Imperial College London
Department of Computing

Timed and Choreographical Multiparty Session Types Theory

Weizhen Yang

May, 2017

Supervised by Prof. Nobuko Yoshida

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration of Originality

I, being a candidate for PhD in Computer Science, hereby declare that this report and the work described in it are my own work.

Weizhen Yang

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

Multiparty session types (MPSTs) are the formalism for describing protocols among multiple participants from a global point of view. They are used to check type-safety and deadlock freedom for programs. In real-time distributed systems, however, protocol descriptions often include: time-related specifications, such as deadline and timeout, by imposing time restrictions on interactions to guarantee the quality of service; and parallel computation for improving efficiency. The original theory of multiparty session types lacks the ability for expressing time and parallel actions in the global description. Therefore, we propose the theory of timed choreographical session types to solve the problem.

To express time-related specifications, we associate time constraints and clock resets to asynchronous message passing in global types and local types. We define the labelled transition systems of timed global and local types, and prove that soundness and completeness are preserved through projection. We add a delay primitive to session calculus to achieve a minimum extension of asynchronous π-calculus and prove theorems of type preservation and subject reduction. We also formalise the theorem of time error-freedom and properties of feasibility and wait-freedom to guarantee that no deadlock is caused by time error. Additionally, we show that timed global session types enjoy progress property if the global types without time constraints and resets are verified to progress.

In the theory of choreographical session types, for the purpose of precise thread management, explicit fork/merge operations for parallel composition and choice/join for branching are defined in both global and local session types and session calculus. To ensure well-behaved communications, we formalise the definition of well-formedness for global types and present sufficient conditions for verifying well-formedness. Based on choreographical session calculus, we define the operational semantics and typing system for processes and prove the theorems of subject reduction and type safety.

When we model scenarios of concurrent real-time systems, it is natural to combine the formalisms of timed session types and choreographical session

types. Then we add time assertions to labelled message passing operations in choreographical global and local types, and delay primitive in session calculus. Due to the orthogonality of time and parallel composition, we easily prove the theorems of subject reduction, and time error-freedom within the timed choreographical context. Due to introduced complexity in global types, we present the progress enablement and time progress property to guarantee that processes will progress.

Acknowledgements

The person I would like to acknowledge most is my supervisor Prof. Nobuko Yoshida. During my four years of PhD study, her guidance and support help me to overcome various difficulties and finally fulfil the goal of this Phd study. Nobuko, thank you very much for your patience, kindness and professional and personal guidance.

Furthermore, I would like to thank my second supervisor Dr. Iain Phillips for the time he spent and the advice he gave on my PhD thesis.

I thank my examiners Prof. Michael Huth and Prof. Maribel Fernandez as well. They gave constructive and detailed suggestions that improved the readability and quality of the final thesis.

I also appreciate Dr. Laura Bocchi's contribution to the theory of timed session types, Dr. Julien Lange's advice on formalising the theory of timed choreographical session types, and Dr. Nicholas Ng's kind help. Additionally, I thank all colleagues from Mobility Reading Group for your accompaniment.

I am grateful to Department of Computing of Imperial College London for allowing me to study as a doctoral student and get academic training from the Department.

Last but not least, I thank my family, especially my parents and my husband. I could hardly complete the study without your support and love.

Contents

1. Introduction **13**

1.1. Motivation 13

1.2. Goal and Objectives 16

1.3. Contributions 16

1.3.1. Thesis Outline 17

1.3.2. Publications and Talks 18

2. Background **19**

2.1. π-Calculus 19

2.2. Session Types 20

2.2.1. Binary Session Types 21

2.2.2. Multiparty Session Types 24

2.3. Timed Automata and Other Graphical Models 28

2.3.1. Timed Automata 28

2.3.2. Petri Nets 30

2.3.3. Harel's Statecharts 31

2.3.4. Business Process Modeling and Notation (BPMN) 36

3. Timed Multiparty Session Types **39**

3.1. Motivation: for Distributed Timed Protocol 39

3.2. Timed Multiparty Global Session Types 40

3.2.1. Syntax of Timed Global Types 40

3.2.2. Labelled Transition System for Timed Global Types . . . 42

3.3. Timed Local Types 44

3.3.1. Syntax of Timed Local Types and Projection Definition . . 44

3.3.2. LTS for Timed Local Types 46

3.4. Multiparty Session Processes with Delays 48

3.4.1. Syntax of Timed Session Calculus 49

3.4.2. Structural Equivalence and Proof Rules 51

3.4.3. Extension to Runtime Processes . . . 52
3.5. Type Preservation of Timed Processes . . . 56
3.6. Time-Progress of Timed Processes . . . 69
3.6.1. Feasibility and Wait-freedom . . . 69
3.6.2. Auxiliary Definitions for Reasoning on Clocks . . . 70
3.6.3. Time Graphs . . . 71
3.6.4. Checking Feasibility and Wait-Freedom . . . 72
3.6.5. Time-Progress for Processes . . . 76
3.7. Related Works . . . 82
3.8. Summary . . . 86

4. Choreographical Multiparty Session Types 87
4.1. Motivation: Fork and Join . . . 87
4.2. Global Types for Choreographical Multiparty Sessions . . . 89
4.2.1. Syntax of Global Types . . . 89
4.2.2. Well-formed Global Types . . . 92
4.3. Local Types and Projections . . . 97
4.4. General Multiparty Session Processes . . . 101
4.4.1. Syntax of Choreographical Session Calculus . . . 101
4.4.2. Typing Multiparty Interactions . . . 105
4.4.3. Typing for Run-time Processes . . . 109
4.5. Safety of Typed Multiparty Session Processes . . . 112
4.6. Related Works . . . 125
4.7. Summary . . . 126

5. Timed Choreographical Multiparty Session Types 127
5.1. Motivation: An Application of Time and Parallel Combination . . . 127
5.2. Timed Choreographical Global Types . . . 129
5.3. Timed Choreographical Local Types . . . 134
5.4. Timed Choreographical Processes . . . 139
5.4.1. Syntax of Process Calculus . . . 140
5.4.2. Typing Timed Choreographical Processes . . . 141
5.4.3. Typing Run-Time Processes . . . 145
5.5. Safety of Timed Choreographical Multiparty Session Types . . . 147

5.6. Sufficient Conditions for Progress Enablement and Time-progress of Processes . . . 158
5.6.1. Time Graph . . . 159
5.6.2. Checking Progress Enabling . . . 160
5.6.3. Time-Progress for Timed Choreographical Processes . . . 162
5.7. Related Works . . . 166
5.8. Summary . . . 167

6. Conclusion and Future Work **168**
6.1. Conclusion . . . 168
6.2. Future Work . . . 171
6.2.1. Further Improvement on Timed Choreographical Session Types . . . 172
6.2.2. Possible Extensions to Multiparty Session Types . . . 174
6.2.3. Implementation . . . 176

Bibliography **178**

A. Supporting Materials for Chapter 4 **188**
A.1. Appendix for Section 4.2.1 . . . 188
A.2. Appendix for Section 4.4 . . . 191
A.3. Examples for global types and well-formed inference . . . 191

List of Figures

1.1. Vision of Applying Session Types to Programming 15

2.1. Syntax of the π-calculus 20
2.2. Basic Reduction Rules of the π-calculus 20
2.3. Major Development of Session Types 21
2.4. Session Calculus Syntax in [YV07] 22
2.5. Duality Principle for Type Checking 23
2.6. Session-based Distributed Programming 24
2.7. An Execution of the Three Buyer Protocol [CDCYP15] 25
2.8. Global and Session Types 26
2.9. Process Syntax and Naming Conventions 27
2.10. Implementation of the Three Buyer Protocol. 28
2.11. Timed Automaton with Two Clocks [AD94] 30
2.12. Example of Statecharts and How XOR-Composition is achieved . 32
2.13. Illustration of Default State 33
2.14. Illustration of 'Enter-by-History' 33
2.15. Example of Entering State Group according to Conditions 34
2.16. Example of Entering State Group by Selection 34
2.17. Example of Orthogonality 35
2.18. Illustration of Time Constraints in Harel's Statecharts 36
2.19. Basic Elements of BPMN 37
2.20. Construction of a Choreography Task 37
2.21. An Example of Choreography 38

3.1. Message Sequence Chart-like Notation of the Three-party Protocol 40
3.2. Labelled Transition for Timed Global Types 44
3.3. Labelled Transition Rules for Local Timed Types 47
3.4. Reduction for processes 50
3.5. Proof Rules for Programs 53

3.6. Proof Rules for Processes (the rules on the top are for queues) . . 55
3.7. Reduction Rules for Specifications 55
3.8. Extended Syntax of Processes for Time-error Freedom. 65
3.9. Modified/Added Rules of Extended Reductions 67
3.10. Extended Reduction Rules with Error Processes 68
3.11. Examples of non-feasible global types and their amended feasible versions 69

4.1. Register Example: Global View in Activity Diagram (Left) v.s. Local View in State Transition Diagram (Right) 88
4.2. Examples of Choreographical Global Types 93
4.3. Incorrect Global Types 95
4.4. Receiver Computation (up to permutation of | and $+$) 96
4.5. Linearity Computation (up to permutation of | and $+$) 97
4.6. Register Example: Local Types 99
4.7. Local Type Congruence (up to commutativity of | and +) 99
4.8. Local State Equivalence 100
4.9. Transition Rules for Local Choreographical Types 100
4.10. Operational Semantics of Session Calculus 104
4.11. Register Example: Processes 106
4.12. Typing System for State Processes 108
4.13. Structural Congruence for Processes and Networks 109
4.14. Typing Rules for Run-Time State Processes 110
4.15. Typing System for Queues 110
4.16. Typing System for Run-Time State Processes 111
4.17. Environment Transition System 111
4.18. Type Transition System, Structure Rules and Environment Communication Rule 112

5.1. BPMN Activity Diagram for Trading Example 129
5.2. Global Types for the Trading Example 131
5.3. Labelled Transition System for Timed Choreographical Global Types 132
5.4. Local Types for the Trading Example 136
5.5. Selected Transition Rules for Timed Choreographical Local Types 137
5.6. Operational Semantics of Timed Choreographical Session Calculus 142
5.7. Typing System for Timed Choreographical Processes (Part 1) . . . 143

5.8. Typing System for Timed Choreographical Processes (Part 2) and Network 144
5.9. Selected Structural Congruence Rules 145
5.10. Reduction Rules for Session Environment 147
5.11. Extended Operational Semantics with Error State 156
5.12. Extended Typing Rules with Time Assertions 157

6.1. Global Types for the Example 173
6.2. Harel's Statecharts for Example 6.1 174
6.3. BPMN choreography for Example 6.1 175
6.4. Scribble Example with Time Extension 177
6.5. An Example of Choreographical Timed Global Session Types with Interleaved Join/Merge 178
6.6. Illustration of Scribble Extension with State Variable x 178
6.7. Illustration of Scribble Extension with *goto* 179

A.1. Examples of Global Types (2) 189
A.2. Active Sender Computation 190
A.3. Example 4: global state machine 195

1. Introduction

This thesis is concerned with the research of extending the theory of multiparty session types. Session types model interactions among participants in the communication system and define a class of π-calculus that verifies the properties of behaviours such as safety. Extending the theory of multiparty session types allows us to model more real life scenarios and verify behaviours before implementation.

1.1. Motivation

The π-*calculus* [Mil99] is a process algebra for providing the semantics for a high-level concurrent distributed programming language. Compared with λ-calculus, another formal language for modelling computation, π-calculus allows channel names to be communicated over channels and is used for describing communicating systems, whose network configuration may change over time. In practice, π-calculus is good at modelling protocols that describe the communicating behaviours of processes. People use process calculi to model and verify prescribed protocols with support from model checking tools such as the Concurrency Workbench [CPS93] and SLMC [Cai04].

It is observed that in concurrent systems, interactions often occur with a significant amount of exchange of messages with various types. Then *session types* were introduced as a formal framework in the polyadic π-calculus with type systems, so as to allow the typing of channel names by structured sequences of types and represent the trace of usage of the channels. Session types were first proposed in binary form [THK94] and then developed to *Multiparty Session Types* (MPSTs) [BC08, CDCYP15]. In recent years, some works on extending session types are proposed to improve expressivity and allow session types to gain more computational properties, such as higher-order processes [MY07, MY09] and subtyping [SW01a, GH05]. Also, several works [VGR06, Vas12] embed the session concept into object-oriented programming practice so that one could achieve robust typing systems that are suitable for practical programming and

are supported by a sound theoretical foundation. Naturally, there are some implementations of session types by realising sessions directly in programming languages, such as in *Haskell* [NT04], *C#* [CNP01] and Java [HYH08, GVR+10]. Later, implementations of describing session types in high-level languages are proposed. We highlight Scribble [YHNN13] among those implementations. It is specially designed as a language for describing global (choreography) and local (service end-point) behaviours in session types. Extensible tools are available to perform validation of programs and to implement the verified specifications in different programming languages, such as in C [NYH12] and Java [HYH08].

The above practice of implementing session types into programming languages shows that session types could help programmers to produce *safe* programs. For example, [NCY15] shows that a safe and efficient MPI backbone code can be automatically generated from global specifications described in Scribble and programmers only need to fill in the details. But one problem is that we could not expect all software developers would learn session types for coding. The developers usually start with some graphs describing the behaviours and interactions that the program should implement, such as UML sequence charts and state transition diagrams. This observation inspires us to wonder if we could transform commonly used graphical formalisms into session types and our vision is shown in Figure 1.1. In this way, programmers could start to design their software using various methods and these blueprints will be transformed into (multiparty) session types. Then within the framework of session types and with the help of model checking tools, we could verify if this design is safe and error-free or not, and if the results reach the expectations, we could utilise tools like Scribble to produce backbone code from session types descriptions in different programming languages much as in [NCY15] and [HYH08]. One of the advantages is to speed up the software development life cycle, to save costs and gain efficiency for practical considerations. The other advantage is to avoid redundant programming efforts. If the software is expected to be implemented in different languages, without the help of session types, the programmers would need to write the program of the same design several times and could not guarantee the safety of their written code.

Since session types play such an important role in application development, we then investigate any characteristics that are commonly used in protocol specifications but are missing from current variants of session types. And it is found that we need to add time and fork/join to session types.

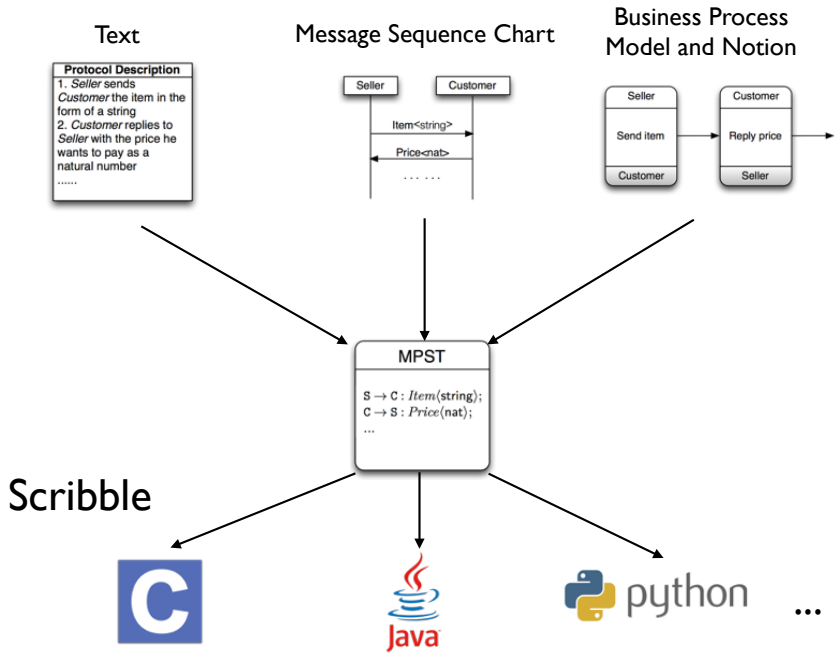


Figure 1.1.: Vision of Applying Session Types to Programming

Some models for describing and analysing system behaviours contain real-time specifications. For example, *Communicating Timed Automata* (CTAs) [KY06], which extend the theory of timed automata [AD94], enable a precise specification and verification of real-time distributed protocols. From the theoretical side, in the untimed setting recent work brings Communicating Finite State Machines into choreographic frameworks, by seeking correspondence with projected local types [DY13]. Inspired by this work, we proceed along these lines by applying the idealised mathematical semantics of CTAs to the design of timed MPSTs, which contain clocks, clock constraints, and resets, so as to fill the gap between the abstract specification by CTAs and the verification of real-time programs.

On the other side, some web service protocols include multiple thread operations. In the previous work in [DY12], the idea of choreographical multiparty session types has been proposed to treat expressive forking, merging and joining protocols that are absent from existing session frameworks. Based on this idea, we further complete the theory of choreographical MPSTs and combine this variant with timed MPSTs, so as to enable session types to express distributed parallel systems with time requirements.

1.2. Goal and Objectives

The goal of this PhD study is summarised as:

'To propose a theory that extends multiparty session types with time and parallelism, which consists of well-formed global/local types and session calculus, and enjoys safety and error-free properties.'

This goal is divided into following objectives:

1. On the foundation of session types, we define the theory of timed multiparty session types, including the global session type, local session types and the corresponding extended asynchronous π-calculus. The theorems of type safety and error-freedom should be verified by proof rules of timed session calculus.

2. Then we extend the current multiparty session types with parallelism by adding fork/join operations in the framework. The extended global types should guarantee a well-formedness condition and the corresponding multiparty session processes should satisfy properties, such as type safety.

3. We merge the above two extensions together to propose timed choreographical multiparty session types so as to be able to model concurrent real-time systems. This combined version should ensure the properties that either timed multiparty session types or choreographical session types own.

1.3. Contributions

This thesis shows the following contributions for the theoretical research on multiparty session types:

- Extend session types with time. This work is the first attempt to add timing information directly to session types in the literature. With this gap filled, multiparty session types could model real-time concurrent systems and programs with time requirements, such as timeout exceptions and deadlines.

- Investigate time-related properties for session types. After adding time to session types, we found that even if the programs are type safe, programs could still reduce to error because of bad timing for reduction. To prevent this situation from happening, we propose time-related theorems

and algorithms for guaranteeing safety and progress properties for MPSTs. [BYY14] is the published outcome of the above two points.

- Supplement the theory of choreographical session types. We explain how flexible fork/join operations are beneficial regarding thread management and state explosion. We also illustrate the syntax of global/local session types with more examples. The full proofs for theorems like subject reduction are finished.

- Propose the theory of timed choreographical session types. We define the syntax of global/local types and session calculus and illustrate with a use case. Besides, we prove the theorems of subject reduction, type safety and time error-freedom for timed choreographical processes and define progress enablement property and time-progress property of processes.

1.3.1. Thesis Outline

Chapter 2 Background briefly describes the theory of session types and timed automata. We start with the syntax of the π-calculus, which is fundamental to the session calculus, and then we present the theory of binary session types and how multiparty session types evolve from binary session types. Besides timed automata, we also show other graphical formalisms for expressing communication systems.

