 3,\ \varnothing).\texttt{end}\end{array}\right\},\\ !\mathit{pong}(x > 3,\ \{x\}).\left\{\begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\texttt{end},\\ !\mathit{timeout}(x > 3,\ \varnothing).\texttt{end}\end{array}\right\}\end{array}\right\}\right)}\ \text{(See Eq. (C.3.9))}\ [\texttt{VSend}]}{\varnothing;\ \theta_0 \vdash\ p^{\leq 3} \triangleright \mathit{ping} :\ \texttt{set}\ \textcircled{z}.\texttt{delay}\ (t \leq 4)\,.\ \texttt{if}\ (\textcircled{z} \leq 3)\ \texttt{then}\ p \triangleleft \mathit{pong}.\mathbf{0}\ \texttt{else}\ p^{\infty} \triangleright \mathit{timeout} : \mathbf{0}\ \texttt{after}\ \texttt{delay}\ (0.1)\,.\ p \triangleleft \mathit{pong}.\ p^{\leq 3} \triangleright \mathit{ping} : \mathbf{0}\ \texttt{after}\ \texttt{delay}\ (0.1)\,.p \triangleleft \mathit{timeout}.\mathbf{0} \blacktriangleright p : \left(\nu_0,\ \left\{\begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\left\{\begin{array}{l} !\mathit{pong}(x \leq 3,\ \{x\}).\texttt{end},\\ ?\mathit{timeout}(x > 3,\ \varnothing).\texttt{end}\end{array}\right\},\\ !\mathit{pong}(x > 3,\ \{x\}).\left\{\begin{array}{l} ?\mathit{ping}(x \leq 3,\ \{x\}).\texttt{end},\\ !\mathit{timeout}(x > 3,\ \varnothing).\texttt{end}\end{array}\right\}\end{array}\right\}\right)}\ [\texttt{Timeout}]\ [\texttt{Branch}]$$

(C.3.1)

$$\dfrac{\dfrac{\dfrac{}{(t=0)}\quad \text{(See Eq. (C.3.3))}}{\varnothing;\ \theta_0+0 \vdash \begin{array}{l}\texttt{set}\,\textcircled{z}.\texttt{delay}\,(t \le 4)\,.\\ \quad \texttt{if}\big(\textcircled{z} \le 3\big)\\ \quad\ \texttt{then}\ p \lhd pong.\mathbf{0}\\ \quad\ \texttt{else}\ p^{\infty} \rhd timeout : \mathbf{0}\end{array} \blacktriangleright p : \left(\begin{array}{l}(\nu_0[x \mapsto 0]) + 0,\\ \end{array}\ \begin{Bmatrix}!pong(x \le 3,\ \{x\}).\texttt{end},\\ ?timeout(x > 3,\ \varnothing).\texttt{end}\end{Bmatrix}\right)}\ [\texttt{Timer}]$$

$$\vdots$$

$$\dfrac{\dfrac{\dfrac{}{(t=3)}\quad \text{(See Eq. (C.3.6))}}{\varnothing;\ \theta_0+3 \vdash \begin{array}{l}\texttt{set}\,\textcircled{z}.\texttt{delay}\,(t \le 4)\,.\\ \quad \texttt{if}\big(\textcircled{z} \le 3\big)\\ \quad\ \texttt{then}\ p \lhd pong.\mathbf{0}\\ \quad\ \texttt{else}\ p^{\infty} \rhd timeout : \mathbf{0}\end{array} \blacktriangleright p : \left(\begin{array}{l}(\nu_0[x \mapsto 0]) + 3,\\ \end{array}\ \begin{Bmatrix}!pong(x \le 3,\ \{x\}).\texttt{end},\\ ?timeout(x > 3,\ \varnothing).\texttt{end}\end{Bmatrix}\right)}\ [\texttt{Timer}]$$

$$\dfrac{\dfrac{\dfrac{\forall t : \nu_0 + t \models (x \le 3) \iff t \in (\le 3)}{\varnothing;\ \theta_0 \vdash \begin{array}{l}p^{\le 3} \rhd ping :\\ \quad \texttt{set}\,\textcircled{z}.\texttt{delay}\,(t \le 4)\,.\\ \quad \texttt{if}\big(\textcircled{z} \le 3\big)\\ \quad\ \texttt{then}\ p \lhd pong.\mathbf{0}\\ \quad\ \texttt{else}\ p^{\infty} \rhd timeout : \mathbf{0}\end{array} \blacktriangleright p : \left(\nu_0, ?ping(x \le 3,\ \{x\}).\ \begin{Bmatrix}!pong(x \le 3,\ \{x\}).\texttt{end},\\ ?timeout(x > 3,\ \varnothing).\texttt{end}\end{Bmatrix}\right)}\ [\texttt{VRecv}]}{\neg(1 = 2 = 1)}$$

$$\varnothing;\ \theta_0 \vdash \begin{array}{l}p^{\le 3} \rhd ping :\\ \quad \texttt{set}\,\textcircled{z}.\texttt{delay}\,(t \le 4)\,.\\ \quad \texttt{if}\big(\textcircled{z} \le 3\big)\\ \quad\ \texttt{then}\ p \lhd pong.\mathbf{0}\\ \quad\ \texttt{else}\ p^{\infty} \rhd timeout : \mathbf{0}\end{array} \blacktriangleright p : \left(\nu_0, \left\{\begin{array}{l}?ping(x \le 3,\ \{x\}).\\ \quad \begin{Bmatrix}!pong(x \le 3,\ \{x\}).\texttt{end},\\ ?timeout(x > 3,\ \varnothing).\texttt{end}\end{Bmatrix},\\ !pong(x > 3,\ \{x\}).\\ \quad \begin{Bmatrix}?ping(x \le 3,\ \{x\}).\texttt{end},\\ !timeout(x > 3,\ \varnothing).\texttt{end}\end{Bmatrix}\end{array}\right\}\right) \quad [\texttt{Branch}]$$