Chapter 3 Timed Multiparty Session Types presents the theory of timed MPSTs. Motivated by a distributed timed protocol, we define the semantics of global/local types for timed session types and labelled transition systems respectively. Then we show the corresponding timed session calculus and investigate the conditions that could ensure deadlock freedom of processes.

Chapter 4 Choreographical Multiparty Session Types presents the semantics of choreographical global types and local types and defines the conditions that guarantee the well-formedness of global types. We also present choreographical multiparty session calculus and theorems for subject reduction and type safety.

Chapter 5 Timed Choreographical Multiparty Session Types describes the theory of the combined variant of timed MPSTs and choreographical MPSTs. We present the semantics and labelled transition system for global/local

types and provide sufficient conditions for well-formedness. Then we define the corresponding session calculus and the theory of time error-freedom. Finally, we present sufficient conditions for guaranteeing the progress-enabling of global types and time-progress property of processes.

Chapter 6 Conclusion and Future Work summarises the main points of this thesis. Also, we discuss some future ideas that are composed of three parts: further development of timed choreographical session types like subtyping; possible extensions to the theory of multiparty session types like adding dependent types; and practical implementation issues.

1.3.2. Publications and Talks

Publications

- Laura Bocchi, Weizhen Yang, Nobuko Yoshida. Timed Multiparty Session Types. CONCUR 2014.

  This paper presents the theory of extension of multiparty session types with time, with connection to communicating timed automata. This work forms Chapter 3.

Presentation

- Generalised Timed Choreography. YR CONCUR. Madrid 2015, Spain

  At this workshop, we briefly presented the theory of timed choreographical session types, including the syntax of types, motivating example and key theorems. The abstract was submitted. This presentation forms part of Chapter 5.

2. Background

In this chapter, we present the background works that form the basis of this dissertation. In Section 2.1 we briefly introduce the π-calculus, which is fundamental for the theory of session types shown in Section 2.2. In Section 2.2, we give the original notions of session types, illustrated with an example. In the last Section 2.3, we show the basic theory of timed automata and other graphical representations for describing behaviours of the system.

2.1. π-Calculus

π-calculus is a process algebra proposed by Milner, Sangiorgi and Walker [Mil99, SW01b] and is formalised for describing and analysing concurrent mobile systems. It is viewed as a continuation of the process calculus, CCS (Calculus of Communicating Systems) [Mil82]. The syntax and reduction rules are shown in Figure 2.1 and Figure 2.2, respectively. π-calculus uses message-passing in transitions instead of value-passing in the CCS. The difference is that message-passing allows *channel names* to be communicated through channels. This feature allows π-calculus to model the networks whose configuration may change with time. Besides, π-calculus is expressive, as it can encode data structure, encode functions as processes, encode higher-order π, and encode polyadic communication with monadic [Mil91, Tur96] etc. As the extensions of π-calculus, spi-calculus [AG99] and applied π-calculus [RS11] are for reasoning about cryptographic protocols. Recent research [Puh06] shows that π-calculus can be used as the theoretical basis of Business Process Modelling.

Asynchronous π-calculus, which we are interested in, is a variant of π-calculus and is fundamental to session types. The reason for such importance is that asynchronous communication is viewed as a more realistic model for concurrency and other variants of π-calculus are translatable into asynchronous π-calculus. The difference between asynchronous π-calculus and synchronous π-calculus is that the asynchronous one requires no continuation after output. And processes in the

Syntax

$P, Q ::=$	processes
$\mathbf{0}$	nil process
$P \mid Q$	parallel composition
$(\nu a)P$	generation of a
$!P$	replication of
$\bar{u}\langle v \rangle.P$	output of v on channel u
$u(x).P$	input of *distinct* variables x on u

Figure 2.1.: Syntax of the π-calculus

Reduction rules

$$\bar{a}\langle v \rangle.P \mid a(x).Q \;\rightarrow\; P \mid Q\{^{v}/_{x}\} \qquad \text{[Com]}$$

$$\frac{P \;\rightarrow\; P'}{P \mid Q \;\rightarrow\; P' \mid Q} \qquad \text{[Par]}$$

$$\frac{P \;\rightarrow\; P'}{(\nu a)P \;\rightarrow\; (\nu a)P'} \qquad \text{[Res]}$$

$$\frac{P \equiv Q \;\rightarrow\; Q' \equiv P'}{P \;\rightarrow\; P'} \qquad \text{[Struct]}$$

Figure 2.2.: Basic Reduction Rules of the π-calculus

original π-calculus can be expressed by the asynchronous π-calculus by using an extra channel to send the acknowledgement from input processes.

2.2. Session Types

The 'session' concept, proposed in [HVK], represents a series of interactive communications. The session calculus is a session-based asynchronous π-calculus and is the theoretical support of session types. Session types are a typing system for the session calculus.

The development concerning session types is shown in Figure 2.3. The initial version of session types is binary session types, which only models the communication between two participants. Later multiparty session types are

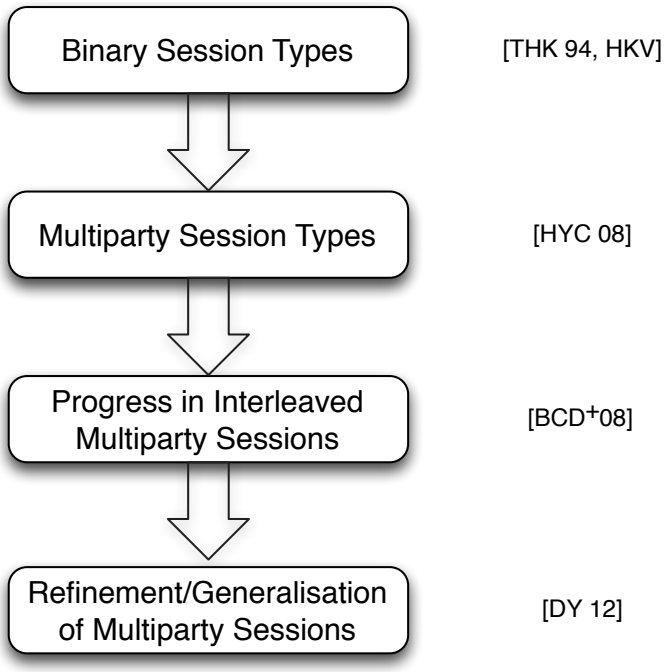


Figure 2.3.: Major Development of Session Types

proposed to enable modelling of communication among multiple participants. Next, a notion of multiparty sessions is presented to allow interleaved and interfering multiparty sessions. Other evolutions of session types are later proposed, like refinement/generalisation of multiparty sessions, session types with assertions. Here we mainly introduce binary session types and multiparty session types, as they are the basis of all other versions of session types.

2.2.1. Binary Session Types

As previously stated, the binary session types is fundamental, so we present it first. The theory of binary session types consists of basic language constructs and a type discipline for modelling communication-based concurrent systems. The language constructs express the interactions in the program in a simple way and they can be easily translated into asynchronous π-calculus. The type discipline not only provides a high-level type abstraction of the interactions, but also checks the typability between the processes in the program.

The syntax is shown in Figure 2.4. The process $\overline{a}(k).P$ sends the request for

$$
\begin{array}{rll}
P & ::= \overline{a}(k).P & \text{session request} \\
& \mid a(k).P & \text{session acceptance} \\
& \mid \overline{k}\langle e\rangle & \text{data sending} \\
& \mid k(x).P & \text{data reception} \\
& \mid k \triangleleft l & \text{label selection} \\
& \mid k \triangleright \{l_1 : P_1 [] \cdots [] l_n : P_n\} & \text{label branching} \\
& \mid \overline{k}\langle k'\rangle & \text{channel sending} \\
& \mid k(k').P & \text{channel reception} \\
& \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{conditional branch} \\
& \mid P \mid Q & \text{parallel composition} \\
& \mid \mathbf{0} & \text{inaction} \\
& \mid (\nu u)P & \text{name/channel hiding} \\
& \mid \mathsf{def}\ D\ \mathsf{in}\ P & \text{recursion} \\
& \mid X[ek] & \text{process variables} \\
e & ::= c & \text{constant} \\
& \mid e + e' \mid e - e' \mid e \times e \mid \mathtt{not}(e) \mid \ldots & \text{operators} \\
D & ::= X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \mathsf{and} \cdots \mathsf{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n & \text{declaration for recursion}
\end{array}
$$

Figure 2.4.: Session Calculus Syntax in [YV07]

initiating a session via a name a and creates a fresh channel, and then P will use the channel for later interactions. The process $a(k).P$ is the symmetric action of receiving the request of session initiation via a and continues P with a new channel k. Data sending $\overline{k}\langle e\rangle$ and receiving $k(x).P$ are standard synchronous message passing, where e denotes an expression like arithmetic/boolean operations and names. In label selecting $k \triangleleft l$ and branching $k \triangleright \{l_1 : P_1 [] \cdots [] l_n : P_n\}$, the labels $l_1, ..., l_n$ are pairwise distinct. The channel sending and reception are also known as delegations that allow passing a channel being used in one session to another process. In conditional branch, e should be a Boolean value. The name/channel hiding binds a local scope name or channel to the process. In recursion $\mathsf{def}\ D\ \mathsf{in}\ P$, process P can occur one or more times.

The goal of the session concept is to provide a basic way of describing complex interactions with clear and well-organised abstractions. The session calculus has three primitives: value passing, label branching and delegation: the value passing

$$\overline{\uparrow[\tilde{S}];\alpha} = \downarrow[\tilde{S}];\overline{\alpha} \qquad \overline{\oplus\{l_i:\alpha_i\}} = \&\{l_i:\overline{\alpha_i}\} \qquad \overline{\uparrow[\alpha];\beta} = \downarrow[\alpha];\overline{\beta}$$

$$\overline{\downarrow[\tilde{S}];\alpha} = \uparrow[\tilde{S}];\overline{\alpha} \qquad \overline{\&\{l_i:\alpha_i\}} = \oplus\{l_i:\overline{\alpha_i}\} \qquad \overline{\downarrow[\alpha];\beta} = \uparrow[\alpha];\overline{\beta}$$

$$\overline{1} = 1 \qquad \overline{t} = t \qquad \overline{\mu t.\alpha} = \mu t.\overline{\alpha}$$

Figure 2.5.: Duality Principle for Type Checking

feature is inherited from π-calculus; label branching is used for structuring external choice, and without this feature, the session calculus and session types will be limited; delegation allows the entity to delegate the processes of a certain request to another entity, especially if we pass parts of responsibilities of the parent process to the subprocesses, then we can assign processing to lower level or smaller processes, without sacrificing the benefits of using session types, as all delegated processes can follow a subset of session types from the top level. It is important to note that the top-level process does not need to be aware of the delegation, which allows us to design distributed systems with session types from a higher-level view.

Supported by the simple and flexible session calculus, session types provide a basic typing principle for communication-based programming. With the help of typing systems, we could check if the program is coherent concerning communication structure and if it contains errors or not.

The typability of the program ensures two communicating processes are in a compatible pattern. We present the basic notion of sorts that defines what kind of message is sent and the notion of types that abstract the interaction to be done over the channel.

Definition 2.2.1 (Types)**.** Given *type variables (t,t',...)* and *sort variables (s,s',...), sorts (S,S',...)* and *types* $(\alpha, \beta, ...)$ are defined by the following grammar.

$$\begin{array}{lcl} S & ::= & \mathit{nat} \mid \mathit{bool} \mid \langle \alpha, \overline{\alpha} \rangle \mid s \mid \mu s.S \\ \alpha & ::= & \downarrow[\tilde{S}];\alpha \mid \downarrow[\alpha];\beta \mid \&\{l_1:\alpha_1,...,l_n:\alpha_n\} \mid \mathbf{1} \mid \bot \\ & \mid & \uparrow[\tilde{S}];\alpha \mid \uparrow[\alpha];\beta \mid \oplus\{l_1:\alpha_1,...,l_n:\alpha_n\} \mid t \mid \mu t.\alpha \end{array}$$

The way of examining typability is to apply the duality principle, whose inductive definition is shown in Figure 2.5. With duality checking, we could ensure the program will run without undesirable program behaviour caused by a discrepancy between different data types.

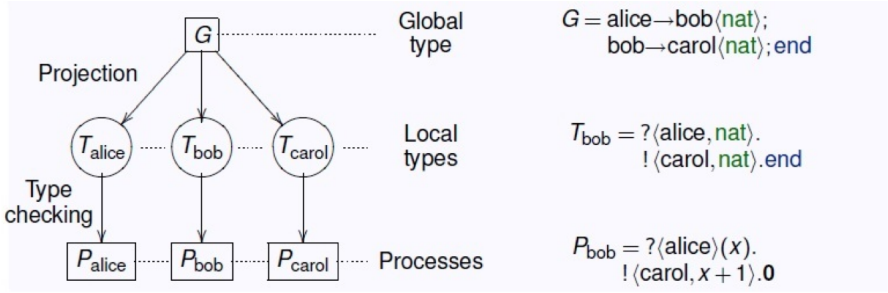


Figure 2.6.: Session-based Distributed Programming

2.2.2. Multiparty Session Types

The problem with binary session types is that it limits the session to be owned by two participants. Motivated by this, multiparty session types [YV07] are proposed to close the gap. The framework of multiparty session types is shown in Figure 2.6, which illustrates the general steps of distributed programming in a session-based approach.

1. The intended communication protocols should be described in a global session type, which specifies the protocol that all participants should follow in the communication.

2. The global type is projected to endpoints to form the local projection for each participant, which gives a local view of the intended protocol.

3. The typing system will perform the checking of processes.

This approach is a top-down style: we start with a global specification, and end with well type-checked processes. It is usually assumed that global types are complete, that is, all interactions between participants should be specified. Some works like [CDCP11] loosen the assumption for global types, and some works like [LTY15] realise the bottom-up manner, that is, they construct global types from processes and local types.

This top-down framework is classic and is applied through the main chapters of the thesis. We show a three buyer example presented in [CDCYP15] to illustrate this classic framework. This example consists of three participants: Alice, Bob, and Carol, who cooperate with each other for buying a book from a Seller. A protocol for three participants is described informally below:

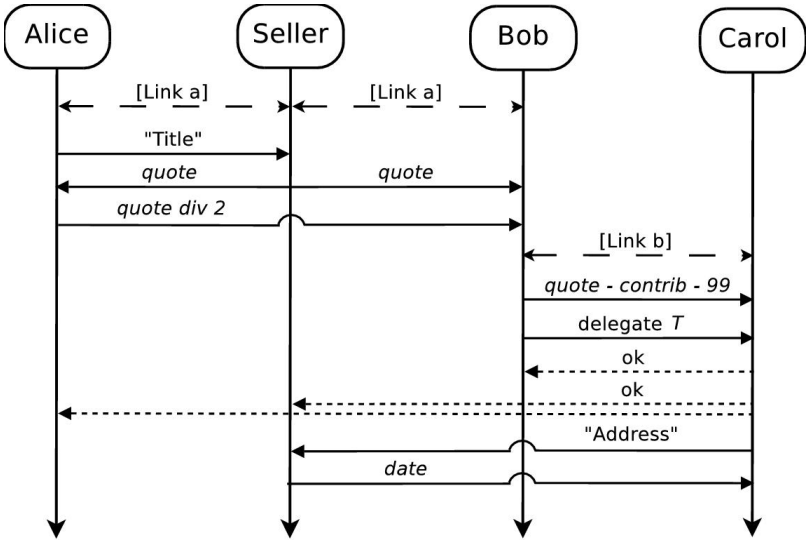


Figure 2.7.: An Execution of the Three Buyer Protocol [CDCYP15]

1. Alice sends a book title to Seller and Seller sends back a quote to Alice and Bob. Then Alice tells Bob the amount that she can contribute.

2. If the remaining price is within Bob's budget, Bob accepts the offer by notifying both Seller and Alice, then sends his address to Seller and Seller answers with the delivery date.

3. If the price exceeds Bob's budget, Bob asks Carol to collaborate and establish a new session. Bob tells Carol the amount she has to contribute and *delegates* the remaining interactions with Alice and Seller to her.

4. If Carol's contribution is within her budget, she accepts the quote, notifies Alice, Bob and Seller, and continues the rest of the protocol with Seller and Alice *as if she were Bob*. Otherwise, she notifies Alice, Bob and Seller to quit the protocol.

Figure 2.7 depicts an execution of the above protocol where Bob asks Carol to collaborate (by delegating the remaining interactions with Alice and Seller) and the transaction terminates successfully.

To correctly model this protocol, the syntax of global types and local types is shown in Figure 2.8.

Given the protocol, the global types can be presented as G_a and G_b. G_a describes the communication among Alice, Bob and Seller and G_b shows the

S	::=	$\mathsf{bool} \mid \ldots \mid \mathsf{G}$	Sorts
U	::=	$S \mid \mathsf{T}$	Exchange types
	Global types		
G	::=	$\mathtt{p} \to \Pi : \langle S \rangle.G$	Value exchange
	$\mid$	$\mathtt{p} \to \mathtt{p}' : \langle \mathsf{T} \rangle.G$	Channel exchange
	$\mid$	$\mathtt{p} \to \Pi : \{l_i : G_i\}_{i \in I}$	Branching
	$\mid$	$\mu \mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end}$	Recursion/end
	Session types		
T	::=	$!\langle \Pi, S \rangle.T$	*send value*
	$\mid$	$!\langle \mathtt{p}, \mathsf{T} \rangle.T$	Send channel
	$\mid$	$?(\mathtt{p}, U).T$	Receive
	$\mid$	$\oplus \langle \Pi, \{l_i : T_i\}_{i \in I} \rangle$	Selection
	$\mid$	$\&(\mathtt{p}, \{l_i : T_i\}_{i \in I})$	Branching
	$\mid$	$\mu \mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end}$	Recursion/end

Figure 2.8.: Global and Session Types

delegated interactions between Bob and Carol.

$$
\begin{aligned}
\mathsf{G}_a \;=\; & \\
& (1)\quad 2 \longrightarrow 3 : \quad \langle \mathsf{string} \rangle. \\
& (2)\quad 3 \longrightarrow \{1,2\} : \langle \mathsf{int} \rangle. \\
& (3)\quad 2 \longrightarrow 1 : \quad \langle \mathsf{int} \rangle. \\
& (4)\quad 1 \longrightarrow \{2,3\} : \{\mathsf{ok} : \quad 1 \longrightarrow 3 : \langle \mathsf{string} \rangle. \\
& (5)\quad \qquad\qquad\qquad\quad 3 \longrightarrow 1 : \langle \mathsf{date} \rangle.\mathsf{end}, \\
& (6)\quad \qquad\qquad\quad \mathsf{quit} : \mathsf{end}\} \\
\mathsf{G}_b \;=\; & \\
& (1)\quad 2 \longrightarrow 1 : \langle \mathsf{int} \rangle. \\
& (2)\quad 2 \longrightarrow 1 : \langle \mathsf{T} \rangle. \\
& (3)\quad 1 \longrightarrow 2 : \{\mathsf{ok} : \mathsf{end}, \mathsf{quit} : \mathsf{end}\} \\
\mathsf{T} \;=\; & \oplus \langle \{2,3\}, \{\mathsf{ok} : !\langle 3, \mathsf{string} \rangle.?(3, \mathsf{date}).\mathsf{end}, \mathsf{quit} : \mathsf{end}\} \rangle
\end{aligned}
$$

The calculus of multiparty sessions is shown in Figure 2.9.

Figure 2.10 shows an implementation of the protocol presenting G_a and G_b in the processes Seller, Alice, Bob, and Carol by the session calculus. By applying the typing rules in [CDCYP15], we prove that this implementation is type-safe.

P	::=	$\bar{u}[\mathtt{p}](y).P$	Multicast request
	|	$u[\mathtt{p}](y).P$	Accept
	|	$c!\langle \Pi, e\rangle; P$	Value sending
	|	$c?(\mathtt{p}, x); P$	Value reception
	|	$c!\langle\langle \mathtt{p}, c'\rangle\rangle; P$	Channel delegation
	|	$c?((\mathtt{q}, y)); P$	Channel reception
	|	$c \oplus \langle \Pi, l\rangle; P$	Selection
	|	$c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$	Branching
	|	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional
	|	$P \mid Q$	Parallel
	|	$\mathbf{0}$	Inaction
	|	$(\nu a : \mathsf{G})P$	Service name hiding
	|	$\mathsf{def}\ D\ \mathsf{in}\ P$	Recursion
	|	$X\langle e, c\rangle$	Process call
	|	$(\nu s)P$	Session hiding
	|	$s\colon h$	Message queue
D	::=	$X(x, y) = P$	Declaration
$\mathcal{E}$	::=	$[\,] \mid P \mid (\nu a : G)\mathcal{E}$	Evaluation context
	|	$(\nu s)\mathcal{E} \mid \mathsf{def}\ D\ \mathsf{in}\ \mathcal{E}$	
	|	$\mathcal{E} \mid \mathcal{E}$	
a, b			Service name
x			Value variable
y, z, t			Channel Variable
s			Session name
$\mathtt{p}, \mathtt{q}$			Participant number
X, Y			Process variable
l			Label
$s[\mathtt{p}]$			Channel with role
u	::=	$x \mid a$	Identifier
v	::=	$a \mid \mathsf{true}$	Value
	|	false	
e	::=	$v \mid x$	
	|	$e\ \mathsf{and}\ e'$	Expression
	|	$\mathsf{not}\ e \ldots$	
Π	::=	$\{\mathtt{p}\} \mid \{\mathtt{p}\} \cup \Pi$	Set of participants
c	::=	$y \mid s[\mathtt{p}]$	Channel
m	::=	$(\mathtt{q}, \Pi, v)$	Message in transit
	|	$(\mathtt{q}, \mathtt{p}, s[\mathtt{p}'])$	
	|	$(\mathtt{q}, \Pi, l)$	
h	::=	$h \cdot m \mid \varnothing$	Queue

Figure 2.9.: Process Syntax and Naming Conventions

$$
\begin{aligned}
\text{Seller} &= \bar{a}[3](y).y?(1,\mathit{title}).y!\langle\{1,2\},\mathsf{quote}\rangle.\\
&\quad y\&(2,\{\mathsf{ok} : y?(2,\mathit{address}).y!\langle 2,\mathsf{date}\rangle.\mathbf{0},\mathsf{quit} : \mathbf{0}\})\\
\\
\text{Alice} &= a[2](y).y!\langle 3,\texttt{"Title"}\rangle;y?(3,\mathit{quote}));y!\langle 2,\mathit{quote}\,\mathsf{div}\,2\rangle;\\
&\quad y\&(2,\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\})\\
\\
\text{Bob} &= a[1](y).y?(3,\mathit{quote});y?(1,\mathit{contrib});\\
&\quad \mathsf{if}\ (\mathit{quote}-\mathit{contrib} < 100)\ \mathsf{then}\\
&\qquad y\oplus\langle\{1,3\},\mathsf{ok}\rangle.y!\langle 3,\texttt{"Address"}\rangle;y?(3,\mathit{date});\mathbf{0}\\
&\quad \mathsf{else}\\
&\qquad \bar{b}[2](z).z!\langle 1,\mathit{quote}-\mathit{contrib}-99\rangle;z!\langle\langle 1,y\rangle\rangle;z\&(1,\{\mathsf{ok} : \mathbf{0},\mathsf{quit} : \mathbf{0}\})\\
\\
\text{Carol} &= b[1](z).z?(2,x).z?((2,t)).\\
&\quad \mathsf{if}\ (x < 100)\ \mathsf{then}\\
&\qquad z\oplus\langle 2,\mathsf{ok}\rangle.t\oplus\langle\{1,3\},\mathsf{ok}\rangle.t!\langle 3,\texttt{"Address"}\rangle.t?(3,\mathit{date}).\mathbf{0}\\
&\quad \mathsf{else}\\
&\qquad z\oplus\langle 2,\mathsf{quit}\rangle.t\oplus\langle\{1,3\},\mathsf{quit}\rangle.\mathbf{0}
\end{aligned}
$$

Figure 2.10.: Implementation of the Three Buyer Protocol.

2.3. Timed Automata and Other Graphical Models

This section is concerned with visual formalisms for modelling behaviours of systems. Here we present three well-known formalisms, *timed automata*, *Harel's statecharts* and *Business Process Modelling and Notation (BPMN)*. As one of the objectives of this dissertation, we aim to establish the relationship between session types and these graphical models.

2.3.1. Timed Automata

In automata theory, timed automata are used to model and analyse time-related behaviours in the systems, such as real-time systems and networks. A timed automaton is a finite automaton that is coupled with real-time values so that timed automata could represent behaviours of real-time systems. A timed automaton owns a finite number of clocks, denoted as set C, and it is assumed that all the clock values increase at the same pace. The transition is guarded by time constraint δ and clocks can be reset during the transition.

To define timed automata formally, we need to say what type of clock constraints are allowed on the edges. The simplest form of a constraint compares

a clock value with a time constant. We allow only the Boolean combinations of such simple constraints.

Definition 2.3.1. For a set of X of clock variables, the set $\phi(X)$ of clock constraints is defined inductively by

$$\delta \quad ::= \quad \mathsf{true} \mid x > c \mid x = c \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where c is a bound constant in $\mathbf{Q}$, the set of nonnegative rationals and $x \in X$ is a clock.

A clock interpretation ν for the set of clocks X maps a real value to each clock, that is, from X to $\mathbb{R}^{\geq 0}$. We say that a clock interpretation ν for X satisfies a clock constraint δ over X iff δ evaluates to true using the values given by ν.

Now we have the definition of timed automaton.

Definition 2.3.2. A *timed automaton* $\mathcal{A}$ is a tuple $(Q, q_0, \mathcal{A}ct, X, E, F)$ such that

- Q is the set of states,
- $q_0 \in Q$ is the initial state,
- $\mathcal{A}ct$ is the alphabet,
- X are the clocks,
- $E \subseteq (Q \times Q \times \mathcal{A}ct \times 2^X \times \phi(X))$ are the transitions, where 2^X is the reset predicate, $\phi(X)$ the clock constraints on X,
- $F \subseteq Q$ are the final states.

An edge $(q, q', a, \lambda, \delta)$ from E is a transition from state q to q' with action a, clock constraint δ and clock resets λ. The timed automation $\mathcal{A}$ starts in one of its start states at time 0 with all its clocks initialised to 0. As time passes, the values of all clocks increase, which reflect the elapsed time. At time τ_i, $\mathcal{A}$ changes state from q to q' using some transition of the form $(q, a, \delta, \lambda, q')$ reading the input a, if the current values of clocks satisfy δ. With this transition, the clock specified in λ is reset to 0, and thus starts counting time with respect to the time of occurrence of this transition. This behaviour is captured by defining runs of timed transition tables. A run shows the state and the values of all the clocks along a transition path. Figure 2.11 shows a simple example of timed automata.

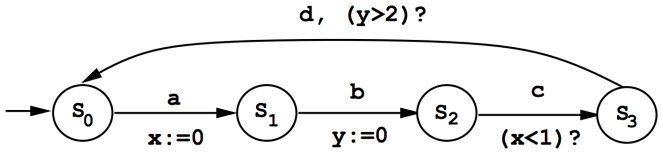


Figure 2.11.: Timed Automaton with Two Clocks [AD94]

For a time sequence $\tau = \tau_1\tau_2...$ we define $\tau_0 = 0$ and τ_i gives the time of occurring ith event.

Definition 2.3.1. *A run r, denoted by $(\bar{q}, \bar{\nu})$, of a timed automation $(Q, q_0, \mathcal{A}ct, X, E, F)$ over a timed word (a, τ) is an infinite sequence of the form*

$$r : \langle q_0, \nu_0 \rangle \xrightarrow[\tau_1]{a_1} \langle q_1, \nu_1 \rangle \xrightarrow[\tau_2]{a_2} \langle q_2, \nu_2 \rangle \xrightarrow[\tau_3]{a_3} ...$$

with $q \in Q$ and $\nu \in [X \to R]$, for all $i \geq 0$, satisfying the following requirements:

- Initiation*: $q_0 \in Q_0$, and $\nu_0(x) = 0$ for all $x \in X$.*
- Consecution*: for all $i \in \geq 1$, there is an edge in E of the form $(q_{i-1}, a_i, \delta_i, \lambda_i, q_i)$ such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies δ_i and $\nu_i = [\lambda_i \mapsto 0](\nu_{i-1} + \tau_i - \tau_{i-1})$.*

The set $inf(r)$ consists of those states $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$.

Later in Chapter 3 and 5, we adapt the definition of clock constraints and resets in the syntax of timed session types and since we need to analyse the traces of timed global and local types, we also define timed runs similarly to Definition 2.3.1.

2.3.2. Petri Nets

Petri nets are a tool to model concurrent systems and reason about them. Petri nets not only offer a graphical notation to represent processes with choice, recursion and concurrent operations, like UML activity diagrams, but also have an exact mathematical definition for execution processes and a well-developed mathematical theory for analysing processes. We present the definition of Petri nets below.

Definition 2.3.3. A *net* N is a tuple (S,T,F) such that

- S is a set of places,
- T is a set of transitions and $S \cap T = \emptyset$,
- F is a set of directed arcs and $F \subseteq (S \cup T) \times (S \cup T)$ satisfying $F \times (S \cup S) = F \times (T \cup T) = \emptyset$.

All places and transitions are said to be *elements* of N. For an element x of N, its *preset* $\cdot x$ is $\cdot x = \{y \in N \mid (y,x) \in F_N\}$, and its *postset* $x\cdot$ is $x\cdot = \{y \in N \mid (x,y) \in F_N\}$. An *undirected path* is a nonempty sequence $x_0 \ldots x_k$ of elements satisfying $x_i \in \cdot x_{i-1} \cup x_{i-1}\cdot$ for each i $(0 \leq i \leq k)$. The net is *weakly connected* if, for each two elements x and y, there exists an undirected path leading from x to y.

The states of a Petri net are defined by its markings, and state changes are caused by the occurrences of transitions. A *marking* of a net N is a mapping $m : S_N \to \mathbb{N}$ where $\mathbb{N} = 0,1,2,\ldots$. A place s is marked by a marking m if $m(s) > 0$. A *marked net* is a net equipped with a marking, called *initial marking*, denoted as m_0. Let $b \in \mathbb{N}$. A place s of a marked net is called *bounded* with bound b *(b-bounded)* if $m(s) < b$ for each reachable marking m. Marked nets with only 1-bounded places are usually called 1-safe or safe marked nets.

A transition t is *enabled* by a marking m if m marks all places in $\cdot t$. The transition t is called *dead* at a marking m if it is not *enabled* at any marking reachable from m. A marked net is said to be *live* if there is no dead transition at any reachable marking. Therefore, each live marked net with at least one transition is *deadlock-free*.

It is often required that every place of a marked net eventually can be marked. If no transition of a marked net is dead then each non-isolated place is marked at some *reachable* marking.

The above definitions are used in defining thread-correctness for choreographical session types in Section 4.2.2.

2.3.3. Harel's Statecharts

Harel's statecharts [Har87] is proposed on the basis of state transition diagrams. It provides a visual formalism for the specification of reactive systems. Since it is an extension of state transition diagrams of finite state machines, it uses states and events/state transition as a natural medium for describing the dynamic behaviours of a complex system.

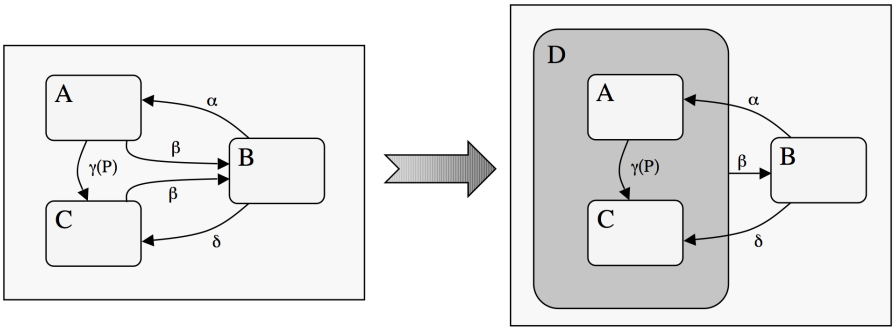


Figure 2.12.: Example of Statecharts and How XOR-Composition is achieved

As reactive systems become large-scale and complex, their specification and design become a problem. If we continued to use the original state transition diagram for modelling those systems, it would result in an unmanageable, exponentially growing multitude of states, all of which have to be arranged in a 'flat' unstratified fashion. This arrangement leads to an unstructured, unrealistic, and chaotic state diagram. Thus, Harel's statecharts are proposed to solve the problem by way of providing a formalism of describing states and transitions of the system in a modular fashion and of employing hierarchy and orthogonality/concurrency.

We use rounded rectangles to denote states at any level, and use encapsulation to show the hierarchy relation. Arrows are allowed to originate and terminate at any level. An arrow will be labelled with an event (or an abbreviation of one) and optionally also with a parenthesized condition.

Figure 2.12 gives an example and illustrates how XOR-composition is made. We have three states A, B, and C. The event γ occurring in state A transfers the system to state C, if and only if condition P holds at the time of occurrence. Since event β transits the system to B from either A or C, we can cluster the latter two states into a new super-state D and replace the two β arrows by one. Thus, D is then the exclusive-or (XOR) composition of A and C; in other words, one must be either in A or in C. The superstate D can be viewed as an abstraction of A and C, which captures one common feature of A and C, that is, event β leads to B. By clustering states into superstates, statecharts could economise in the number of arrows.

The default state is also defined as the start state of the system when entering

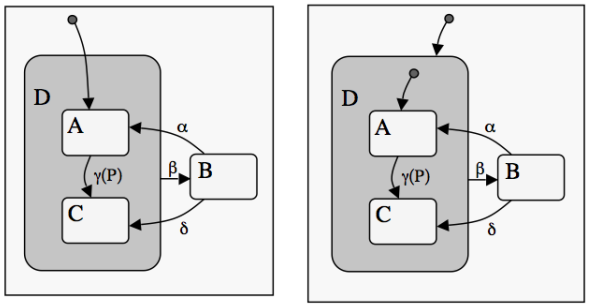


Figure 2.13.: Illustration of Default State

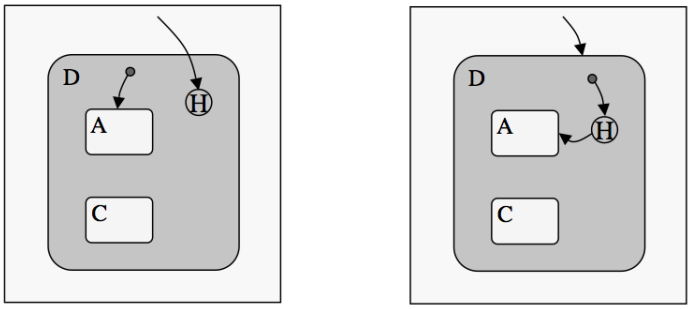


Figure 2.14.: Illustration of 'Enter-by-History'

from the 'outside' world, as illustrated in Figure 2.13.

It is observed that one of the most interesting and frequent ways of entering a group of states is by the system's history in that group. The simplest kind of this 'enter-by-history' is entering the state most recently visited, as shown in Figure 2.14. It should be noted that history is applied only on the level in which it appears.

There are two other ways of entering a group of states. One is through conditions. In this method, a connective C is used to abbreviate more complicated entrances to substates of a superstate with a simple, direct arrow, as illustrated in Figure 2.15.

The other way is to enter through selection connective S. Selection occurs when the entering state is determined by the 'value' of a generic event so that the event is the selection of one of a number of clearly defined options and the connective has chosen to model those options as states. An example is shown in Figure 2.16.

Orthogonality is another important feature of Harel's statecharts, which is

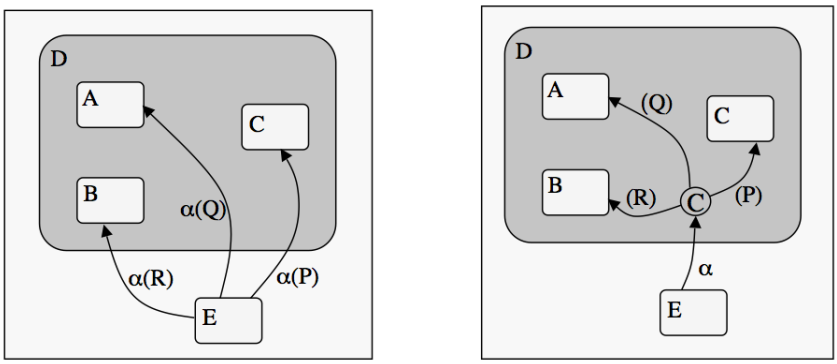


Figure 2.15.: Example of Entering State Group according to Conditions

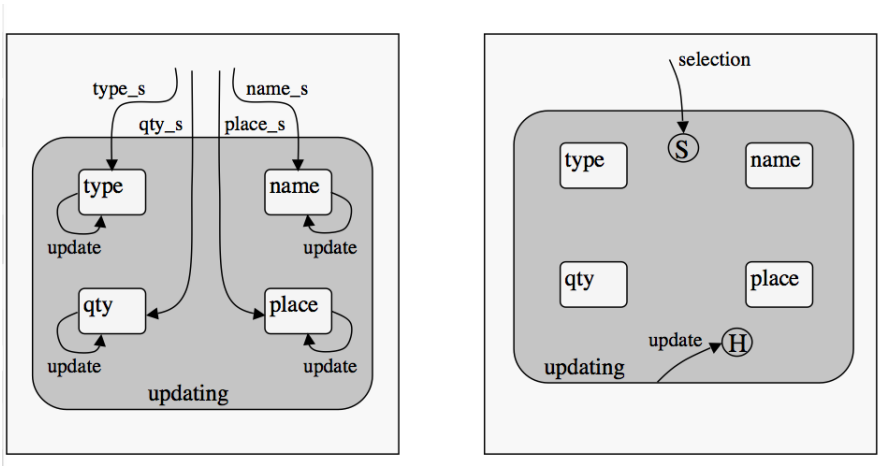


Figure 2.16.: Example of Entering State Group by Selection

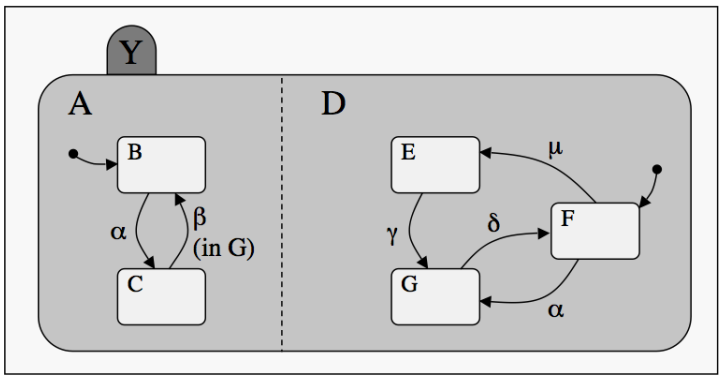


Figure 2.17.: Example of Orthogonality

explained through an example in Figure 2.17. The notation used in statecharts is the physical splitting of a box into components using dashed lines. A state Y consists of AND components A and D, with the property that being in Y indicates being in some combination of B or C with E, F or G. Then we view that Y is the orthogonal product of A and D. Without any additional information, entering Y from the outside is entering the combination (B, F) by the default arrows. If event α occurs, it transfers B to C and F to G simultaneously, resulting in the new combined state (C, G). This illustrates a certain kind of synchronisation: a single event causing two simultaneous happenings. On the other hand, the occurrence of event μ at (B,F) only affects D component, leading to (B,E), which illustrates some kind of independence. Both behaviours are part of the orthogonality of A and D, which is the term we use to describe the AND decomposition.