(C.3.2)

$$\dfrac{\begin{array}{l|l} (t=0) & (\text{See Eq. (C.3.4)}) \\ \vdots & \\ (t=4) & (\text{See Eq. (C.3.5)}) \end{array}}{\begin{array}{l} \varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{delay}\,(t \leq 4)\,. \quad \blacktriangleright \quad p : \left(\nu_0[x \mapsto 0],\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right) \\ \qquad \texttt{if}\big(\textcircled{z} \leq 3\big) \\ \qquad \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \qquad \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \end{array}}\ [\texttt{Del-}d]$$

$$\dfrac{\begin{array}{l} \varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{set}\,\textcircled{z}.\texttt{delay}\,(t \leq 4)\,. \blacktriangleright \quad p : \left(\nu_0[x \mapsto 0],\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right) \\ \qquad \texttt{if}\big(\textcircled{z} \leq 3\big) \\ \qquad \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \qquad \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \end{array}}{\begin{array}{l} \varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{delay}\,(t \leq 4)\,. \quad \blacktriangleright \quad p : \left(\nu_0[x \mapsto 0],\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right) \\ \qquad \texttt{if}\big(\textcircled{z} \leq 3\big) \\ \qquad \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \qquad \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \end{array}}\ [\texttt{Timer}]$$

(C.3.3)

$$\dfrac{\varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{0} \blacktriangleright \varnothing \quad [\texttt{End}]}{\varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{0} \blacktriangleright \quad p : \Big(\nu_0[x \mapsto 0],\ \texttt{end}\Big)}\ [\texttt{Weak}]$$

$$\dfrac{\nu_0[x \mapsto 0] \models (x \leq 3)}{\varnothing;\ \theta_0', \textcircled{z} : 0 \vdash p \triangleleft pong.\texttt{0} \blacktriangleright \quad p : \left(\nu_0[x \mapsto 0],\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right)}\ [\texttt{VSend}]$$

$$\dfrac{\theta_0', \textcircled{z} : 0 \models (\textcircled{z} \leq 3)}{\begin{array}{l} \varnothing;\ (\theta_0', \textcircled{z} : 0) + 0 \vdash \texttt{if}\big(\textcircled{z} \leq 3\big) \quad \blacktriangleright \quad p : \left((\nu_0[x \mapsto 0]) + 0,\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right) \\ \qquad \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \qquad \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \end{array}}\ [\texttt{IfTrue}]$$

$$\dfrac{\begin{array}{l} \varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{delay}\,(0)\,. \quad \blacktriangleright \quad p : \left(\nu_0[x \mapsto 0],\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right) \\ \qquad \texttt{if}\big(\textcircled{z} \leq 3\big) \\ \qquad \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \qquad \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \end{array}}{\begin{array}{l} \varnothing;\ \theta_0', \textcircled{z} : 0 \vdash \texttt{delay}\,(t \leq 4)\,. \quad \blacktriangleright \quad p : \left(\nu_0[x \mapsto 0],\ \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\texttt{end}, \\ ?timeout(x > 3,\ \varnothing).\texttt{end} \end{array}\right\}\right) \\ \qquad \texttt{if}\big(\textcircled{z} \leq 3\big) \\ \qquad \texttt{then}\ p \triangleleft pong.\texttt{0} \\ \qquad \texttt{else}\ p^{\infty} \triangleright timeout : \texttt{0} \end{array}}\ [\texttt{Del-}t]$$

(C.3.4)

$$
\begin{array}{c|c}
(t' = 0) & \cfrac{\varnothing;\ \theta'_4, ⓩ : 4 \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\texttt{End}]}{\varnothing;\ (\theta'_4, ⓩ : 4) + 0 \vdash \mathtt{0} \blacktriangleright p : \Big((\nu_4[x \mapsto 4]) + 0, \mathtt{end} \Big)}\ [\texttt{Weak}] \\
\vdots & \\
(t' = \infty) & \cfrac{\varnothing;\ \theta'_\infty, ⓩ : \infty \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\texttt{End}]}{\varnothing;\ (\theta'_4, ⓩ : 4) + \infty \vdash \mathtt{0} \blacktriangleright p : \Big((\nu_4[x \mapsto 4]) + \infty, \mathtt{end} \Big)}\ [\texttt{Weak}]
\end{array}
$$

$$
\cfrac{\cfrac{\text{None base-type} \quad \forall t' : \nu_4[x \mapsto 4] + t' \models (x > 3) \Longleftrightarrow t' \in \infty}{\varnothing;\ \theta'_4, ⓩ : 4 \vdash p^\infty \rhd timeout : \mathtt{0} \blacktriangleright p : \Big(\nu_4[x \mapsto 4], ?timeout(x > 3,\ \varnothing).\mathtt{end} \Big)}\ [\texttt{VRecv}] \qquad \neg(1 = 2 = 1)}{\varnothing;\ \theta'_4, ⓩ : 4 \vdash p^\infty \rhd timeout : \mathtt{0} \blacktriangleright p : \left(\nu_4[x \mapsto 4], \begin{Bmatrix} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix} \right)}\ [\texttt{Branch}]
$$

$$
\cfrac{\cfrac{\theta'_4, ⓩ : 4 \not\models (ⓩ \leq 3)}{\begin{array}{l} \varnothing;\ (\theta'_0, ⓩ : 0) + 4 \vdash \mathtt{if}\,(ⓩ \leq 3) \\ \qquad \mathtt{then}\ p \lhd pong.\mathtt{0} \\ \qquad \mathtt{else}\ p^\infty \rhd timeout : \mathtt{0} \end{array} \blacktriangleright p : \left((\nu_0[x \mapsto 0]) + 4, \begin{Bmatrix} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix} \right)}\ [\texttt{IfFalse}]}{\begin{array}{l} \varnothing;\ \theta'_0, ⓩ : 0 \vdash \mathtt{delay}\,(4)\,. \\ \qquad \mathtt{if}\,(ⓩ \leq 3) \\ \qquad \mathtt{then}\ p \lhd pong.\mathtt{0} \\ \qquad \mathtt{else}\ p^\infty \rhd timeout : \mathtt{0} \end{array} \blacktriangleright p : \left(\nu_0[x \mapsto 0], \begin{Bmatrix} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix} \right)}\ [\texttt{Del-}t]
\tag{C.3.5}
$$