One of the additional features of Harel's statecharts is applying time constraints. This is done using the event

expression timeout(enter state, bound)

in which the enter state is the source of transition and the bound, including lower and upper bound, shows the time interval that the enter state is available.

In the discussion of future works in Section 6.2, we present an example to show how to express proposed timed choreographical session types in Harel's statecharts.

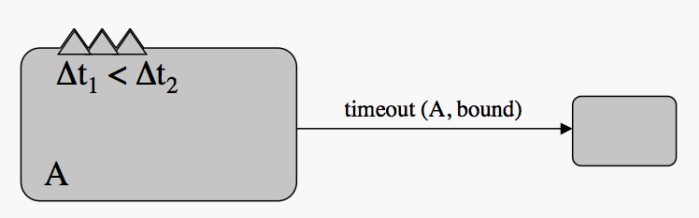


Figure 2.18.: Illustration of Time Constraints in Harel's Statecharts

2.3.4. Business Process Modeling and Notation (BPMN)

BPMN [BPM] is a standard graphical representation for modelling business processes. It is based on a flowchart technique similar to the activity diagram in Unified Modeling Language (UML) and is designed to particularly coordinate processes and messages flowing between different processes/participants. Nowadays business processes expand to multiple participants and this makes the coordination become complex. BPMN is developed as a standard modelling method to address this issue. Since BPMN targets both high-level business users and low-level process implementers, it is designed to be easy to read and understand for business users and to be able to generate a business diagram with enough details for physical implementation. One thing to note is the difference between BPMN and UML. UML employs an object-oriented approach for modelling applications and software design, while BPMN takes a process-oriented method for modelling systems. These two ways of presentation provide different views of the system; thus we can say they are compatible instead of competing.

The BPMN diagram consists of four basic elements: *event* denotes something which happens during the course of a process; *activity* describes the kind of work that must be done; *gateway* decides the forking and merging points of paths, depending on the given conditions; *connection* objects are used to connect those flow objects.

There are many branches of business diagrams in BPMN. The one we are most concerned with is Choreography. Choreography formalises the way that business participants coordinate their interactions, and it focuses on the exchange of information (messages) between participants. Choreography is a definition of expected behaviour, basically a procedural business contract, between interacting participants. It presents message exchanges and their logical relation as conversations.

The basic element is a BPMN Choreography task and in that sense, a Chore-

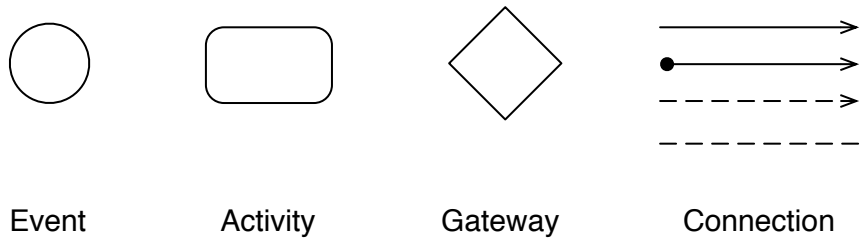


Figure 2.19.: Basic Elements of BPMN

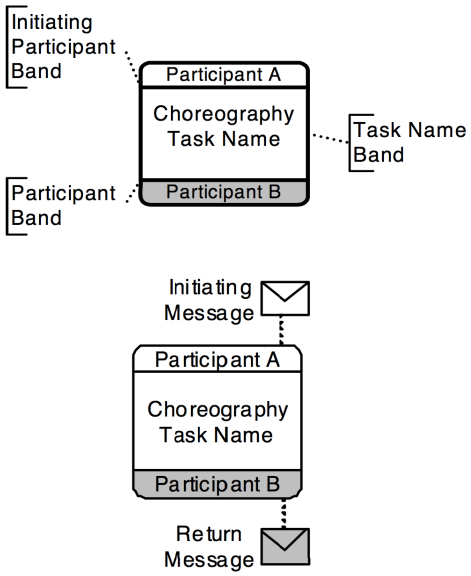


Figure 2.20.: Construction of a Choreography Task

ography defines the order in which Choreography tasks occur. A Choreography task represents an interaction, which is one or two message exchanges between two participants, as shown in Figure 2.20.

Figure 2.21 is an example of a Choreography diagram. In Section 6.2, we illustrate how to present timed choreographical session types in BPMN choreography diagram.

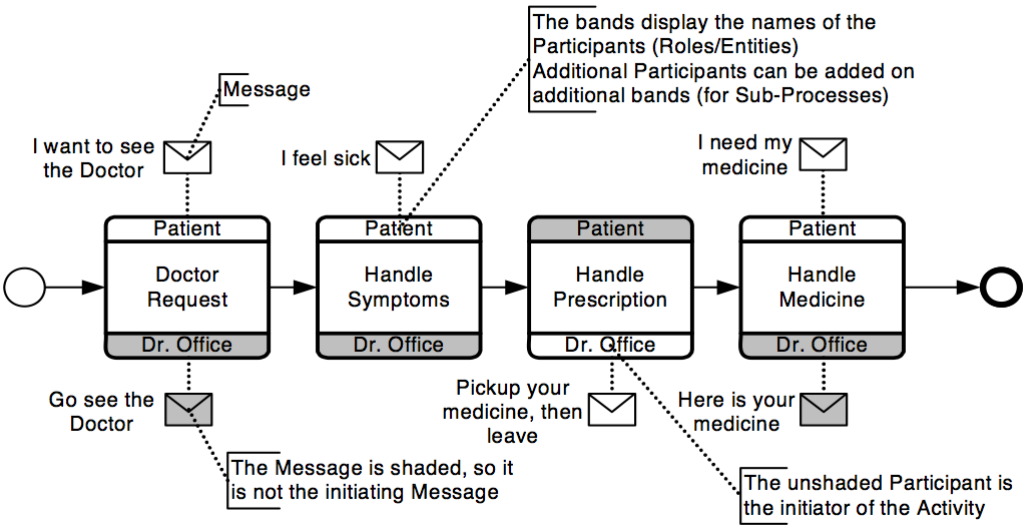


Figure 2.21.: An Example of Choreography

3. Timed Multiparty Session Types

This chapter introduces the theory of Timed Multiparty Session Types. First, we give the semantics of timed global types and local types and the projection algorithm. Then we give a simple π-calculus for programs with delays that can be used to synchronise the communications in a session. A compositional proof system for the π-calculus enables a modular verification for time-error freedom. In Section 3.6, we define the two conditions for ensuring that process will not deadlock because of inappropriate constraints and present time-progress property to guarantee that a process enjoys progress property if its untimed counterpart does not deadlock. The last parts are related works and the conclusion of this chapter.

3.1. Motivation: for Distributed Timed Protocol

The scenario, provided by our partner, the Ocean Observatories Initiative (OOI) [OOI], motivates us to add time in session types. The use case is about deploying a sensor network that consists of different ocean instruments controlled via remote service agents. In many OOI use-cases, execution time is important: requests are associated with deadlines; services are scheduled at specified time intervals. These time requirements can be expressed by combining global protocol descriptions from MPSTs and time from CTAs. We show the example of a three-party protocol that calculates the average water temperature via periodic sampling. In this protocol, we have three participants: a master `M` that initiates the sampling activity, a worker `W` that is a sensor with fixed and periodic response time w, and an aggregator service `A` that accumulates the data; their time constraints are expressed using clocks $x_{\mathtt{M}}$, $x_{\mathtt{W}}$, and $x_{\mathtt{A}}$, initially set to 0. We visualise the protocol using a Message Sequence Chart (MSC)-like notation. Delays l (average latency of the network) and w (sampling time) are expressed in milliseconds. Note that in this scenario, clocks can be reset at different times but time elapses at the same pace for all the participants of the protocol.

(1): Initially, `M` sends `W` a message of type `task` and then resets its clock $x_{\mathtt{M}}$

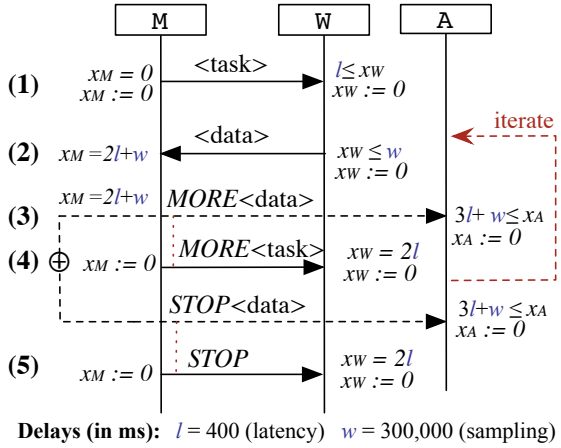


Figure 3.1.: Message Sequence Chart-like Notation of the Three-party Protocol

(setting $x_{\mathtt{M}} := 0$). After at least l, W can receive the message and reset $x_{\mathtt{W}}$.

(2): W completes the task and replies to M with a message of type `data` in at most w time units. M receives and resets $x_{\mathtt{M}}$ at time $x_{\mathtt{M}} = 2l + w$.

(3): M sends A a message of type `data` along with either label MORE, indicating the sampling continues for another iteration, or label STOP, which means that enough data has been gathered.

(4): If MORE was selected, then M assigns W another task with label MORE, resets $x_{\mathtt{M}}$ and another iteration begins.

(5): If STOP was selected then M tells W that this session ends.

3.2. Timed Multiparty Global Session Types

Global types [CDCYP15, HYC08] are specifications of the interactions (causalities and carried data types) of multiparty sessions. As they can be automatically projected onto a set of local types that describe the protocol from the view of each participant, global types are fundamental for local verification for processes. Thus it is necessary to start with the defining syntax of global types.

3.2.1. Syntax of Timed Global Types

We extend global types with constraints on clocks, yielding *timed global types* (*TG*s). We use the definitions from timed automata in Section 2.3.1: let X be a set of *clocks* ranged over by $x_1, \ldots, x_n$, taking values in $\mathbb{R}^{\geq 0}$ and n be the number

of participants in the protocol. A *clock assignment* $\nu : X \mapsto \mathbb{R}^{\geq 0}$ returns the time of the clocks in X. And $\nu + t$ returns the clock assignment mapping all $x \in X$ to $\nu(x) + t$. We write ν_0 for the initial assignment mapping all clocks to 0 (when the domain is understood). The set $\Phi(X)$ of *clock constraints* over X is defined as:

$$\delta \quad ::= \quad \mathsf{true} \mid x > c \mid x = c \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where c is a bound time constant in $\mathbb{Q}^{\geq 0}$ and we can derive false, $<$, $\leq$, $\geq$, $\vee$ in the standard way. The set of free clocks in δ, denoted as $fn(\delta)$, is inductively as:

$$fn(\mathsf{true}) = \emptyset \qquad fn(x > c) = fn(x = c) = \{x\}$$
$$fn(\neg\delta) = fn(\delta) \qquad fn(\delta_1 \wedge \delta_2) = fn(\delta_1) \cup fn(\delta_2)$$

We write $\delta(\tilde{x})$ when $fn(\delta) = \tilde{x}$, and let $\nu \models \delta$ denote that δ is satisfied by ν. A reset predicate over X, ranged over by λ, is a subset of X. The empty set of λ means that no reset occurs, otherwise the clock for each $x \in \lambda$ is set to 0. We write $[\lambda \mapsto 0]\nu$ for the clock assignment that is like ν except all clocks in λ are assigned to 0.

Participants ($\mathtt{p}, \mathtt{q}, \mathtt{p}_1, \ldots \in \mathbb{N}$) interact via asynchronous message exchange. An interaction consists of a send action and a receive action, and each action annotated with a clock constraint and a reset predicate. The constraint specifies when that action can be executed and the reset predicate specifies if clocks must be reset or not. For simplicity, we assume that each participant is associated with one clock and denote by $x_{\mathtt{p}}$ the clock of $\mathtt{p}$.

Definition 3.2.1 (Syntax of Global Types)**.** The syntax for sorts S, timed global types G is:

$$\begin{array}{lcl} S & ::= & \langle G \rangle \mid (T, \delta) \mid \mathsf{bool} \mid \mathsf{nat} \mid \ldots \\ G & ::= & \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{A_i\}.G_i\}_{i \in I} \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathtt{end} \\ A & ::= & \{\delta_O, \lambda_O, \delta_I, \lambda_I\} \end{array}$$

In G, type $\mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{A_i\}.G_i\}_{i \in I}$ models an *interaction*: $\mathtt{p}$ chooses a branch $i \in I$, where I is a finite set of branch indices, and sends $\mathtt{q}$ the branching label l_i along with a message of sort S_i. Then the session continues as G_i. It is noted that sort (T, δ) stands for session delegation. Each interaction branch is annotated with a *time assertion* $A_i = \{\delta_O, \lambda_O, \delta_I, \lambda_I\}$, where δ_O and λ_O defined on $\{x_{\mathtt{p}}\}$ are the clock constraint and reset predicate for the output action, and δ_I and λ_I defined on $\{x_{\mathtt{q}}\}$ are for the input action. We write $\mathtt{p} \to \mathtt{q} : \langle S \rangle \{A\}.G'$ for the interactions with only one branch. In *recursive type* $\mu\mathbf{t}.G$, a *type variable* $\mathbf{t}$ is associated to the recursion body G. We assume that type variables are guarded in the standard way

and `end` occurs at least once in G . We write $\mathcal{P}(G)$ as the set of participants of G.

Example 3.2.2 (Temperature calculation)**.** We show below the global timed type G for the protocol in §3.1 . We write $_$ for empty reset predicates.

$$
\begin{array}{lcl}
G & \stackrel{\text{def}}{=} & \mathtt{M} \to \mathtt{W} : \langle \mathtt{task} \rangle \{B_{\mathtt{O}}^1, B_{\mathtt{I}}^1\}.\mu \mathbf{t}.G' \\
G' & \stackrel{\text{def}}{=} & \mathtt{W} \to \mathtt{M} : \langle \mathtt{data} \rangle \{B_{\mathtt{O}}^2, B_{\mathtt{I}}^2\}. \\
 & & \mathtt{M} \to \mathtt{R} : \{ \mathsf{more} : \langle \mathtt{data} \rangle \{B_{\mathtt{O}}^3, B_{\mathtt{I}}^3\}. \\
 & & \qquad \mathtt{M} \to \mathtt{W} : \mathsf{more} \langle \mathtt{task} \rangle \{B_{\mathtt{O}}^4, B_{\mathtt{I}}^4\}.\mathbf{t}, \\
 & & \qquad \mathsf{stop} : \langle \mathtt{data} \rangle \{B_{\mathtt{O}}^3, B_{\mathtt{I}}^3\}. \\
 & & \qquad \mathtt{M} \to \mathtt{W} : \mathsf{stop} \langle \rangle \{B_{\mathtt{O}}^4, B_{\mathtt{I}}^4\}.\mathtt{end} \}
\end{array}
$$

$$
\begin{array}{l}
B_{\mathtt{O}}^1 = \{x_{\mathtt{M}} = 0, x_{\mathtt{M}}\} \\
B_{\mathtt{I}}^1 = \{l \le x_{\mathtt{W}}, x_{\mathtt{W}}\} \\
B_{\mathtt{O}}^2 = \{x_{\mathtt{W}} \le w, x_{\mathtt{W}}\} \\
B_{\mathtt{I}}^2 = \{x_{\mathtt{M}} = 2l + w, _\} \\
B_{\mathtt{O}}^3 = \{x_{\mathtt{M}} = 2l + w, _\} \\
B_{\mathtt{I}}^3 = \{3l + w \le x_{\mathtt{R}}, x_{\mathtt{R}}\} \\
B_{\mathtt{O}}^4 = \{x_{\mathtt{M}} = 2l + w, x_{\mathtt{M}}\} \\
B_{\mathtt{I}}^4 = \{x_{\mathtt{W}} = 2l, x_{\mathtt{W}}\}
\end{array}
$$

3.2.2. Labelled Transition System for Timed Global Types

The LTS for *TG*s is defined over states of the form (ν, G) and labels as:

$$\ell ::= \mathtt{pq}!l\langle S \rangle \mid \mathtt{pq}?l\langle S \rangle \mid t$$

where $\mathtt{pq}!l\langle S \rangle$ is a send action (i.e., $\mathtt{p}$ sends $l\langle S \rangle$ to $\mathtt{q}$), $\mathtt{pq}?l\langle S \rangle$ is the dual receive action, and $t \in \mathbb{R}^{\geq 0}$ is a time action modelling time passing. We denote the set of labels by $\mathcal{L}$ and let

$$
\begin{array}{l}
\mathit{subj}(\mathtt{pq}!l\langle S \rangle) = \mathtt{p} \\
\mathit{subj}(\mathtt{qp}?l\langle S \rangle) = \mathtt{p} \\
\mathit{subj}(t) = \emptyset
\end{array}
$$

The syntax of G is extended with $\mathtt{p} \rightsquigarrow \mathtt{q} : \{l : \langle S \rangle \{A\}.G\}$ to model the state in which message $l\langle S \rangle$ has been sent by $\mathtt{p}$ but not yet received by $\mathtt{q}$ (as in [DY13]).