$$
\cfrac{\cfrac{\begin{array}{c|l} (t = 0) & (\text{See Eq. (C.3.7)}) \\ \vdots & \\ (t = 4) & (\text{See Eq. (C.3.8)}) \end{array}}{\begin{array}{l} \varnothing;\ \theta'_3, ⓩ : 0 \vdash \mathtt{delay}\,(t \leq 4)\,. \\ \qquad \mathtt{if}\,(ⓩ \leq 3) \\ \qquad \mathtt{then}\ p \lhd pong.\mathtt{0} \\ \qquad \mathtt{else}\ p^\infty \rhd timeout : \mathtt{0} \end{array} \blacktriangleright p : \left(\nu_3[x \mapsto 3], \begin{Bmatrix} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix} \right)}\ [\texttt{Del-}d]}{\begin{array}{l} \varnothing;\ \theta'_3, ⓩ : 3 \vdash \mathtt{set}\,ⓩ.\mathtt{delay}\,(t \leq 4)\,. \\ \qquad \mathtt{if}\,(ⓩ \leq 3) \\ \qquad \mathtt{then}\ p \lhd pong.\mathtt{0} \\ \qquad \mathtt{else}\ p^\infty \rhd timeout : \mathtt{0} \end{array} \blacktriangleright p : \left(\nu_3[x \mapsto 3], \begin{Bmatrix} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix} \right)}\ [\texttt{Timer}]
\tag{C.3.6}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing;\ \theta'_3, ⓩ : 3 \vdash \mathtt{0} \blacktriangleright \varnothing}\ [\mathtt{End}]}{\varnothing;\ \theta'_3, ⓩ : 3 \vdash \mathtt{0} \blacktriangleright p : \left(\nu_3[x \mapsto 0], \mathtt{end}\right)}\ [\mathtt{Weak}]}{\varnothing;\ \theta'_3, ⓩ : 3 \vdash p \lhd pong.\mathtt{0} \blacktriangleright p : \left(\nu_3[x \mapsto 3], \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{array}\right\}\right)}\ [\mathtt{VSend}]}{\varnothing;\ (\theta'_3, ⓩ : 3) + 0 \vdash \mathtt{if}\big(ⓩ \leq 3\big)\ \mathtt{then}\ p \lhd pong.\mathtt{0}\ \mathtt{else}\ p^{\infty} \rhd timeout : \mathtt{0} \blacktriangleright p : \left(\begin{array}{l} (\nu_3[x \mapsto 3]) + 0, \\ \end{array}\left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{array}\right\}\right)}\ [\mathtt{IfTrue}]}{\varnothing;\ \theta'_3, ⓩ : 3 \vdash \mathtt{delay}\,(0)\,.\ \mathtt{if}\big(ⓩ \leq 3\big)\ \mathtt{then}\ p \lhd pong.\mathtt{0}\ \mathtt{else}\ p^{\infty} \rhd timeout : \mathtt{0} \blacktriangleright p : \left(\nu_3[x \mapsto 3], \left\{\begin{array}{l} !pong(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?timeout(x > 3,\ \varnothing).\mathtt{end} \end{array}\right\}\right)}\ [\mathtt{Del}\text{-}t] \tag{C.3.7}$$

$$
\begin{array}{l|l}
(t' = 0) & \dfrac{\varnothing;\ \theta'_7, \textcircled{z} : 7 \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}]}{\varnothing;\ (\theta'_7, \textcircled{z} : 7) + 0 \vdash \mathtt{0} \blacktriangleright p : \left((\nu_7[x \mapsto 7]) + 0, \mathtt{end}\right)}\ [\mathtt{Weak}] \\
\vdots & \\
(t' = \infty) & \dfrac{\varnothing;\ \theta'_\infty, \textcircled{z} : \infty \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}]}{\varnothing;\ (\theta'_7, \textcircled{z} : 7) + \infty \vdash \mathtt{0} \blacktriangleright p : \left((\nu_7[x \mapsto 7]) + \infty, \mathtt{end}\right)}\ [\mathtt{Weak}]
\end{array}
$$

$$
\dfrac{\dfrac{\mathtt{None}\ \text{base-type} \quad \forall t' : \nu_7[x \mapsto 7] + t' \models (x > 3) \iff t' \in \infty}{\varnothing;\ \theta'_7, \textcircled{z} : 7 \vdash p^\infty \triangleright \mathit{timeout} : \mathtt{0} \blacktriangleright p : \left(\nu_7[x \mapsto 7], ?\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\right)}\ [\mathtt{VRecv}] \quad \neg(1 = 2 = 1)}{\varnothing;\ \theta'_7, \textcircled{z} : 7 \vdash p^\infty \triangleright \mathit{timeout} : \mathtt{0} \blacktriangleright p : \left(\nu_7[x \mapsto 7], \begin{Bmatrix} !\mathit{pong}(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix}\right)}\ [\mathtt{Branch}]
$$

$$
\dfrac{\dfrac{\theta'_7, \textcircled{z} : 7 \not\models (\textcircled{z} \leq 3)}{\varnothing;\ (\theta'_3, \textcircled{z} : 3) + 4 \vdash \mathtt{if}(\textcircled{z} \leq 3) \blacktriangleright p : \left((\nu_3[x \mapsto 3]) + 4, \begin{Bmatrix} !\mathit{pong}(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix}\right)\ \mathtt{then}\ p \lhd \mathit{pong}.\mathtt{0}\ \mathtt{else}\ p^\infty \triangleright \mathit{timeout} : \mathtt{0}}\ [\mathtt{IfFalse}]}{\varnothing;\ \theta'_3, \textcircled{z} : 3 \vdash \mathtt{delay}\,(4).\ \mathtt{if}(\textcircled{z} \leq 3) \blacktriangleright p : \left(\nu_3[x \mapsto 3], \begin{Bmatrix} !\mathit{pong}(x \leq 3,\ \{x\}).\mathtt{end}, \\ ?\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end} \end{Bmatrix}\right)\ \mathtt{then}\ p \lhd \mathit{pong}.\mathtt{0}\ \mathtt{else}\ p^\infty \triangleright \mathit{timeout} : \mathtt{0}}\ [\mathtt{Del}\text{-}t] \tag{C.3.8}
$$

$$
\dfrac{
\dfrac{
\begin{array}{l}(t=0)\\ \vdots \\ (t=3)\end{array}
\quad
\begin{array}{c}
\dfrac{\varnothing;\ \theta_{3.1} \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}]}{\varnothing;\ \theta_{3.1} \vdash \mathtt{0} \blacktriangleright\ p : \left((\nu_{3.1}[x \mapsto 0]) + 0, \mathtt{end}\right)}\ [\mathtt{Weak}]
\\ \\
\dfrac{\varnothing;\ \theta_{3.1} \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}]}{\varnothing;\ \theta_{3.1} \vdash \mathtt{0} \blacktriangleright\ p : \left((\nu_{3.1}[x \mapsto 0]) + 3[x \mapsto 0], \mathtt{end}\right)}\ [\mathtt{Weak}]
\end{array}
}{\varnothing;\ \theta_{3.1} \vdash\ p^{\leq 3} \triangleright \mathit{ping} : \mathtt{0} \blacktriangleright\ p : \left(\nu_{3.1}[x \mapsto 0], ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}\right)}\ [\mathtt{VRecv}]
}{\varnothing;\ \theta_{3.1} \vdash\ p^{\leq 3} \triangleright \mathit{ping} : \mathtt{0} \blacktriangleright\ p : \left(\nu_{3.1}[x \mapsto 0], \begin{Bmatrix} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\end{Bmatrix}\right)}\ [\mathtt{Branch}]
$$

$$
\dfrac{
\dfrac{\varnothing;\ \theta_{6.2} \vdash \mathtt{0} \blacktriangleright \varnothing \quad [\mathtt{End}]}{\varnothing;\ \theta_{6.2} \vdash \mathtt{0} \blacktriangleright\ p : \left(\nu_{6.2}[x \mapsto 3.1], \mathtt{end}\right)}\ [\mathtt{Weak}]
}{\varnothing;\ \theta_{6.1} + 0.1 \vdash\ p \lhd \mathit{timeout}.\mathtt{0} \blacktriangleright\ p : \left((\nu_{6.1}[x \mapsto 3]) + 0.1, \begin{Bmatrix} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\end{Bmatrix}\right)}\ [\mathtt{VSend}]
$$

$$
\dfrac{
\dfrac{
\nu_{6.2}[x \mapsto 3.1] \models (x > 3)
}{
\begin{array}{l}\varnothing;\ \theta_{3.1} + 3 \vdash \mathtt{delay}\,(0.1)\,. \\ \qquad\quad p \lhd \mathit{timeout}.\mathtt{0}\end{array}
\ \blacktriangleright\ p : \left((\nu_{3.1}[x \mapsto 0]) + 3, \begin{Bmatrix} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\end{Bmatrix}\right)
}\ [\mathtt{Del}\text{-}t]
}{
\begin{array}{l}\varnothing;\ \theta_{3.1} \vdash\ p^{\leq 3} \triangleright \mathit{ping} : \mathtt{0} \\ \qquad \mathtt{after}\ \mathtt{delay}\,(0.1)\,. \\ \qquad p \lhd \mathit{timeout}.\mathtt{0}\end{array}
\ \blacktriangleright\ p : \left(\nu_{3.1}[x \mapsto 0], \begin{Bmatrix} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\end{Bmatrix}\right)
}\ [\mathtt{Timeout}]
$$

$$
\dfrac{
\begin{array}{l}\varnothing;\ \theta_3 + 0.1 \vdash\ p \lhd \mathit{pong}. \\ \qquad\quad p^{\leq 3} \triangleright \mathit{ping} : \mathtt{0} \\ \qquad\quad \mathtt{after}\ \mathtt{delay}\,(0.1)\,.p \lhd \mathit{timeout}.\mathtt{0}\end{array}
\ \blacktriangleright\ p : \left(\nu_3 + 0.1, !\mathit{pong}(x > 3,\ \{x\}).\begin{Bmatrix} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\end{Bmatrix}\right)
}{
\nu_{3.1} \models (x > 3)
}\ [\mathtt{Del}\text{-}t]
$$

$$
\begin{array}{l}\varnothing;\ \theta_3 \vdash \mathtt{delay}\,(0.1)\,. \\ \qquad p \lhd \mathit{pong}. \\ \qquad p^{\leq 3} \triangleright \mathit{ping} : \mathtt{0} \\ \qquad \mathtt{after}\ \mathtt{delay}\,(0.1)\,.p \lhd \mathit{timeout}.\mathtt{0}\end{array}
\ \blacktriangleright\ p : \left(\nu_3, !\mathit{pong}(x > 3,\ \{x\}).\begin{Bmatrix} ?\mathit{ping}(x \leq 3,\ \{x\}).\mathtt{end}, \\ !\mathit{timeout}(x > 3,\ \varnothing).\mathtt{end}\end{Bmatrix}\right)
\tag{C.3.9}
$$

Acronyms

2FA **T**wo **F**actor **A**uthentication (Protocol).

AMPST **A**ffine **M**ulti**p**arty **S**ession **T**ypes, (Lagaillardie et al. 2022).

API **A**pplication **P**rogramming **I**nterface.

ATMP **A**ffine **T**imed **M**ulti**p**arty Session Types, (Hou et al. 2024).

ATST **A**synchronous **T**imed **S**ession **T**ypes, (Bocchi et al. 2019).

CAPI **C**onversation **A**pplication **P**rogramming **I**nterface, (Hu et al. 2013).

CCS **C**alculus of **C**ommunicating **S**ystems, (Milner 1982).

CFSM **C**ommunicating **F**inite **S**tate **M**achines, (Brand et al. 1983).