When defining the semantics for *TG*s, we need to prevent time steps that are too big and would make the clock constraint of some of the actions ready to be executed, or *ready actions*, unsatisfiable. The ready actions are those that have no causal relationship with actions that occur earlier, syntactically. The *TG* in (3.1) has two ready actions since the send actions of $\mathtt{p}$ and of $\mathtt{k}$ can happen in any order due to asynchrony; the time elapsing intervals that would invalidate either $\{x_{\mathtt{p}} \le 20\}$ or $\{x_{\mathtt{k}} < 10\}$ should be prevented.

$$\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{x_{\mathtt{p}} \le 20, _, \mathsf{true}, _\}.\mathtt{k} \to \mathtt{r} : \langle \mathsf{int} \rangle \{x_{\mathtt{k}} < 10, _, x_{\mathtt{r}} = 10, _\} \quad (3.1)$$

In general, a ready action can have multiple branches and hence be annotated with a set of constraints (e.g., $\{\delta_i\}_{i \in I}$). Below, function $\mathtt{ready}(G, D)$ returns the set

of constraints of all ready actions in G. D is a participants and is initially empty. It is used to keep track of the causal dependencies between actions. We write $\mathtt{ready}(G)$ for $\mathtt{ready}(G,\emptyset)$.

$$\mathtt{ready}(\mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{\delta_{Oi}, \lambda_{Oi}, \delta_{Ii}, \lambda_{Ii}\}.G_i\}_{i\in I}, D) = \begin{cases} \{\delta_{Oi}\}_{i\in I} \bigcup_{i\in I} \mathtt{ready}(G_i, D \cup \{\mathtt{p},\mathtt{q}\}) & \text{if } \mathtt{p} \notin D \\ \bigcup_{i\in I} \mathtt{ready}(G_i, D \cup \{\mathtt{p},\mathtt{q}\}) & \text{otherwise} \end{cases}$$

$$\mathtt{ready}(\mathtt{p} \rightsquigarrow \mathtt{q} : \{l : \langle S \rangle \{\delta_O, \lambda_O, \delta_I, \lambda_I\}.G\}, D) = \begin{cases} \{\delta_I\} \cup \mathtt{ready}(G, D \cup \{\mathtt{q}\}) & \text{if } \mathtt{q} \notin D \\ \mathtt{ready}(G, D \cup \{\mathtt{q}\}) & \text{otherwise} \end{cases}$$

$$\mathtt{ready}(\mu \mathbf{t}.G, D) = \mathtt{ready}(G, D)$$

$$\mathtt{ready}(\mathbf{t}, D) = \mathtt{ready}(\mathtt{end}, D) = \emptyset$$

We write $\nu \models^* \mathtt{ready}(G)$ when the constraints of all ready actions of G are still satisfiable in ν. Formally,

$$\nu \models^* \mathtt{ready}(G) \text{ iff } \forall \{\delta_i\}_{i\in I} \in \mathtt{ready}(G)\, \exists t \geq 0,\ j \in I,\ \nu + t \models \delta_j$$

By requiring the satisfiability of *some* $j \in I$ we allow that each action can be executed at any time allowed by its constraints, instead of the earliest possible time. In this way, the semantics of *TG*s specify the full range of correct behaviours.

The transition rules for global types are given in Figure 3.2. We assume the execution always begins with initial assignment ν_0. Rule $\lfloor$SELECT$\rfloor$ models the usual selection activity except that the clock constraint of the selected branch j is checked against the current assignment, i.e., $\nu \models \delta_O$, and the clock assignment ν' is updated with reset predicate λ_O. Rule $\lfloor$BRANCH$\rfloor$ is dual for selection. Rule $\lfloor$TIME$\rfloor$ shows time passing by incrementing all clocks; the clause in the premise prevents time steps that make the clock constraints of some ready action unsatisfiable.

Rules $\lfloor$ASYNC1$\rfloor$ and $\lfloor$ASYNC2$\rfloor$ model interactions that appear later (syntactically) but are not causally dependent on the first interaction. For instance, in (3.2) below

$$\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{x_{\mathtt{p}} < 20,\ _\ , \mathsf{true},\ _\ \}.\mathtt{r} \to \mathtt{k} : \langle \mathsf{int} \rangle \{x_{\mathtt{r}} < 10,\ _\ , \mathsf{true},\ _\ \} \qquad (3.2)$$

the second interaction could happen before the first one as $\mathtt{p}$ and $\mathtt{r}$ are not causally related; in this case (3.2) would move by $\lfloor$ASYNC1$\rfloor$ to

$$\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{x_{\mathtt{p}} < 20,\ _\ , \mathsf{true},\ _\ \}.\mathtt{r} \rightsquigarrow \mathtt{k} : \langle \mathsf{int} \rangle \{x_{\mathtt{r}} < 10,\ _\ , \mathsf{true},\ _\ \} \qquad (3.3)$$

$$\frac{j \in I \quad A_j = \{\delta_O, \lambda_O, \delta_I, \lambda_I\} \quad \nu \models \delta_O \quad \nu' = [\lambda_O \mapsto 0]\nu}{(\nu, \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{A_i\}.G_i\}_{i \in I}) \xrightarrow{\mathtt{pq}!l_j\langle S_j \rangle} (\nu', \mathtt{p} \rightsquigarrow \mathtt{q} : l_j \langle S_j \rangle \{A_j\}.G_j)} \lfloor \text{SELECT} \rfloor$$

$$\frac{\nu \models \delta_I \quad \nu' = [\lambda_I \mapsto 0]\nu}{(\nu, \mathtt{p} \rightsquigarrow \mathtt{q} : \{l : \langle S \rangle \{\delta_O, \lambda_O, \delta_I, \lambda_I\}.G\}) \xrightarrow{\mathtt{pq}?l\langle S \rangle} (\nu', G)} \lfloor \text{BRANCH} \rfloor$$

$$\frac{(\nu, G[\mu \mathbf{t}.G/\mathbf{t}]) \xrightarrow{\ell} (\nu', G')}{(\nu, \mu \mathbf{t}.G) \xrightarrow{\ell} (\nu', G')} \lfloor \text{REC} \rfloor$$

$$\frac{\forall k \in I \quad (\nu, G_k) \xrightarrow{\ell} (\nu', G'_k) \quad \mathtt{p}, \mathtt{q} \notin subj(\ell) \quad \ell \neq t}{(\nu, \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{A_i\}.G_i\}_{i \in I}) \xrightarrow{\ell} (\nu', \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{A_i\}.G'_i\}_{i \in I})} \lfloor \text{ASYNC1} \rfloor$$

$$\frac{(\nu, G) \xrightarrow{\ell} (\nu', G') \quad \mathtt{q} \notin subj(\ell)}{(\nu, \mathtt{p} \rightsquigarrow \mathtt{q} : \{l : \langle S \rangle \{A\}.G\}) \xrightarrow{\ell} (\nu', \mathtt{p} \rightsquigarrow \mathtt{q} : \{l : \langle S \rangle \{A\}.G'\})} \lfloor \text{ASYNC2} \rfloor$$

$$\frac{\nu + t \models^* \mathtt{ready}(G)}{(\nu, G) \xrightarrow{t} (\nu + t, G)} \lfloor \text{TIME} \rfloor$$

Figure 3.2.: Labelled Transition for Timed Global Types

3.3. Timed Local Types

Similarly, we extend local types with constraints on clocks, yielding *timed local session types* (*TL*s).

3.3.1. Syntax of Timed Local Types and Projection Definition

Definition 3.3.1 (Syntax of Local Types)**.** The syntax for sorts S and timed local types T is:

$$\begin{array}{lll} S & ::= & \langle G \rangle \mid (T, \delta) \mid \mathsf{bool} \mid \mathsf{nat} \mid \ldots \quad \text{(sort } (T, \delta) \text{ is for session delegation)} \\ T & ::= & \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I} \mid \mathtt{p}\&\{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I} \mid \mu \mathbf{t}.T \mid \mathbf{t} \mid \mathtt{end} \\ B & ::= & \{\delta, \lambda\} \end{array}$$

In T, we model interactions from a participant's local viewpoint. It is denoted either as *selection* types $\mathtt{p} \oplus \{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}$ or *branching* types $\mathtt{p}\&\{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}$. We write the projection of G onto $\mathtt{p} \in \mathcal{P}(G)$ as $G \downarrow_{\mathtt{p}}$.

The definition is standard except each time assertion $\{\delta_O, \lambda_O, \delta_I, \lambda_I\}$ is projected onto the sender/receiver by keeping only the output part $\{\delta_O, \lambda_O\}$ or the input part $\{\delta_I, \lambda_I\}$, e.g., if $G = \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{B_i, B_i'\}.G_i\}_{i \in I}$ then $G \downarrow_{\mathtt{p}} = \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{B_i\}.G_i \downarrow_{\mathtt{p}}\}_{i \in I}$ and $G \downarrow_{\mathtt{q}} = \mathtt{p}\&\{l_i : \langle S_i \rangle \{B_i'\}.G_i \downarrow_{\mathtt{q}}\}_{i \in I}$.

Definition 3.3.2 describes how to project algorithmically a *TG* onto its participants. In this chapter, we consider the global types that are projectable on all their participants.

Definition 3.3.2 (Projection)**.** Given G the *projection of G on a* $\mathtt{p} \in G$, written $G \downarrow_{\mathtt{p}}$ is recursively defined as follows assuming $\mathtt{p} \neq \mathtt{q}$:

(1) $\mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{B_{Oi}, B_{Ii}\}.G_i\}_{i \in I} \downarrow_{\mathtt{r}} = \begin{cases} \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{B_{Oi}\}.G_i \downarrow_{\mathtt{r}}\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p}\&\{l_i : \langle S_i \rangle \{B_{Ii}\}.G_i \downarrow_{\mathtt{r}}\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{q} \\ \sqcup_{i \in I} G_i \downarrow_{\mathtt{r}} & \text{otherwise} \end{cases}$

(2) $(\mu \mathbf{t}.G) \downarrow_{\mathtt{r}} = \begin{cases} \mu \mathbf{t}.(G \downarrow_{\mathtt{r}}) & \text{if } G \downarrow_{\mathtt{r}} \neq \mathbf{t} \\ \mathtt{end} & \text{otherwise} \end{cases}$

(3) $\mathbf{t} \downarrow_{\mathtt{r}} = \mathbf{t}$

(4) $\mathtt{end} \downarrow_{\mathtt{r}} = \mathtt{end}$

If no side condition applies then G is *not projectable* on $\mathtt{p}$. The time assertions are projected straightforwardly: $\{\delta_O, \lambda_O, \delta_I, \lambda_I\}$ is projected onto the sender side by keeping the output part $\{\delta_O, \lambda_O\}$ and onto the receiver side by keeping the input part $\{\delta_I, \lambda_I\}$. The rest of project definition is standard. Case (1) uses the merge operator $\sqcup$ [YVPH09, DY11] to ensure that if the locally observable behaviour of the projected local type is not independent of the chosen branch, then it is identifiable by $\mathtt{r}$ via a unique label. If the branches cannot be merged then the side condition of case (1) is not satisfied.

The merge operator $\sqcup$ [YVPH09, DY11] is defined below

$$\begin{aligned} &\mathtt{p}\&\{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I} \sqcup \mathtt{p}'\&\{l_j : \langle S_j' \rangle \{B_j'\}.T_j'\}_{j \in J} \stackrel{\text{def}}{=} \\ &\quad \mathtt{p}\&(\{l_k : \langle S_k \rangle \{B_k\}.T_k\}_{k \in I \setminus J} \cup \{l_k : \langle S_k' \rangle \{B_k'\}.T_k'\}_{k \in J \setminus I} \\ &\quad \cup \{l_k : \langle S_k \rangle \{B_k\}.T_k \sqcup T_k'\}_{k \in I \cap J}) \\ &\text{where for each } k \in I \cap J,\ \mathtt{p}_k = \mathtt{p}_k',\ S_k = S_k' \text{ and } B_k = B_k'. \end{aligned}$$

and ensures that either (a) the locally observable behaviour of the projected local type is either independent of the chosen branch (i.e., $G_i \downarrow_{\mathtt{r}} = G_j \downarrow_{\mathtt{r}}$ for all $i, j \in I$), or (b) the chosen branch is identifiable by $\mathtt{r}$ via a unique label. If the side condition of the merge operator is not satisfied then the two timed local types are not mergeable;

in this case, the side condition of case (1) of Definition 3.3.2 of projection is not satisfied.

Definition 3.3.3 (Projectable *TG*)**.** *We say that G is* projectable *if there exist* $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}(G)}$ *such that for all* $\mathtt{p}\in\mathcal{P}(G)$, $G\downarrow_{\mathtt{p}} = T_{\mathtt{p}}$.

In this chapter, we implicitly consider all G projectable.

Example 3.3.4 (Temperature calculation)**.** We show below the local timed type G for the protocol in §3.1 . We write _ for empty reset predicates.

$$
\begin{array}{lcl}
G\downarrow_{\mathtt{M}} & \stackrel{\text{def}}{=} & \mathtt{W}\oplus\langle\mathtt{task}\rangle\{B^1_{\mathtt{O}}\}.\\
 & & \mu\mathbf{t}.\,\mathtt{W}\&\langle\mathtt{data}\rangle\{B^2_{\mathtt{I}}\}.\\
 & & \mathtt{R}\oplus\{\mathsf{more}:\langle\mathtt{data}\rangle\{B^3_{\mathtt{O}}\}.\\
 & & \quad \mathtt{W}\oplus\mathsf{more}:\langle\mathtt{task}\rangle\{B^4_{\mathtt{O}}\}.\mathbf{t},\\
 & & \quad \mathsf{stop}:\langle\mathtt{data}\rangle\{B^3_{\mathtt{O}}\}.\\
 & & \quad \mathtt{W}\oplus\mathsf{stop}:\langle\rangle\{B^4_{\mathtt{O}}\}.\mathtt{end}\}\\
G\downarrow_{\mathtt{W}} & \stackrel{\text{def}}{=} & \mathtt{M}\&\langle\mathtt{task}\rangle\{B^1_{\mathtt{I}}\}.\\
 & & \mu\mathbf{t}.\,\mathtt{M}\oplus\langle\mathtt{data}\rangle\{B^2_{\mathtt{O}}\}.\\
 & & \mathtt{M}\&\{\mathsf{more}:\langle\mathtt{task}\rangle\{B^4_{\mathtt{I}}\}.\mathbf{t},\\
 & & \quad \mathsf{stop}:\langle\rangle\{B^4_{\mathtt{O}}\}.\mathtt{end}\}\\
G\downarrow_{\mathtt{R}} & \stackrel{\text{def}}{=} & \mu\mathbf{t}.\,\mathtt{M}\&\{\mathsf{more}:\langle\mathtt{data}\rangle\{B^3_{\mathtt{I}}\}.\mathbf{t},\\
 & & \quad \mathsf{stop}:\langle\mathtt{data}\rangle\{B^3_{\mathtt{O}}\}.\mathtt{end}\}
\end{array}
$$

$$
\begin{array}{l}
B^1_{\mathtt{O}} = \{x_{\mathtt{M}} = 0, x_{\mathtt{M}}\}\\
B^1_{\mathtt{I}} = \{l \le x_{\mathtt{W}}, x_{\mathtt{W}}\}\\
B^2_{\mathtt{O}} = \{x_{\mathtt{W}} \le w, x_{\mathtt{W}}\}\\
B^2_{\mathtt{I}} = \{x_{\mathtt{M}} = 2l + w,\ _\ \}\\
B^3_{\mathtt{O}} = \{x_{\mathtt{M}} = 2l + w,\ _\ \}\\
B^3_{\mathtt{I}} = \{3l + w \le x_{\mathtt{R}}, x_{\mathtt{R}}\}\\
B^4_{\mathtt{O}} = \{x_{\mathtt{M}} = 2l + w, x_{\mathtt{M}}\}\\
B^4_{\mathtt{I}} \stackrel{\text{def}}{=} \{x_{\mathtt{W}} = 2l, x_{\mathtt{W}}\}
\end{array}
$$

3.3.2. LTS for Timed Local Types

The LTS for *TLs* is defined on states (ν, T), labels $\mathcal{L}$ and by the rules in Figure 3.3, using the same sets of labels as for timed global types, where ν is a clock assignment with domain $x_{\mathtt{p}}$.

In rule $\lfloor$LTIME$\rfloor$, $\nu + t \models^* \mathtt{ready}(T)$ is defined as in §3.2 but relying on the function $\mathtt{ready}()$ on timed local types from Definition 3.3.5.

Definition 3.3.5 (Ready local clock constraints)**.**

$$
\begin{array}{lcl}
\mathtt{ready}(\mathtt{p}\oplus\{l_i:\langle S_i\rangle\{\delta_i,\lambda_i\}.T_i\}_{i\in I}) & = & \mathtt{ready}(\mathtt{p}\&\{l_i:\langle S_i\rangle\{\delta_i,\lambda_i\}.T_i\}_{i\in I}) = \{\delta_j\}_{i\in I}\\
\mathtt{ready}(\mu\mathbf{t}.T) & = & \mathtt{ready}(T)\\
\mathtt{ready}(\mathbf{t}) & = & \mathtt{ready}(\mathtt{end}) = \emptyset
\end{array}
$$

Given a set of participants $\{1,\ldots,n\}$, we define the configurations $s ::= (T_1,\ldots,T_n,$

$$\frac{j \in I \quad B_j = \{\delta, \lambda\} \quad \nu \models \delta \quad \nu' = [\lambda \mapsto 0]\nu}{(\nu, \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}) \xrightarrow{\mathtt{pq}!l_j\langle S_j \rangle} (\nu', T_j)} \quad \lfloor \textsc{LSel} \rfloor$$

$$\frac{j \in I \quad B_j = \{\delta, \lambda\} \quad \nu \models \delta \quad \nu' = [\lambda \mapsto 0]\nu}{(\nu, \mathtt{q} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \xrightarrow{\mathtt{qp}?l_j\langle S_j \rangle} (\nu', T_j)} \quad \lfloor \textsc{LBra} \rfloor$$

$$\frac{(\nu, T[\mu \mathbf{t}.T/\mathbf{t}]) \xrightarrow{\ell} (\nu', T')}{(\nu, \mu \mathbf{t}.T) \xrightarrow{\ell} (\nu', T')} \qquad \frac{\nu + t \models^* \mathtt{ready}(T)}{(\nu, T) \xrightarrow{t} (\nu + t, T)} \quad \lfloor \textsc{LRec/LTime} \rfloor$$

Figure 3.3.: Labelled Transition Rules for Local Timed Types

$\tilde{w}$) where $\tilde{w} ::= \{w_{ij}\}_{i \neq j \in \{1,\ldots,n\}}$ are unidirectional, possibly empty (denoted by ε), unbounded channels with elements of the form $l\langle S \rangle$. The LTS for configuration is defined as follows and ν is the overriding union (i.e., $\oplus_{i \in \{1,\ldots,n\}} \nu_i$) of the clock assignments ν_i of the participants.