CTA **C**ommunicating **T**imed **A**utomata, (Krcál et al. 2006).

DBM **D**ifference **B**ound **M**atricies, (Bengtsson et al. 2003).

DSL **D**omain **S**pecific **L**anguage

FIFO **F**irst **i**n **F**irst **o**ut

FSM **F**inite **S**tate **M**achines

FTMPST **F**ault **T**olerant **M**ulti**p**arty **S**ession **T**ypes, (Peters et al. 2023).

IoT **I**nternet **o**f **T**hings.

LTS **L**abeled **T**ransition **S**ystem.

MAGπ **M**ultiparty **A**synchronous **G**eneralised π-calculus, (Brun et al. 2023).

MPST **M**ulti**p**arty **S**ession **T**ypes, (Honda et al. 2008, 2016).

MSA **M**ultiparty **S**ession **A**ctors, (Neykova et al. 2014).

PID **P**rocess **ID**.

RBST **R**ate **B**ased **S**ession **T**ypes, (Iraci et al. 2023).

SMT **S**atisfiability **M**odulo **T**heory

SMTP **S**imple **M**ail **T**ransfer **P**rotocol, (Klensin 2008).

STP **S**imple **T**heorem **P**rover.

TA **T**imed **A**utomata, (Alur et al. 1994).

TMPST **T**imed **M**ulti**p**arty **S**ession **T**ypes, (Bocchi et al. 2014).

TOAST **T**ime**o**ut **A**synchronous **S**ession **T**ypes, (Pears et al. 2024c).

TOP **T**ime-**O**ut **P**rotocol, (Esfarjani et al. 1998).

TPL **T**emporal **P**rocess **L**anguage, (Bocchi et al. 2022)

TST **T**imed **S**ession **T**ypes, (Bartoletti et al. 2017).

Bibliography

Aceto, Luca, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir (2024). "A monitoring tool for linear-time μHML". In: *SCP* 232, p. 103031. ISSN: 0167-6423. DOI: 10.1016/j.scico.2023.103031.

Alur, Rajeev and David L. Dill (1994). "A Theory of Timed Automata". In: *TCS* 126 (2), pp. 183–235. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8.

Bartocci, Ezio, Yliès Falcone, Adrian Francalanza, and Giles Reger (2018). "Introduction to Runtime Verification". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer, pp. 1–33. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1.

Bartoletti, Massimo, Laura Bocchi, and Maurizio Murgia (2018). "Progress-Preserving Refinements of CTA". In: *CONCUR*. Vol. 118. LIPIcs. Schloss Dagstuhl, 40:1–40:19. ISBN: 978-3-95977-087-3. DOI: 10.4230/LIPIcs.CONCUR.2018.40.

Bartoletti, Massimo, Tiziana Cimoli, and Maurizio Murgia (2017). "Timed Session Types". In: *Logical Methods in Computer Science* 13.4. DOI: 10.23638/LMCS-13(4:25)2017.

Bengtsson, Johan and Wang Yi (2003). "Timed Automata: Semantics, Algorithms and Tools". In: *Lectures on Concurrency and Petri Nets*. Vol. 3098. LNCS. Springer, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3.

Berger, Martin and Nobuko Yoshida (2007). "Timed, Distributed, Probabilistic, Typed Processes". In: *APLAS*. Vol. 4807. LNCS. Springer, pp. 158–174. DOI: 10.1007/978-3-540-76637-7_11.

Bernardi, Giovanni and Matthew Hennessy (2016). "Using higher-order contracts to model session types". In: *Logical Methods in Computer Science* 12 (2). DOI: 10.2168/LMCS-12(2:10)2016.

Bettini, Lorenzo, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida (2008). "Global Progress in Dynamically Interleaved Multiparty Sessions". In: *CONCUR*. Vol. 5201. LNCS. Springer, pp. 418–433. DOI: 10.1007/978-3-540-85361-9_33.

Bocchi, Laura, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida (2013). "Monitoring Networks through Multiparty Session Types". In: *FTDS*. Springer, pp. 50–65. ISBN: 978-3-642-38592-6. DOI: 10.1007/978-3-642-38592-6_5.

Bocchi, Laura, Julien Lange, Simon Thompson, and A. Laura Voinea (2022). "A Model of Actors and Grey Failures". In: *COORDINATION*. Vol. 13271. LNCS. Springer, pp. 140–158. DOI: 10.1007/978-3-031-08143-9_9.

Bocchi, Laura, Julien Lange, and Nobuko Yoshida (2015). "Meeting Deadlines Together". In: *CONCUR*. Vol. 42. LIPIcs, pp. 283–296. ISBN: 978-3-939897-91-0. DOI: 10.4230/LIPIcs.CONCUR.2015.283.

Bocchi, Laura, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida (2019). "Asynchronous timed session types: from duality to time-sensitive processes". In: *ESOP*. Vol. 11423. LNCS. Springer, pp. 583–610. ISBN: 9783030171834. DOI: 10.1007/978-3-030-17184-1_21.

Bocchi, Laura, Dominic Orchard, and A. Laura Voinea (2023). "A Theory of Composing Protocols". In: *The Art, Science, and Engineering of Programming* 7.2. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2023/7/6.

Bocchi, Laura, Weizhen Yang, and Nobuko Yoshida (2014). "Timed Multiparty Session Types". In: *CONCUR*. Vol. 8704. LNCS. Springer, pp. 419–434. DOI: 10.1007/978-3-662-44584-6_29.

Bono, Viviana and Luca Padovani (2012). "Typing Copyless Message Passing". In: *Logical Methods in Computer Science* 8 (1). DOI: 10.2168/LMCS-8(1:17)2012.

Brand, Daniel and Pitro Zafiropulo (1983). "On Communicating Finite-State Machines". In: *Journal of the ACM* 30.2, pp. 323–342. DOI: 10.1145/322374.322380.

Brun, Matthew Alan Le and Ornela Dardha (2023). "MAGπ: Types for Failure-Prone Communication". In: *Programming Languages and Systems*. Vol. 13990. LNCS. Springer, pp. 363–391. ISBN: 978-3-031-30044-8. DOI: 10.1007/978-3-031-30044-8_14.