$(\nu, (T_1, \ldots, T_n, \tilde{w})) \xrightarrow{\ell} (\nu', (T_1', \ldots, T_n', \tilde{w}'))$ iff:

$$\ell = \mathtt{pq}!l\langle S \rangle \Rightarrow (\nu_{\mathtt{p}}, T_{\mathtt{p}}) \xrightarrow{\ell} (\nu_{\mathtt{p}}', T_{\mathtt{p}}') \wedge w_{\mathtt{pq}}' = w_{\mathtt{pq}} \cdot l\langle S \rangle \wedge (ij \neq \mathtt{pq} \Rightarrow w_{ij} = w_{ij}' \wedge T_i = T_i')$$
$$\ell = \mathtt{pq}?l\langle S \rangle \Rightarrow (\nu_{\mathtt{q}}, T_{\mathtt{q}}) \xrightarrow{\ell} (\nu_{\mathtt{q}}', T_{\mathtt{q}}') \wedge l\langle S \rangle \cdot w_{\mathtt{pq}}' = w_{\mathtt{pq}} \wedge (ij \neq \mathtt{pq} \Rightarrow w_{ij} = w_{ij}' \wedge T_j = T_j')$$
$$\ell = t \Rightarrow \forall i \neq j \in \{1,\ldots,n\}.(\nu_i, T_i) \xrightarrow{\ell} (\nu_i + t, T_i) \wedge w_{ij} = w_{ij}' (\text{with } \mathtt{p}, \mathtt{q}, i, j \in \{1,\ldots,n\})$$

We denote by $\approx$ the standard weak bisimulation, and often omit initial assignments.

We prove that soundness and completeness are preserved through projection. Before showing the theorem, we present the following lemma.

Lemma 3.3.6. *Let G be a (projectable, see Definition 3.3.3) TG, and ν be the overriding union of the assignments $\nu_{\mathtt{p}}$ of all $\mathtt{p} \in \mathcal{P}(G)$, then for all $\mathtt{p} \in \mathcal{P}(G)$:*

$$\nu + t \models^* \mathtt{ready}(G) \quad \textit{iff} \quad \nu_{\mathtt{p}} + t \models^* \mathtt{ready}(G \downarrow_{\mathtt{p}})$$

Theorem 3.3.7 (Soundness and completeness of projection)**.** *Let G be a timed global type and $\{T_1, \ldots, T_n\} = \{G \downarrow_{\mathtt{p}}\}_{\mathtt{p} \in \mathcal{P}(G)}$ be the set of its projections; then $G \approx (T_1, \ldots, T_n, \tilde{\varepsilon})$.*

Proof. The proof of Theorem 3.3.7 proceeds as the proof of Theorem 3.1 [DY13]. First, we define (as in [DY13]) a behaviour-preserving subtyping relation $T < T'$ between timed local types which allows adding branches in branching types but not selection types. This subtyping is extended to configurations as follows:

$(\nu, \tilde{T}, \tilde{w}) < (\nu', \tilde{T}', \tilde{w}')$ if $\tilde{w} = \tilde{w}'$, $\nu = \nu'$ and $\forall \mathtt{p} \in \mathcal{P}$, $T_{\mathtt{p}} < T'_{\mathtt{p}}$ and we obtain the following property: if $(\nu, \tilde{T}, \tilde{w}) < (\nu', \tilde{T}', \tilde{w}')$ then $(\nu, \tilde{T}, \tilde{w}) \approx (\nu', \tilde{T}', \tilde{w}')$.

Second, to match global and local types step by step, as in [DY13], we extend projection to intermediary steps and define projected configuration of a global type G, written $[[G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq} \in \mathcal{P}(G)}$ to take into account the content of channels by adding into the channels the messages sent in intermediary states i.e.,

$$[[\mathtt{p}' \rightsquigarrow \mathtt{q}' : \{l : \langle S \rangle \{A\}.G\}]]\{w_{\mathtt{pq}}\}_{\mathtt{pq} \in \mathcal{P}(G)} = [[G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq} \in \mathcal{P}(G)}[w_{\mathtt{p'q'}} = w_{\mathtt{p'q'}} \cdot l\langle S \rangle]$$

Third, as in [DY13] we prove the following step equivalence which directly yields the theorem: if $[[G]] < (\nu, \tilde{T}, \tilde{w})$ then $G \xrightarrow{\ell} G' \Leftrightarrow (\nu, \tilde{T}, \tilde{w}) \xrightarrow{\ell} (\nu', \tilde{T}', \tilde{w}')$ and $[[G']] < (\nu', \tilde{T}', \tilde{w}')$.

The step equivalence is proved by induction on the structure of the transition of G. For soundness we assume $(\nu, G) \xrightarrow{\ell} (\nu', G')$ and show that the corresponding configuration $(\nu, \tilde{T}, \tilde{w})$ can then make a step ℓ into $(\nu'', \tilde{T}'', \tilde{w}'') < (\nu', \tilde{T}', \tilde{w}')$ where $(\nu', \tilde{T}', \tilde{w}')$ is the projection $[[G']]\{w_{\mathtt{pq}}\}_{\mathtt{pq} \in \mathcal{P}(G)}$ of G'. All the cases are identical to [DY13], observing that, by projection, if the clock constraints of G are satisfied then also the constraints in the corresponding configuration are satisfied since both cock constraints and reset predicates of G are the same as those of its projections. The only step that is not treated in [DY13] is the time transition by $\lfloor$TIME$\rfloor$. This cDese follows by Lemma 3.3.6. In fact, if $(\nu, G) \xrightarrow{t} (\nu', G')$ then the premise of $\lfloor$TIME$\rfloor$ – $\nu + t \models^* \mathtt{ready}(G)$ – is satisfied. By Lemma 3.3.6, $\nu + t \models^* \mathtt{ready}(G)$ yields $\nu + t \models^* \mathtt{ready}(T_i)$ for all projections T_i hence the premise of $\lfloor$LTIME$\rfloor$ for all T_i is satisfied and the configuration can make a time step t.

Completeness proceeds similarly to [DY13] except the step $\lfloor$TIME$\rfloor$ which also follows by Lemma 3.3.6. □

3.4. Multiparty Session Processes with Delays

In this section, we present the theory of timed multiparty session calculus. We define the semantics of multiparty session processes with delay primitives and show the process reduction rules. The typing system of timed session calculus is defined for verifying the program. We also extend to runtime processes that include runtime queues and define the corresponding reduction rules and proof rules. Finally, we define and prove the theorems of type preservation and timed error-freedom.

3.4.1. Syntax of Timed Session Calculus

We model processes using a timed extension of the asynchronous session calculus[CDCYP15]. The syntax of the calculus with delays is presented below.

Definition 3.4.1 (Syntax of Timed Session Calculus)**.**

$P ::=$	$\overline{u}[\mathtt{n}](y).P$	*Request*
$\mid$	$u[\mathtt{i}](y).P$	*Accept*
$\mid$	$c[\mathtt{p}] \triangleleft l\langle e\rangle; P$	*Select*
$\mid$	$c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I}$	*Branching*
$\mid$	$c[\mathtt{p}] \triangleleft \langle\langle c', \mathtt{v}\rangle\rangle; P$	*Delegate*
$\mid$	$c[\mathtt{p}] \triangleright ((y)).P$	*Session receive*
$\mid$	$\mathtt{delay}(t).P$	*Delay*
$\mid$	$\mathtt{if}\, e\, \mathtt{then}\, P\, \mathtt{else}\, Q$	*Conditional*
$\mid$	$P \mid Q$	*Parallel*
$\mid$	0	*Inaction*
$\mid$	$\mu X.P$	*Recursion*
$\mid$	X	*Variable*
$\mid$	$(\nu a)P$	*Hide Shared*
$\mid$	$(\nu s)P$	*Hide Session*
$\mid$	$s : h$	*Queue*
$h ::=$	$\emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, m)$	*(queue content)*
$m ::=$	$l\langle v\rangle \mid s[\mathtt{p}]$	*(messages)*
$c ::=$	$s \mid y$	*(session names/variables)*
$u ::=$	$a \mid u$	*(shared names/variables)*
$e ::=$	$v \mid \neg e \mid e' \,\mathtt{op}\, e'$	*(expressions)*
$v ::=$	$c \mid u \mid \mathsf{true} \mid \ldots$	*(values)*

Process $\overline{u}[\mathtt{n}](y).P$ requests, along u, to start a new session y (with n participants) as participant `1` and continues as P. Process $u[\mathtt{i}](y).P$ accepts an invitation to engage in a new session as participant `i`. Select $c[\mathtt{p}] \triangleleft l\langle e\rangle; P$ sends message $l\langle e\rangle$ to participant `p` in session c and continues as P. Branching is dual. In e we let `op` denote predefined binary operators over expressions. Request and accept bind y in P; branching binds z_i in P_i. We introduce a new primitive $\mathtt{delay}(t).P$ that executes P after waiting exactly t units of time. Note that t is a constant (as in [BY07, LZ05, LP12]). We often omit 0, and the label in a singleton selection or

$$
\begin{array}{rcll}
\overline{a}[\mathtt{n}](y).P_1 \mid \prod_{i\in\{2,..,n\}} a[\mathtt{i}](y).P_i & \longrightarrow & (\nu s)(\prod_{i\in\{1,..,n\}} P_i[s[\mathtt{i}]/y] \mid s:\emptyset) & (s \notin \mathsf{fn}P_i) \\
 & & & \lfloor \textsc{Link} \rfloor \\
s[\mathtt{q}] \triangleleft l\langle e\rangle; P \mid s:h & \longrightarrow & P \mid s : h \cdot (\mathtt{p},\mathtt{q}, l\langle v\rangle) \qquad (e \downarrow v) & \lfloor \textsc{Sel} \rfloor \\
s[\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in J} \mid s:(\mathtt{p},\mathtt{q},l_j\langle v\rangle)\cdot h & \longrightarrow & P_j[v/z_j] \mid s:h \qquad (j\in J) & \lfloor \textsc{Bra} \rfloor \\
s[\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P \mid s:h & \longrightarrow & P \mid s : h \cdot (\mathtt{p},\mathtt{q},s'[\mathtt{p}']) & \lfloor \textsc{Del} \rfloor \\
s[\mathtt{q}] \triangleright ((y)).P \mid s:(\mathtt{p},\mathtt{q},s'[\mathtt{p}'])\cdot h & \longrightarrow & P[s'[\mathtt{p}']/y] \mid s:h & \lfloor \textsc{SRec} \rfloor \\
\prod_{j\in J} s_j : h_j \mid \prod_{i\in I} \mathtt{delay}(t).P_i & \longrightarrow & \prod_{j\in J} s_j : h_j \mid \prod_{i\in I} P_i & \lfloor \textsc{Delay} \rfloor \\
P \longrightarrow P' \quad P \not\equiv (\nu\tilde{n})(\mathtt{delay}(t).R \mid Q') & \textit{imply} & P \mid Q \longrightarrow P' \mid Q & \lfloor \textsc{Com} \rfloor
\end{array}
$$

Figure 3.4.: Reduction for processes

branching. We denote with $\mathsf{fn}P$ the set of free variables and names of P. We define *programs* as processes that have not yet engaged in any session, namely that have no queues, no session name hiding, and no free session names/variables.

The reduction rules are given in Figure 3.4. In $\lfloor \textsc{Sel} \rfloor$ we write $e \downarrow v$ when expression e evaluates to value v. Rule $\lfloor \textsc{Delay} \rfloor$ allows a delay to elapse simultaneously for a number of parallel processes. Note that t must be the same for all P_i and that we can use the structural equivalence rules to conveniently arrange delays. Rule $\lfloor \textsc{Com} \rfloor$ (n stands for s or a) enables part of the system to reduce as long as the reduction is not by $\lfloor \textsc{Delay} \rfloor$ on P. The other rules are standard.

Example 3.4.2 (Temperature calculation)**.** Process $P_{\mathtt{M}}$ is a possible implementation of participant `M` of the protocol in Example 3.3.4; note the correspondence between $P_{\mathtt{M}}$ and $G\downarrow_{\mathtt{M}}$ from Example 3.3.4. Assuming that at least one task is needed in each session, we let `task()` be a local function returning the next task, and `more_tasks()` return `true` if more tasks have to be submitted and `false` otherwise.

$$
\begin{array}{rcl}
P_{\mathtt{M}} & \stackrel{\text{def}}{=} & s[\mathtt{M}][\mathtt{W}] \triangleleft \langle \mathtt{task}() \rangle; \mu X.\mathtt{delay}(2l+w).\ s[\mathtt{M}][\mathtt{W}] \triangleright (y); \mathtt{if}\ \mathtt{more_tasks}() \\
 & & \mathtt{then}\ s[\mathtt{M}][\mathtt{R}] \triangleleft \mathsf{more}\langle y\rangle; s[\mathtt{M}][\mathtt{W}] \triangleleft \mathsf{more}\langle \mathtt{task}()\rangle; X \\
 & & \mathtt{else}\ s[\mathtt{M}][\mathtt{R}] \triangleleft \mathsf{stop}\langle y\rangle; s[\mathtt{M}][\mathtt{W}] \triangleleft \mathsf{stop}\langle\rangle; \mathtt{end}
\end{array}
$$

3.4.2. Structural Equivalence and Proof Rules

Structural equivalence for processes is the least equivalence relation that satisfies the standard rules from [CDCYP15] plus the following rules for delays:

$$
\begin{array}{l}
P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
(\nu n)P \mid Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin \mathsf{fn}Q \\
(\nu n)(\nu n')P \equiv (\nu n')(\nu n)P \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \\
\mu X.P \equiv P[\mu X.P/X] \\
(\nu s)s : \emptyset = 0 \\
s : (\mathtt{p},\mathtt{q},\mathtt{n}) \cdot (\mathtt{p}',\mathtt{q}',\mathtt{n}') \cdot h \equiv s : (\mathtt{p}',\mathtt{q}',\mathtt{n}') \cdot (\mathtt{p},\mathtt{q},\mathtt{n}) \cdot h \qquad \text{if } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'
\end{array}
$$

$$
\begin{array}{l}
\mathtt{delay}(t+t').P \equiv \mathtt{delay}(t).\mathtt{delay}(t').P \\
\mathtt{delay}(0).P \equiv P \\
\mathtt{delay}(t).(\nu a)P \equiv (\nu a)\mathtt{delay}(t).P \\
\mathtt{delay}(t).(P \mid Q) \equiv \mathtt{delay}(t).P \mid \mathtt{delay}(t).Q
\end{array}
$$

The rules at the top are standard. We just recall that rule $(\nu s)s : \emptyset = 0$ is for garbage collection of queues of terminated sessions, and the rule for queues in the last line (on the top) is for permuting messages that are causally unrelated (i.e., can be received in any order due to asynchrony). At the bottom, $\mathtt{delay}(t).\mathtt{delay}(t').P \equiv \mathtt{delay}(t+t').P$ breaks a delay into smaller intervals, $\mathtt{delay}(0).P \equiv P$ always allows time to elapse for idle processes, and the rules in the second line distribute a delay within a shared name restriction and in parallel compositions.

Proof rules. We validate programs against specifications based on *TL*s, using judgements of the form $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash e : S$ defined on the following environments:

$$\Gamma ::= \emptyset \mid \Gamma, u : S \mid \Gamma, X : \Delta \qquad\qquad \Delta ::= \emptyset \mid \Delta, c : (\nu, T)$$

The *type environment* Γ maps shared variables/names to sorts and process variables to their types, and the *session environment* Δ holds information on the ongoing sessions, e.g., $\Delta(s[\mathtt{p}]) = (\nu, T)$ when the process being validated is acting as $\mathtt{p}$ in session s specified by T; ν is a 'virtual' clock assignment built during the validation.

Resets can generate infinite time scenarios in recursive protocols. To ensure sound typing we introduce a condition, *infinite satisfiability*, that guarantees

regularity across different instances of a recursion.

Definition 3.4.3 (Infinitely satisfiable)**.** G is infinitely satisfiable if either: (1) constraints in recursion bodies have no resets, no equalities nor upper bounds (i.e., $x < c$ or $x \leq c$), or (2) all participants reset at each iteration.

In the rest of this section, we assume that *TG*s are infinitely satisfiable. As usual, in the validation of P we check $\Gamma \vdash P' \triangleright \Delta$ where P' is obtained by unfolding once all recursions $\mu X.P''$ occurring in P. This ensures that both the first instance of a recursion and the successive ones (all similar by infinite satisfiability) satisfy the specification.

We show in Figure 3.5 proof rules for programs. Rule $\lfloor$VREQ$\rfloor$ for session request adds a new instance of session for participant `1` to Δ in the premise. The newly instantiated session is associated with an initial assignment ν_0 for the clock of participant `1`. Rule $\lfloor$VACC$\rfloor$ for session accept is similar but initiates a new session for participant `i`. Rule $\lfloor$VBRA$\rfloor$ for branching processes checks all the branches in I. If the received message is a session (i.e., $S_i = (T_d, \delta_d)$) a new assignment $z_i : (\nu_d, T_d)$ is added to Δ in the premise. This can be any assignment such that $\nu_d \models \delta_d$. Rule $\lfloor$VSEL$\rfloor$ for selection processes checks that the clock constraint δ_j of the selected branch j satisfies the current assignment ν, and updates, in the session environment of the premise, the clock assignment as prescribed by λ_j. Rule $\lfloor$VDEL$\rfloor$ for delegation requires δ_d to be satisfied in the assignment ν_d of the delegated session, which is removed from the premise. Rule $\lfloor$VTIME$\rfloor$ increments the clock assignments of all sessions in Δ. Rule $\lfloor$VEND$\rfloor$ validates $\mathbf{0}$ if there are no more actions prescribed by Δ. Rule $\lfloor$VDEF$\rfloor$ extends Γ with the assignment for process variable X. Rules $\lfloor$VPAR$\rfloor$ and $\lfloor$VCOND$\rfloor$ are standard. Rule $\lfloor$VCALL$\rfloor$ validates, as usual, recursive call X against $\Gamma(X)$ (and possibly some terminated sessions Δ').

3.4.3. Extension to Runtime Processes

In this subsection, we extend the typing system of programs to type run-time processes in which sessions have been initiated and queues may be non-empty. This extension is necessary for subject reduction.