Burlò, Christian Bartolo, Adrian Francalanza, and Alceste Scalas (2021). "On the Monitorability of Session Types, in Theory and Practice". In: *ECOOP*. LIPIcs. Schloss Dagstuhl, 20:1–20:30. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.20.

Burlò, Christian Bartolo, Adrian Francalanza, Alceste Scalas, Catia Trubiani, and Emilio Tuosto (2022). "PSTMonitor: Monitor synthesis from probabilistic session types". In: *SCP* 222, p. 102847. ISSN: 0167-6423. DOI: 10.1016/j.scico.2022.102847.

Carbone, Marco, Ornela Dardha, and Montesi Fabrizio (2014). "Progress as Compositional Lock-Freedom". In: *Coordination Models and Languages*. Springer, pp. 49–64. ISBN: 978-3-662-43376-8. DOI: 10.1007/978-3-662-43376-8_4.

Carbone, Marco, Kohei Honda, and Nobuko Yoshida (2008). "Structured Interactional Exceptions in Session Types". In: *CONCUR*. Vol. 5201. LNCS. Springer Berlin Heidelberg, pp. 402–417. DOI: 10.1007/978-3-540-85361-9_32.

Casal, Filipe, Andreia Mordido, and Vasco T. Vasconcelos (2020). "Mixed Sessions: the Other Side of the Tape". In: *EPTCS* 314, pp. 46–60. ISSN: 2075-2180. DOI: 10.4204/eptcs.314.5.

Cassar, Ian, Adrian Francalanza, Duncan Attard, Luca Aceto, and Anna Ingólfsdóttir (2017). "A Suite of Monitoring Tools for Erlang". In: *RV-CuBES*. Vol. 3. EasyChair, pp. 41–47. DOI: 10.29007/7lrd.

Castro-Perez, David, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida (2021). "Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes". In: *PLDI*. LIPIcs. ACM, pp. 237–251. DOI: 10.1145/3453483.3454041.

Castro-Perez, David, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida (2019). "Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures". In: *POPL*. Vol. 3. ACM, pp. 1–30. DOI: 10.1145/3290342.

Castro-Perez, David and Nobuko Yoshida (Nov. 2020). "CAMP: cost-aware multiparty session protocols". In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428223.

Castro-Perez, David and Nobuko Yoshida (2023). "Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols". In: *ECOOP*. Vol. 263. Schloss Dagstuhl, 6:1–6:30. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.6.

Cesarini, Francesco and Steve Vinoski (2016). *Designing for Scalability with Erlang/OTP*. O'Reilly Media, Inc. ISBN: 9781449361563.

Chen, Tzu-chun, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida (2017). "On the Preciseness of Subtyping in Session Types". In: *Logical Methods in Computer Science* 13.2. DOI: 10.23638/LMCS-13(2:12)2017.

Coppo, Mario, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani (2016). "Global progress for dynamically interleaved multiparty sessions". In: *Mathematical Structures in Computer Science* 26.2, pp. 238–302. DOI: 10.1017/S0960129514000188.

Dardha, Ornela, Elena Giachino, and Davide Sangiorgi (2017). "Session types revisited". In: *Information and Computation* 256, pp. 253–286. ISSN: 0890-5401. DOI: 10.1016/j.ic.2017.06.002.

Demangeon, Romain, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida (2015). "Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python". In: *FMSD* 46.3, pp. 197–225. DOI: 10.1007/S10703-014-0218-8.

Deniélou, Pierre-Malo and Nobuko Yoshida (2012). "Multiparty Session Types Meet Communicating Automata". In: *ESOP*. Vol. 7966. LNCS. Springer, pp. 194–213. ISBN: 978-3-642-28869-2. DOI: 10.1007/978-3-642-28869-2_10.

Deniélou, Pierre-Malo and Nobuko Yoshida (2013). "Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types". In: *ICALP*. Vol. 7966. LNCS. Springer, pp. 174–186. DOI: 10.1007/978-3-642-39212-2_18.

Dezani-Ciancaglini, Mariangiola, Ugo de'Liguoro, and Nobuko Yoshida (2007). "On Progress for Structured Communications". In: *TGC*. Vol. 4912. LNCS. Springer, pp. 257–275. DOI: 10.1007/978-3-540-78663-4_18.

Efthivoulidis, G., E. Verentziotis, A. Meliones, T. Varvarigou, A. Kontizas, and G. Deconinck (1999). "Fault Tolerant Communication in Embedded Parallel High

Performance Computing". In: *Parallel Computational Fluid Dynamics 1998*, pp. 405–414. ISBN: 978-0-444-82850-7. DOI: 10.1016/B978-044482850-7/50110-8.

Erlang on TOAST (2024). URL: https://github.com/jonahpears/Erlang-on-TOAST.

Esfarjani, K. and S. Y. Nof (1998). "Client-server model of integrated production facilities". In: *International Journal of Production Research* 36.12, pp. 3295–3321. DOI: 10.1080/002075498192076.

Fowler, Simon (2016). "An Erlang Implementation of Multiparty Session Actors". In: *EPTCS* 223, pp. 36–50. ISSN: 2075-2180. DOI: 10.4204/eptcs.223.3.

Francalanza, Adrian, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir (2017). "A Foundation for Runtime Monitoring". In: *RV*. Springer, pp. 8–29. ISBN: 978-3-319-67531-2. DOI: 10.1007/978-3-319-67531-2_2.

Francalanza, Adrian, Jorge A. Pérez, and César Sánchez (2018). "Runtime Verification for Decentralised and Distributed Systems". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer, pp. 176–210. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_6.

Gay, Simon J. and Malcolm Hole (2005). "Subtyping for session types in the pi calculus". In: *Acta Informatica* 42.2-3, pp. 191–225. DOI: 10.1007/S00236-005-0177-Z.

Google (2024). *The Go Programming Language*. https://go.dev/.