We first extend timed local types to include run-time queues.

$$\frac{\Gamma, u : G \vdash P \triangleright \Delta, y[1] : (\nu_0, \restriction G1) \quad dom(\nu_0) = \{x_1\}}{\Gamma, u : G \vdash \overline{u}[\mathtt{n}](y).P \triangleright \Delta} \quad \lfloor \textsc{vReq} \rfloor$$

$$\frac{\Gamma, u : G \vdash P \triangleright \Delta, y[\mathtt{i}] : (\nu_0, \restriction G\mathtt{i}) \quad dom(\nu_0) = \{x_\mathtt{i}\} \quad \mathtt{i} \neq 1}{\Gamma, u : G \vdash u[\mathtt{i}](y).P \triangleright \Delta} \quad \lfloor \textsc{vAcc} \rfloor$$

$$\frac{j \in I \quad \Gamma \vdash e : S_j \quad \Gamma \vdash P \triangleright \Delta, c : ([\lambda_j \mapsto 0]\nu, T_j) \quad \nu \models \delta_j}{\Gamma \vdash c[\mathtt{p}] \triangleleft l_j\langle e \rangle; P \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})} \quad \lfloor \textsc{vSel} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : ([\lambda \mapsto 0]\nu, T') \qquad \nu \models \delta \qquad \nu_d \models \delta_d}{\Gamma \vdash c[\mathtt{p}] \triangleleft \langle\langle c' \rangle\rangle; P \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T'), c' : (\nu_d, T_d)} \quad \lfloor \textsc{vDel} \rfloor$$

$$\frac{\forall i \in I \quad \Gamma, z_i : S_i \vdash P_i \triangleright \Delta, c : ([\lambda_i \mapsto 0]\nu, T_i) \quad \nu \models \delta_i}{\Gamma \vdash \ c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \triangleright \Delta, c : (\nu, \mathtt{p} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})} \quad \lfloor \textsc{vBra} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : ([\lambda \mapsto 0]\nu, T'), y : (\nu_d, T_d) \quad \nu \models \delta \quad \nu_d \models \delta_d}{\Gamma \vdash \ c[\mathtt{p}] \triangleright ((y)).P \triangleright \Delta, c : (\nu, \mathtt{p} \& \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T')} \quad \lfloor \textsc{vSrec} \rfloor$$

$$\frac{dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \quad \Gamma \vdash P_i \triangleright \Delta_i \quad i \in \{1,2\}}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \quad \lfloor \textsc{vPar} \rfloor$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P_i \triangleright \Delta \quad i \in \{1,2\}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \triangleright \Delta} \quad \lfloor \textsc{vCond} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \{c_i : (\nu_i + t, T_i)\}_{i \in I}}{\Gamma \vdash \mathtt{delay}(t).P \triangleright \{c_i : (\nu_i, T_i)\}_{i \in I}} \quad \lfloor \textsc{vTime} \rfloor$$

$$\frac{\forall c \in dom(\Delta) \quad \Delta(c) = (\nu, \mathtt{end})}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \quad \lfloor \textsc{vEnd} \rfloor$$

$$\frac{\Gamma, X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.P \triangleright \Delta} \quad \lfloor \textsc{vDef} \rfloor$$

$$\frac{\forall c \in dom(\Delta') \quad \Delta'(c) = (\nu, \mathtt{end})}{\Gamma, X : \Delta \vdash X \triangleright \Delta, \Delta'} \quad \lfloor \textsc{vCall} \rfloor$$

Figure 3.5.: Proof Rules for Programs

$$
\begin{array}{lcll}
M & ::= & \langle \mathtt{p}, U \rangle \mid M;M & \text{Message Types} \\
\mathcal{T} & ::= & M \mid T \mid M;T & \text{Generalised Types} \\
\mathcal{S} & ::= & M \mid (\nu, \mathcal{T}) & \text{Generalised Specifications} \\
U & ::= & l\langle S \rangle &
\end{array}
$$

The equivalence relation $\equiv$ for generalised specifications is as follows:

$$
\begin{array}{l}
\langle \mathtt{p}, U \rangle; \langle \mathtt{p}', U' \rangle; M \equiv \langle \mathtt{p}', U' \rangle; \langle \mathtt{p}, U \rangle; M \quad \text{if } \mathtt{p} \neq \mathtt{p}' \\
M \equiv M' \text{ implies } M; \mathcal{T} \equiv M'; \mathcal{T} \\
\mathcal{T} \equiv \mathcal{T}' \text{ implies } (\nu, \mathcal{T}) \equiv (\nu, \mathcal{T}')
\end{array}
$$

We will use (generalised) session environments mapping $s[\mathtt{p}]$ to generalised specifications. A session environment can be combined with another session environments containing only message types as follows:

$$
\Delta; \{s[\mathtt{p}] : M\} = \begin{cases} \Delta', s[\mathtt{p}] : M'; M & \text{if } \Delta = \Delta', s[\mathtt{p}] : M' \\ \Delta, s[\mathtt{p}] : M & \text{otherwise} \end{cases}
$$

Figure 3.6 generalises the proof rules for programs in Figure 3.5 to processes. The proof rules for queues are on the top of Figure 3.6. The generalised proof rules rely on the following definitions:

- $\Delta \equiv \Delta'$ if $c : \mathcal{S} \in \Delta$ and $\mathcal{S} \neq (\nu, \mathtt{end})$ imply $c : \mathcal{S}' \in \Delta'$ with $\mathcal{S} \equiv \mathcal{S}'$ and vice versa.
- $\Delta * \Delta' = \Delta \setminus dom(\Delta') \cup \Delta' \setminus dom(\Delta) \cup \{y : \mathcal{S} * \mathcal{S}' \mid y : \mathcal{S} \in \Delta \wedge y : \mathcal{S}' \in \Delta'\}$
- $\mathcal{S} * \mathcal{S}' = \begin{cases} \mathcal{S}; \mathcal{S}' & \text{if } \mathcal{S} \text{ is a message type} \\ \mathcal{S}'; \mathcal{S} & \text{if } \mathcal{S}' \text{ is a message type} \\ \text{undefined} & \text{otherwise} \end{cases}$

Note on coherence of $\mathcal{S}$ The projection of the generalised local timed type $\upharpoonright T\mathtt{q}$ is a straightforward extension of the one in [CDCYP15] (where constraints are just projected as they are). The duality relation of local timed types, written $T \bowtie T'$ is also derived from [CDCYP15] (where the relation holds for any clock constraints and reset predicate). $(\nu, T) \bowtie (\nu', T')$ if $T \bowtie T'$. We say that $\Delta = \{s[\mathtt{p}_i] : \mathcal{S}_i\}_{i \in I}$ is coherent if $s[\mathtt{p}] : (\nu, T) \in \Delta$ and $\upharpoonright T\mathtt{q} \neq \mathtt{end}$ imply $s[\mathtt{q}] : T' \in \Delta$ and $\upharpoonright T\mathtt{q} \bowtie \upharpoonright T'\mathtt{p}$.

Reduction rules for session environments are in Figure 3.7.

$$\frac{}{\Gamma \vdash s : \emptyset \triangleright \emptyset} \qquad \frac{\Gamma \vdash s : h \triangleright \Delta \qquad \Gamma \vdash v : S}{\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \triangleright \Delta; \{s[\mathtt{p}] : \langle \mathtt{q}, l\langle S \rangle \rangle\}} \quad \lfloor \textsc{QInit} \rfloor / \lfloor \textsc{QSend} \rfloor$$

$$\frac{\Gamma \vdash s : h \triangleright \Delta}{\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \triangleright \Delta; s'[\mathtt{p}'] : (\nu, T); \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T, \delta) \rangle \rangle\}} \quad \lfloor \textsc{QDel} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Delta \equiv \Delta'}{\Gamma \vdash P \triangleright \Delta'} \qquad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta * \Delta'} \quad \lfloor \textsc{Equiv} \rfloor / \lfloor \textsc{GPar} \rfloor$$

$$\frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a) P \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta'; \{s[\mathtt{p}_i] : \mathcal{S}_i\}_{i \in I} \quad \{s[\mathtt{p}_i] : \mathcal{S}_i\}_{i \in I} \text{ coherent}}{\Gamma \vdash (\nu s) P \triangleright \Delta'}$$

$\lfloor \textsc{GNRes} / \textsc{GSRes} \rfloor$

Figure 3.6.: Proof Rules for Processes (the rules on the top are for queues)

$$\frac{i \in J \quad \delta_i \models \nu \quad \text{if } \lambda_i = \emptyset \text{ then } \nu' = \nu \text{ else } \nu' = 0}{\{s[\mathtt{p}] : (\nu, M; \mathtt{q} \oplus \{l_j : \langle S_j \rangle \{\delta_j, \lambda_j\}.T_j\}_{j \in J})\} \longrightarrow \{s[\mathtt{p}] : (\nu', M; \langle \mathtt{q}, l_i\langle S_i \rangle \rangle; T_i)\}} \quad \text{[RSend]}$$

$$\frac{i \in J \quad \delta_i \models \nu \quad \text{if } \lambda_i = \emptyset \text{ then } \nu' = \nu \text{ else } \nu' = 0}{\begin{array}{r}\{s[\mathtt{p}] : (\nu, \langle \mathtt{q}, l_i\langle S_i \rangle \rangle; \mathcal{T}); s[\mathtt{q}] : (\nu'', \mathtt{p} \& \{l_j : \langle S_j \rangle \{\delta_j, \lambda_i\}.T_j\}_{j \in J})\} \longrightarrow \\ \{s[\mathtt{p}] : (\nu, \mathcal{T}); s[\mathtt{q}] : (\nu', T_i)\}\end{array}} \quad \text{[RRec]}$$

$$\frac{\forall i \in I, j \in J,\ \nu_{ij} + t \models \mathtt{ready}(\mathcal{T}_{ij})}{\{\{s_i[\mathtt{p}_j] : (\nu_{ij}, \mathcal{T}_{ij})\}_{j \in J_i}\}_{i \in I} \longrightarrow \{\{s_i[\mathtt{p}_j] : (\nu_{ij} + t, \mathcal{T}_{ij})\}_{j \in J_i}\}_{i \in I}} \quad \text{[RTime]}$$

$$\frac{\Delta \longrightarrow \Delta' \quad \Delta' \ \mathtt{not\ after}\ \Delta}{\Delta \cup \Delta'' \longrightarrow \Delta' \cup \Delta''} \quad \text{[RPar]}$$

$\Delta' \ \mathtt{not\ after}\ \Delta$ if $\forall s[\mathtt{p}] \in Dom(\Delta)$, $\Delta(s[\mathtt{p}]) = (\nu, \mathcal{S})$ implies
$\Delta'(s[\mathtt{p}])$ is either $(\nu, \mathcal{S}')$ or $(0, \mathcal{S}')$ for some $\mathcal{S}'$

Figure 3.7.: Reduction Rules for Specifications

3.5. Type Preservation of Timed Processes

The proof rules in Figure 3.5 ensure type preservation under structural equivalence, which we prove below after giving an auxiliary lemma.

Lemma 3.5.1. *If* $\Delta = \Delta_P * \Delta_Q$ *and* $dom(\Delta_P) \cap dom(\Delta_Q) = \emptyset$*, then* $\Delta + t = (\Delta_P + t) * (\Delta_Q + t)$ *and* $dom(\Delta_P + t) \cap dom(\Delta_Q + t) = \emptyset$*.*

Before proving the theorem of type preservation under equivalence and reduction, we need auxiliary lemmas.

Lemma 3.5.2 (Inversion Lemma)**.** *The following holds (by induction on derivations):*

1. *If* $\Gamma \vdash P \mid Q \triangleright \Delta$ *then* $\Delta = \Delta_1 * \Delta_2$*, and* $\Gamma \vdash P \triangleright \Delta_1$ *and* $\Gamma \vdash Q \triangleright \Delta_2$*.*
2. *If* $\Gamma \vdash c[\mathrm{p}] \triangleleft l_i\langle e\rangle; P \triangleright \Delta$ *then*
   a) $\Gamma \vdash e : S_j$*,*
   b) $i \in J$*,*
   c) $\Delta = \Delta', c : (\nu, \mathrm{p} \oplus \{l_j : \langle S_j\rangle\{\delta_j, \lambda_j\}.T_j\}_{j\in J})$*,*
   d) $\delta_i \models \nu$*,*
   e) $\Gamma \vdash P \triangleright \Delta', c : (\nu', T_i)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise.*
3. *If* $\Gamma \vdash c[\mathrm{p}] \triangleleft \langle\langle c'\rangle\rangle; P \triangleright \Delta$ *then*
   a) $\Delta = \Delta', c : (\nu, \mathrm{p} \oplus \langle(T_d, \delta_d)\rangle\{\delta, \lambda\}.T), c' : (\nu_d, T_d)$*,*
   b) $\delta \models \nu$*,*
   c) $\delta_d \models \nu_d$*,*
   d) $\Gamma \vdash P \triangleright \Delta', c : (\nu', T)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise.*
4. *If* $\Gamma \vdash c[\mathrm{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \triangleright \Delta$ *then*
   a) $\Delta = \Delta', c : (\nu, \mathrm{p}\&\{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i\in I})$*,*
   b) $\forall i \in I$*,* $\delta_i \models \nu$*,*
   c) $\forall i \in I\ \Gamma \vdash P_i[v/z_i] \triangleright \Delta', c : (\nu', T_i)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise.*
5. *If* $\Gamma \vdash c[\mathrm{p}] \triangleright ((y)).P \triangleright \Delta$ *then*
   a) $\Delta = \Delta', c : (\nu, \mathrm{p}\&\langle(T_d, \delta_d)\rangle\{\delta, \lambda\}.T)$*,*

b) $\delta_i \models \nu$,

c) $\delta_d \models \nu_d$,

d) $\Gamma \vdash P \triangleright \Delta', c : (\nu', T), y : (\nu_d, T_d)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise, and* $\nu_d \models \delta_d$.

6. *If* $\Gamma \vdash \texttt{delay}(t).P \triangleright \Delta$ *then*

   a) $\Delta = \{c_i : (\nu_i, T_i)\}$,

   b) $\Gamma \vdash P \triangleright \{c_i : (\nu_i + t, T_i)\}$

7. *If* $\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \triangleright \Delta$ *then*

   a) $\Delta = \Delta'; \{s : \langle \mathtt{q}, l\langle S \rangle \rangle\}$,

   b) $\Gamma \vdash s : h \triangleright \Delta'$

8. *If* $\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \triangleright \Delta$ *then*

   a) $\Delta = \Delta'; \{s : \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle\}, s'[\mathtt{p}'] : (\nu_d, T_d)$,

   b) $\Gamma \vdash s : h \triangleright \Delta'$

Lemma 3.5.3 (Satisfiability upon validation)**.** *If* $\Gamma \vdash P \triangleright \Delta$ *with* $\Delta = \{c_i : (\nu_i + t, T_i)\}_{i \in I}$ *for some* $t \geq 0$ *then* $\forall i \in I$, $\nu_i + t \models^* \texttt{ready}(T_i)$.

Proof. The proof is my induction on the proof tree of P, with a case analysis of the final rule. We only show the case for selection. The cases for delegation and branching proceed similarly to the case for selection. The remaining cases are straightforward (e.g., for parallel, delay and recursion they hold by inductive hypothesis, for end and recursive call $\texttt{ready}()$ is trivially satisfiable). If the last rule is $\lfloor \text{VSEL} \rfloor$, $\Gamma \vdash c[\mathtt{p}] \triangleleft l_k\langle e \rangle; P \triangleright \Delta$. By Lemma 3.5.2 (2))

$$\Delta = \Delta', c : (\nu + t, \mathtt{p} \oplus \{l_j : \langle S_j \rangle \{\delta_j, \lambda_j\}.T_j\}_{j \in J}) \quad k \in J \tag{3.4}$$

and

$$\delta_k = \nu + t \tag{3.5}$$

By Definition 3.3.5

$$\texttt{ready}(\mathtt{p} \oplus \{l_j : \langle S_j \rangle \{\delta_j, \lambda_j\}.T_j\}_{j \in J}) = \{\delta_j\}_{j \in J} \tag{3.6}$$

By Definition 3.3.5 it is sufficient that one of δ_j is satisfiable for $\{\delta_j\}_{j \in J}$ to be satisfiable. Hence by (3.5), (3.6) is satisfiable after t in ν, i.e., $\nu + t \models^* \{\delta_j\}_{j \in J}$. The thesis follows by induction. □

Theorem 3.5.4 (Type preservation under equivalence)**.** *If* $\Gamma \vdash P \triangleright \Delta$ *and* $P \equiv P'$ *then* $\Gamma \vdash P' \triangleright \Delta$.

Proof. We only show the cases for delay processes. The other cases are as in [CDCYP15].

The case for $\texttt{delay}(t).0 \equiv 0$ follows from the fact that both processes can always be validated (for all t): the first by one application of rule $\lfloor$VTIME$\rfloor$ and one of $\lfloor$END$\rfloor$, and the second one application of rule $\lfloor$END$\rfloor$.

The case for $\texttt{delay}(t+t').P \equiv \texttt{delay}(t).\texttt{delay}(t').P$ proceeds as follows. If $\Gamma \vdash \texttt{delay}(t).\texttt{delay}(t').P \triangleright \Delta$ then by Lemma 3.5.2 (6) we obtain $\Gamma \vdash P \triangleright \Delta + t + t'$ which by associativity of sum of non-negative reals yields

$$\Gamma \vdash P \triangleright \Delta + (t + t') \tag{3.7}$$

By proof rule $\lfloor$VTIME$\rfloor$ on (3.7) we obtain $\Gamma \vdash \texttt{delay}(t+t').P \triangleright \Delta$ as required. If, reversely $\Gamma \vdash \texttt{delay}(t+t').P \triangleright \Delta$ by applying once Lemma 3.5.2 (6) we obtain $\Gamma \vdash P \triangleright \Delta + (t + t')$ which by associativity of $+$ yields

$$\Gamma \vdash P \triangleright \Delta + t + t' \tag{3.8}$$

By applying twice rule $\lfloor$VTIME$\rfloor$ on (3.8) we obtain $\Gamma \vdash \texttt{delay}(t).\texttt{delay}(t').P \triangleright \Delta$.

The case for $\texttt{delay}(t).(\nu a)P \equiv (\nu a)\texttt{delay}(t).P$ proceeds as follows. We first consider the case $\Gamma \vdash \texttt{delay}(t).(\nu a)P \triangleright \Delta$. By premise of $\lfloor$VTIME$\rfloor$,

$$\Gamma \vdash (\nu a)P \triangleright \Delta + t \tag{3.9}$$

and by premise of $\lfloor$GNRES$\rfloor$ on (3.9)

$$\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta + t \tag{3.10}$$

We apply (3.10) as a premise of $\lfloor$VTIME$\rfloor$ obtaining

$$\Gamma, a : \langle G \rangle \vdash \texttt{delay}(t).P \triangleright \Delta \tag{3.11}$$

and apply (3.11) as the premise of $\lfloor$GNRES$\rfloor$ obtaining $\Gamma \vdash (\nu a)\texttt{delay}(t).P \triangleright \Delta$ as required. The case for $\Gamma \vdash (\nu a)\texttt{delay}(t).P \triangleright \Delta$ proceeds similarly.

The case for $\texttt{delay}(t).(P \mid Q) \equiv \texttt{delay}(t).P \mid \texttt{delay}(t).Q$ proceeds as follows.