Gouda, M.G., E.G. Manning, and Y.T. Yu (1984). "On the progress of communication between two finite state machines". In: *Information and Control* 63.3, pp. 200–216. DOI: 10.1016/S0019-9958(84)80014-5.

Hennessy, Matthew and Tim Regan (1995). "A Process Algebra for Timed Systems". In: *Information and Computation* 117.2, pp. 221–239. ISSN: 0890-5401. DOI: 10.1006/inco.1995.1041.

Honda, Kohei, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida (2011). "Scribbling Interactions with a Formal Foundation". In: *Distributed Computing and Internet Technology*. Springer, pp. 55–75. ISBN: 978-3-642-19056-8. DOI: 10.1007/978-3-642-19056-8_4.

Honda, Kohei, Vasco Thudichum Vasconcelos, and Makoto Kubo (1998). "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *ESOP*. Vol. 1381. LNCS. Springer, pp. 122–138. DOI: 10.1007/BFB0053567.

Honda, Kohei, Nobuko Yoshida, and Marco Carbone (2008). "Multiparty asynchronous session types". In: *POPL*. ACM, pp. 273–284. DOI: 10.1145/1328438.1328472.

Honda, Kohei, Nobuko Yoshida, and Marco Carbone (2016). "Multiparty Asynchronous Session Types". In: *Journal of the ACM* 63.1. ISSN: 0004-5411. DOI: 10.1145/2827695.

Hou, Ping, Nicolas Lagaillardie, and Nobuko Yoshida (2024). *Fearless Asynchronous Communications with Timed Multiparty Session Protocols*. arXiv: 2406.19541 [cs.PL].

Hu, Raymond, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda (2010). "Type-Safe Eventful Sessions in Java". In: *ECOOP*. Springer, pp. 329–353. ISBN: 978-3-642-14107-2. DOI: 10.1007/978-3-642-14107-2_16.

Hu, Raymond, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda (2013). "Practical Interruptible Conversations". In: *RV*. Springer, pp. 130–148. ISBN: 978-3-642-40787-1. DOI: 10.1007/978-3-642-40787-1_8.

Hu, Raymond and Nobuko Yoshida (2016). "Hybrid Session Verification Through Endpoint API Generation". In: *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 401–418. ISBN: 978-3-662-49665-7. DOI: 10.1007/978-3-662-49665-7_24.

Hu, Raymond, Nobuko Yoshida, and Kohei Honda (2008). "Session-Based Distributed Programming in Java". In: *ECOOP*. Vol. 5142. LNCS. Springer, pp. 516–541. DOI: 10.1007/978-3-540-70592-5_22.

Iraci, Grant, Cheng-En Chuan, Raymond Hu, and Lukasz Ziarek (2023). "Validating IoT Devices with Rate-Based Session Types". In: *Proc. ACM Program. Lang.* 7.OOPSLA, pp. 1589–1617. DOI: 10.1145/3622854.

Klensin, John (2008). *SMTP, Request for Comments: 5321*. Section 4.5.3.2.7: Server Timeout. URL: https://datatracker.ietf.org/doc/html/rfc5321.

Kobayashi, Naoki (2002). "A Type System for Lock-Free Processes". In: *Information and Computation* 177.2, pp. 122–159. ISSN: 0890-5401. DOI: 10.1006/inco.2002.3171.

Kobayashi, Naoki and Davide Sangiorgi (2010). "A hybrid type system for lock-freedom of mobile processes". In: *ACM TOPLAS* 32.5. ISSN: 0164-0925. DOI: 10.1145/1745312.1745313.

Kokke, Wen (Sept. 2019). "Rusty Variation: Deadlock-free Sessions with Failure in Rust". In: *EPTCS* 304, pp. 48–60. ISSN: 2075-2180. DOI: 10.4204/eptcs.304.4.

Krcál, Pavel and Wang Yi (2006). "Communicating Timed Automata: The More Synchronous, the More Difficult to Verify". In: *CAV*. Vol. 4144. LNCS. Springer, pp. 249–262. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_24.

Kuhn, Roland, Hernán Melgratti, and Emilio Tuosto (2023). "Behavioural Types for Local-First Software". In: *ECOOP*. Vol. 263. LIPIcs. Schloss Dagstuhl, 15:1–15:28. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.15.

Lagaillardie, Nicolas, Rumyana Neykova, and Nobuko Yoshida (2020). "Implementing Multiparty Session Types in Rust". In: *COORDINATION*. Springer, pp. 127–136. DOI: 10.1007/978-3-030-50029-0_8.

Lagaillardie, Nicolas, Rumyana Neykova, and Nobuko Yoshida (2022). "Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types". In: *ECOOP*. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. DOI: 10.48550/arXiv.2204.13464.

Lindley, Sam and J. Garrett Morris (2016). "Talking bananas: structural recursion for session types". In: *ICFP*. ACM, pp. 434–447. ISBN: 9781450342193. DOI: 10.1145/2951913.2951921.

Milner, Robin (1982). *A Calculus of Communicating Systems*. Springer. ISBN: 0387102353. DOI: 10.5555/539036.

Milner, Robin (1999). *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press. ISBN: 978-0-521-65869-0.

Mostrous, Dimitris and Vasco T. Vasconcelos (2011). "Session Typing for a Featherweight Erlang". In: *COORDINATION*. Springer, pp. 95–109. ISBN: 978-3-642-21464-6. DOI: 10.1007/978-3-642-21464-6_7.

Mostrous, Dimitris and Vasco T. Vasconcelos (2018). "Affine Sessions". In: *Logical Methods in Computer Science* 14.4. DOI: 10.23638/LMCS-14(4:14)2018.

Neykova, Rumyana (2013). "Session Types Go Dynamic or How to Verify Your Python Conversations". In: *EPTCS* 137, pp. 95–102. DOI: 10.4204/EPTCS.137.8.

Neykova, Rumyana, Laura Bocchi, and Nobuko Yoshida (2017a). "Timed runtime monitoring for multiparty conversations". In: *FAC* 29.5, pp. 877–910. ISSN: 0934-5043. DOI: 10.1007/s00165-017-0420-8.

Neykova, Rumyana, Raymond Hu, Noboku Yoshida, and Fahd Abdeljallal (2018). "A session type provider: compile-time API generation of distributed protocols with refinements in F#". In: *CC*. ACM, pp. 128–138. DOI: 10.1145/3178372.3179495.

Neykova, Rumyana and Nobuko Yoshida (2014). "Multiparty Session Actors". In: *COORDINATION*. Vol. 8459. LNCS. Springer, pp. 131–146. DOI: 10.1007/978-3-662-43376-8_9.

Neykova, Rumyana and Nobuko Yoshida (2017b). "Let it recover: multiparty protocol-induced recovery". In: *CC*. ACM, pp. 98–108. ISBN: 9781450352338. DOI: 10.1145/3033019.3033031.

Neykova, Rumyana and Nobuko Yoshida (2019). "Featherweight Scribble". In: *Models, Languages, and Tools for Concurrent and Distributed Programming*. Springer, pp. 236–259. ISBN: 978-3-030-21485-2. DOI: 10.1007/978-3-030-21485-2_14.

Neykova, Rumyana, Nobuko Yoshida, and Raymond Hu (2013). "SPY: Local Verification of Global Protocols". In: *RV*. Vol. 8174. LNCS. Springer, pp. 358–363. ISBN: 978-3-642-40787-1. DOI: 10.1007/978-3-642-40787-1_25.

Padovani, Luca (2014). "Deadlock and lock freedom in the linear π-calculus". In: ACM. ISBN: 9781450328869. DOI: 10.1145/2603088.2603116.

Pears, Jonah, Laura Bocchi, and Raymond Hu (2024a). "Erlang on TOAST: Generating Erlang Stubs with Inline TOAST Monitors". In: *Erlang Workshop*. ACM, pp. 33–44. ISBN: 9798400710988. DOI: 10.1145/3677995.3678192.

Pears, Jonah, Laura Bocchi, and Andy King (2023). "Safe asynchronous mixed-choice for timed interactions". In: *COORDINATION*. Vol. 13908. LNCS. Springer, pp. 214–231. DOI: 10.1007/978-3-031-35361-1_12.

Pears, Jonah, Laura Bocchi, Maurizio Murgia, and Andy King (2024b). *Introducing TOAST: Safe Asynchronous Mixed-Choice For Timed Interactions*. arXiv: 2401.11197 [cs.LO].

Pears, Jonah, Laura Bocchi, Maurizio Murgia, and Andy King (2024c). *Timeout Asynchronous Session Types: Safe Asynchronous Mixed-Choice For Timed Interactions*. arXiv: 2401.11197v2 [cs.LO].

Peralta, J., P. Anussornnitisarn, and S. Y. Nof (2003). "Analysis of a time-out protocol and its applications in a single server environment". In: *IJCIM* 16.1, pp. 1–13. DOI: 10.1080/713804980.

Peters, Kirstin, Uwe Nestmann, and Christoph Wagner (2022). "Fault-Tolerant Multiparty Session Types". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Springer, pp. 93–113. ISBN: 978-3-031-08679-3. DOI: 10.1007/978-3-031-08679-3_7.

Peters, Kirstin, Uwe Nestmann, and Christoph Wagner (2023). "FTMPST: Fault-Tolerant Multiparty Session Types". In: vol. 19. DOI: 10.46298/lmcs-19(4:14)2023.

Peters, Kirstin and Nobuko Yoshida (2024). "Mixed choice in session types". In: *Information and Control* 298, p. 105164. ISSN: 0890-5401. DOI: 10.1016/j.ic.2024.105164.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press, p. 645. ISBN: 9780262162098.

Protocol Reengineering Implementation (2023). URL: https://github.com/LauraVoinea/protocol-reengineering-implementation.

Scalas, Alceste and Nobuko Yoshida (2016). "Lightweight Session Programming in Scala". In: *ECOOP*. Vol. 56. LIPIcs. Schloss Dagstuhl, 21:1–21:28. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.21.

Scalas, Alceste and Nobuko Yoshida (2019). "Less is more: multiparty session types revisited". In: *ACM* 3.POPL. DOI: 10.1145/3290343.

Takeuchi, Kaku, Kohei Honda, and Makoto Kubo (1994). "An interaction-based langauge and its typing system". In: *PARLE*. Springer, pp. 398–413. ISBN: 978-3-540-48477-6. DOI: 10.1007/3-540-58184-7_118.

The Coq Proof Assistant (2024). URL: https://coq.inria.fr/.

Vasconcelos, Vasco T. (2012). "Fundamentals of session types". In: *Information and Control* 217, pp. 52–70. DOI: 10.1016/J.IC.2012.05.002.

Vasconcelos, Vasco T., Filipe Casal, Bernardo Almeida, and Andreia Mordido (2020). "Mixed Sessions". In: *ESOP*. Vol. 12075. LNCS. Springer, pp. 715–742. DOI: 10.1007/978-3-030-44914-8_26.

Viering, Malte, Raymond Hu, Patrick Eugster, and Lukasz Ziarek (2021). "A multiparty session typing discipline for fault-tolerant event-driven distributed programming". In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: 10.1145/3485501.

Yoshida, Nobuko, Raymond Hu, Rumyana Neykova, and Nicholas Ng (2014). "The Scribble Protocol Language". In: *TGC*. Springer, pp. 22–41. ISBN: 978-3-319-05119-2. DOI: 10.1007/978-3-319-05119-2_3.

Yoshida, Nobuko and Vasco T. Vasconcelos (2007). "Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication". In: *Electronic Notes in Theoretical Computer Science* 171.4, pp. 73–93. DOI: 10.1016/j.entcs.2007.02.056.

Yoshida, Nobuko, Fangyi Zhou, and Francisco Ferreira (2021). "Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types". In: *Fundamentals of Computation Theory*. Vol. 12867. LNCS. Springer, pp. 18–35. DOI: 10.1007/978-3-030-86593-1_2.

Z3 Theorem Prover (2024). URL: https://github.com/Z3Prover/z3.