If $\Gamma \vdash \texttt{delay}(t).(P \mid Q) \triangleright \Delta$ then by premise of $\lfloor$VPAR$\rfloor$

$$\Gamma \vdash \texttt{delay}(t).P \triangleright \Delta_P \qquad \Gamma \vdash \texttt{delay}(t).Q \triangleright \Delta_Q \tag{3.12}$$

where $\Delta = \Delta_P * \Delta_Q$ and by the premise of $\lfloor$VTIME$\rfloor$ on each judgement in (3.12)

$$\Gamma \vdash P \triangleright \Delta_P + t \qquad \Gamma \vdash Q \triangleright \Delta_Q + t \tag{3.13}$$

We apply both judgements in (3.13) each as the premise of $\lfloor$VPAR$\rfloor$ and by Lemma (3.5.1):

$$\Gamma \vdash P \mid Q \triangleright \Delta + t \tag{3.14}$$

We apply (3.14) as the premise of $\lfloor$VTIME$\rfloor$ obtaining $\Gamma \vdash \texttt{delay}(t).(P \mid Q) \triangleright \Delta$, as required. The case for $\Gamma \vdash \texttt{delay}(t).(P \mid Q) \triangleright \Delta$ proceeds similarly. □

Theorem 3.5.5 (Type preservation for open systems)**.** *If* $\Gamma \vdash P \triangleright \Delta$ *and* $P \longrightarrow P'$*, then there exists* Δ' *such that* $\Delta \longrightarrow \Delta'$ *and* $\Gamma \vdash P' \triangleright \Delta'$

Proof. Theorem 3.5.5 requires to extend the typing rules for programs to processes (this extension is mechanical and similar to the one in [CDCYP15], and is reported in Appendix 3.4.3). The proof is by induction on the derivation $P \longrightarrow P'$, with a case analysis on the final rule (using Theorem 3.5.4 for the structural equivalence).

- $\lfloor$LINK$\rfloor$ - This case proceeds as in [CDCYP15] (Appendix A - Proof of Theorem 4.1).
- $\lfloor$SEL$\rfloor$ - If the reduction is by $\lfloor$SEL$\rfloor$ then $s[\mathtt{q}] \triangleleft l_j\langle e\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle)$ (with $e \downarrow v$), and by hypothesis

  $$\Gamma \vdash s[\mathtt{q}] \triangleleft l_j\langle e\rangle; P \mid s : h \triangleright \Delta$$

  By Lemma 3.5.2 (1)

  $$\Gamma \vdash s[\mathtt{q}] \triangleleft l_j\langle e\rangle; P \triangleright \Delta_1 \tag{3.15}$$

  and

  $$\Gamma \vdash s : h \triangleright \Delta_2 \tag{3.16}$$

  with $\Delta_1 * \Delta_2 = \Delta$. By Lemma 3.5.2 (2) on (3.15)

  $$\Delta_1 = \Delta_1', s : (\nu, \mathtt{q} \oplus \{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \tag{3.17}$$

with $\Gamma \vdash e : S_j$ which, by subject reduction of expressions, yields

$$\Gamma \vdash v : S_j \tag{3.18}$$

Furthermore, also by Lemma 3.5.2 (2) on (3.15):

$$\Gamma \vdash P \triangleright \Delta_1', s : (\nu', T_j) \quad \text{with } \nu' = \nu \text{ if } \lambda_j = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{3.19}$$

$$\delta_j \models \nu \tag{3.20}$$

By $\lfloor$QSEND$\rfloor$ on (3.16) we derive

$$\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \triangleright \Delta_2, \{s : \langle \mathtt{q}, l_j\langle S_j \rangle\rangle\} \tag{3.21}$$

Using $\lfloor$GPAR$\rfloor$ on (3.19) and (3.21)

$$\Gamma \vdash P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \triangleright (\Delta_1', s : (\nu', T_j)) * \Delta_2; \{s : \langle \mathtt{q}, l_j\langle S_j \rangle\rangle\}$$

Note that by $\lfloor$RPAR$\rfloor$ with premise $\lfloor$RSEND$\rfloor$ – the latter can be applied thanks to (3.20) – on $\Delta_1 * \Delta_2$ where Δ_1 is as in (3.17):

$$(\Delta_1', s : (\nu, \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) * \Delta_2 \longrightarrow (\Delta_1', s : (\nu', T_j)) * (\Delta_2, \{s : \langle \mathtt{q}, l_j\langle S_j \rangle\rangle\})$$

- $\lfloor$BRA$\rfloor$ - If the reduction is by $\lfloor$BRA$\rfloor$ then $s[\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \cdot h \longrightarrow P_j[v/z_j] \mid s : h$, and by hypothesis

$$\Gamma \vdash s[\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \cdot h \triangleright \Delta$$

By Lemma 3.5.2 (1)

$$\Gamma \vdash s[\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \triangleright \Delta_1 \tag{3.22}$$

and

$$\Gamma \vdash s : (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \cdot h \triangleright \Delta_2 \tag{3.23}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma 3.5.2 (4) on (3.22)

$$\Delta_1 = \Delta_1', s : (\nu, \mathtt{q} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \tag{3.24}$$

and

$$\forall i \in I \quad \Gamma \vdash P_i[v/z_i] \triangleright \Delta_1', s : (\nu', T_i) \quad \text{with } \nu' = \nu \text{ if } \lambda_i = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{3.25}$$

$$\forall i \in I \quad \delta_i \models \nu \tag{3.26}$$

By Lemma 3.5.2 (7) on (3.23)

$$\Gamma \vdash s : h \triangleright \Delta_2' \qquad \Delta_2 = \Delta_2', \{s : \langle \mathtt{q}, l_i \langle S_i \rangle \rangle\} \tag{3.27}$$

Using $\lfloor$GPAR$\rfloor$ on (3.25) and (3.27) for all $i \in I$:

$$\Gamma \vdash P[v/z_j] \mid s : h \triangleright (\Delta_1', s : (\nu', T_j)) * \Delta_2'$$

Note that by $\lfloor$RPAR$\rfloor$ with premise $\lfloor$RREC$\rfloor$ – the latter can be applied thanks to (3.26) – on $\Delta_1 * \Delta_2$:

$$(\Delta_1', s : (\nu, \mathtt{q} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) * \Delta_2', \{s : \langle \mathtt{q}, l_j \langle S_j \rangle \rangle\} \longrightarrow (\Delta_1', s : (\nu', T_j)) * \Delta_2'$$

- $\lfloor$SREC$\rfloor$ - If the reduction is by $\lfloor$SREC$\rfloor$ then $s[\mathtt{q}] \triangleright ((y)).P \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \longrightarrow P[s'[\mathtt{p}']/y] \mid s : h$, and by hypothesis

$$\Gamma \vdash s[\mathtt{q}] \triangleright ((y)).P \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \triangleright \Delta$$

By Lemma 3.5.2 (1)

$$\Gamma \vdash s[\mathtt{q}] \triangleright ((y)).P \triangleright \Delta_1 \tag{3.28}$$

and

$$\Gamma \vdash s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \triangleright \Delta_2 \tag{3.29}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma 3.5.2 (5) on (3.28)

$$\Delta_1 = \Delta_1', s : (\nu, \mathtt{q} \& \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T) \tag{3.30}$$

and

$$\Gamma \vdash P[s'[\mathtt{p}']/y] \triangleright \Delta_1', s : (\nu', T), s'[\mathtt{p}'] : (\nu_d, T_d) \tag{3.31}$$

with $\nu' = \nu$ if $\lambda_i = \emptyset$ and $\nu' = 0$ otherwise; and

$$\delta \models \nu \qquad \delta_d \models \nu_d \tag{3.32}$$

By Lemma 3.5.2 (8) on (3.29)

$$\Gamma \vdash s : h \triangleright \Delta_2' \qquad \Delta_2 = \Delta_2', \{s : \langle \mathfrak{q}, \langle (T_d, \delta_d) \rangle \rangle\}, s'[\mathfrak{p}'] : (\nu_d, T_d) \tag{3.33}$$

Using $\lfloor$GPar$\rfloor$ on (3.31) and (3.33):

$$\Gamma \vdash P[s'[\mathfrak{p}']/y] \mid s : h \triangleright \Delta_1', s : (\nu', T), s'[\mathfrak{p}'] : (\nu_d, T_d) * \Delta_2'$$

Note that by $\lfloor$RPar$\rfloor$ with premise $\lfloor$RRec$\rfloor$ – the latter can be applied thanks to (3.32) – on $\Delta_1 * \Delta_2$:

$$(\Delta_1', s : (\nu, \mathfrak{q} \& \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T) * \Delta_2', \{s : \langle \mathfrak{q}, \langle (T_d, \delta_d) \rangle \rangle\}, s'[\mathfrak{p}'] : (\nu_d, T_d) \longrightarrow (\Delta_1', s : (\nu', T)) * \Delta_2'$$

- $\lfloor$Del$\rfloor$ - If the reduction is by $\lfloor$Del$\rfloor$ then $s[\mathfrak{q}] \triangleleft \langle\langle s'[\mathfrak{p}'] \rangle\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathfrak{p}, \mathfrak{q}, s'[\mathfrak{p}'])$, and by hypothesis

$$\Gamma \vdash s[\mathfrak{q}] \triangleleft \langle\langle s'[\mathfrak{p}'] \rangle\rangle; P \mid s : h \triangleright \Delta$$

By Lemma 3.5.2 (1)

$$\Gamma \vdash s[\mathfrak{q}] \triangleleft \langle\langle s'[\mathfrak{p}'] \rangle\rangle; P \triangleright \Delta_1 \tag{3.34}$$

and

$$\Gamma \vdash s : h \triangleright \Delta_2 \tag{3.35}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma 3.5.2 (3) on (3.34)

$$\Delta_1 = \Delta_1', s : (\nu, \mathfrak{q} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T), s'[\mathfrak{p}'] : (\nu_d, T_d), \tag{3.36}$$

and

$$\Gamma \vdash P \triangleright \Delta_1', s : (\nu', T) \quad \text{with } \nu' = \nu \text{ if } \lambda = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{3.37}$$

$$\delta \models \nu \text{ and } \delta_d \models \nu_d \tag{3.38}$$

By $\lfloor$QDel$\rfloor$ on (3.35) we derive

$$\Gamma \vdash s : h \cdot (\mathfrak{p}, \mathfrak{q}, s'[\mathfrak{p}']) \triangleright \Delta_2, \{s : \langle \mathfrak{q}, \langle (T_d, \delta_d) \rangle \rangle\} \tag{3.39}$$

Using $\lfloor$GPAR$\rfloor$ on (3.37) and (3.39)

$$\Gamma \vdash P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, \langle (T_d, \delta_d) \rangle) \triangleright (\Delta_1', s : (\nu', T)) * \Delta_2, \{s : \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle\}$$

Note that by $\lfloor$RPAR$\rfloor$ with premise $\lfloor$RSEND$\rfloor$ – the latter can be applied thanks to (3.38) – on $\Delta_1 * \Delta_2$ where Δ_1 is as in (3.17):

$$(\Delta_1', s : (\nu, \mathtt{q} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T) * \Delta_2 \longrightarrow (\Delta_1', s : (\nu', \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle; T)) * \Delta_2$$

with

$$(\Delta_1', s : (\nu', \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle; T)) * \Delta_2 = (\Delta_1', s : (\nu', T)) * \Delta_2, \{s : \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle\}$$

- $\lfloor$DELAY$\rfloor$ - If the reduction is by $\lfloor$DELAY$\rfloor$ then $\mathtt{delay}(t).P_i \mid \prod_{j \in J} s_j : h_j \longrightarrow P \prod_{j \in J} s_j : h_j$, and by hypothesis

$$\Gamma \vdash \mathtt{delay}(t).P \mid \prod_{j \in J} s_j : h_j \triangleright \Delta$$

By Lemma 3.5.2 (1)

$$\Gamma \vdash \mathtt{delay}(t).P_i \triangleright \Delta_1 \qquad \Gamma \vdash \prod_{j \in J} s_j : h_j \triangleright \Delta_2 \qquad (\delta = \delta_1 * \delta_2) \tag{3.40}$$

By Lemma 3.5.2 (6) on (3.40 - judgement on the left hand side) $\Delta_1 = \{s_j[\mathtt{p}_j] : (\nu_j, T_j)\}_{j \in J}$ and

$$\Gamma \vdash P \triangleright \Delta_1' \qquad (\Delta_1' = \{s_j[\mathtt{p}_j] : (\nu_j + t, T_j)\}_{j \in J}) \tag{3.41}$$

Using (3.41) as a premise of ($\lfloor$VPAR$\rfloor$), observing that $dom(\Delta_1) \cap dom(\Delta_2)\} = \emptyset$ implies $dom(\Delta_1') \cap dom(\Delta_2)\} = \emptyset$

$$\Gamma \vdash P \mid \prod_{j \in J} s_j : h_j \triangleright \Delta_1' * \Delta_2 \tag{3.42}$$

By Lemma 3.5.3 on (3.42)(recall $\Delta_1' = \{s_j[\mathtt{p}_j] : (\nu_j + t, T_j)\}_{j \in J}$), T_j is satisfiable after t in ν_j, hence by $\lfloor$RTIME$\rfloor$ $\Delta \longrightarrow \Delta_1' * \Delta_2$.

□

Theorem 3.5.6 (Type preservation)**.** *If* $\Gamma \vdash P \triangleright \emptyset$ *and* $P \longrightarrow P'$ *then* $\Gamma \vdash P' \triangleright \emptyset$.

Proof. In the statement of Theorem 3.5.6, P is typed against empty Δ meaning that either (1) P has not started any session yet, or (2) P models a whole system, that is including all participants of all ongoing sessions, technically $P \equiv (\nu \tilde{s})P'$ for some $\tilde{s}'$ and with P' having no free session names.

This is a special case of Theorem 3.5.5. The proof of Theorem 3.5.5 subsumes that of Theorem 3.5.6). □

Time-Error Freedom

A standard corollary of type preservation is error-freedom. An error state is reached when a process violates the constraints prescribed by its type. Technically we proceed as follows:

1. In the syntax we annotate selection and branching with clock constraints and reset predicates, and add error and clock processes:

   $$P ::= \ldots \mid c[\mathrm{p}] \triangleleft l\langle e \rangle \{\delta, \lambda\}; P \mid c[\mathrm{p}] \triangleright \{l_i(z_i)\{\delta_i, \lambda_i\}.P_i\}_{i \in I} \mid \texttt{error} \mid (s, \nu)$$

2. We add $\texttt{error} \mid P \equiv \texttt{error}$ to structural equivalence;

3. In the reduction rules we modify $\lfloor$Link$\rfloor$ so that a clock process with initial assignment is introduced for each participant in the new session, let $\lfloor$Time$\rfloor$ increment all clock assignments, extend $\lfloor$Sel$\rfloor$/$\lfloor$Bra$\rfloor$ so that they check clock constraints against clock assignments and appropriately reset, and we add two rules ($\lfloor$SelErr$\rfloor$ and $\lfloor$BraErr$\rfloor$) that move to `error` when a process tries to perform an action at a time that does not satisfy the constraint, e.g.,

   $$s[\mathrm{q}] \triangleleft l\langle e \rangle \{\delta, \lambda\}; P \mid (s, \nu) \mid s : h \longrightarrow \texttt{error} \mid (s, \nu) \mid s : h \quad (\nu \not\models \delta) \quad \lfloor \textsc{SelErr} \rfloor$$

The proof rules are adapted straightforwardly (with `error` not validated against any Δ).

Lemma 3.5.7. *If* $\Gamma \vdash P \triangleright \Delta$*, then* $P \not\equiv \texttt{error}$.

Proof. The proof of Lemma 3.5.7 is mechanical by induction on G proceeding by case analysis and inspecting the extended proof rules. □

Theorem 3.5.8 (Time-error freedom)**.** *If* $\Gamma \vdash P \triangleright \Delta$*, and* $P \to^* P'$ *then* $P' \not\equiv \texttt{error}$

Proof. Technically, we can prove error-freedom through the following steps.

P	::=	$\overline{u}[\mathtt{n}](y).P$	Request
		$u[\mathtt{i}](y).P$	Accept
		$c[\mathtt{p}] \triangleleft \{\delta, \lambda\} l\langle e \rangle; P$	Select
		$c[\mathtt{p}] \triangleright \{\{\delta_i, \lambda_i\} l_i(z_i).P_i\}_{i \in J}$	Branching
		$c[\mathtt{p}] \triangleleft \{\delta, \lambda\} \langle\langle c', \nu \rangle\rangle; P$	Delegate
		$c[\mathtt{p}] \triangleright \{\delta, \lambda\} ((y)).P$	Session receive
		$\mathtt{delay}(t).P$	Delay
		$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional
		$P \mid Q$	Parallel
		0	Inaction
		$\mu X.P$	Recursion
		$\mathtt{error}$	Error
		X	Variable
		$(\nu a)P$	Hide Shared
		$(\nu s)P$	Hide Session
		(c, ν)	Clock Assignment
		$s : h$	Queue
h	::=	$\emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, m)$	(queue content)
m	::=	$l\langle v \rangle$	(messages)
c	::=	$s \mid y$	(session names/variables)
u	::=	$a \mid u$	(shared names/variables)
n	::=	$a \mid s$	(names)

Figure 3.8.: Extended Syntax of Processes for Time-error Freedom.

Step 1 - extend processes In our scenario an error state, which we call `error`, is a state reached when a process violates the clock constraints associated with the action prescribed by the corresponding local timed type. To model reduction to error states we extend the syntax of processes with clock constraints, reset predicate and clock assignment (see Figure 3.8).

Step 2 - extend reductions The extended reduction rules are given in Figure 3.9, plus rules $\lfloor$IFT$\rfloor$, $\lfloor$IFF$\rfloor$, $\lfloor$CONT$\rfloor$, $\lfloor$STR$\rfloor$, and $\lfloor$COM$\rfloor$, from Figure 3.4 that remain unchanged. Rule $\lfloor$LINK$\rfloor$ extends session initiation by adding an initial clock assignment $(s[\mathtt{i}], \nu_0)$ for each participant `i` of the new session. The rules for selection and branching, $\lfloor$SEL$\rfloor$ and $\lfloor$BRA$\rfloor$ check clock constraints against clock assignments and appropriately reset the clock assignments. Rules $\lfloor$DEL$\rfloor$ and $\lfloor$SREC$\rfloor$ for session delegation update the set of clock processes. Time actions by $\lfloor$DELAY$\rfloor$ increment the clock assignments. We introduce error reductions to be triggered

when a process tries to perform an action in a time that does not satisfy the prescribed constraint: $\lfloor$EBra$\rfloor$, $\lfloor$ESel$\rfloor$, $\lfloor$EDel$\rfloor$ and $\lfloor$ESRec$\rfloor$ are error reductions for send, branching, delegation, session receive. Note that rules $\lfloor$EBra$\rfloor$ and $\lfloor$ESRec$\rfloor$ can be applied both when p tries to read too early or too late w.r.t. the prescribed time.

Step 3 - Extended Validation Rules Figure 3.10 gives the extended proof rules for processes with errors and clocks. We show the rules that are different from Figure 3.5 and omit the rules that are unchanged.

Let $\Gamma \vdash P_0 \triangleright \Delta$ and $P_0 \longrightarrow^* P \longrightarrow P'$. We proceed by induction and assume that $P \not\equiv \texttt{error}$ and proceed by case analysis on the last reduction $P \longrightarrow P'$.

In the case analysis, we use again induction on the (depth of the tree of the) reduction rule applied. The inductive cases, $\lfloor$Cont$\rfloor$, $\lfloor$Str$\rfloor$ and $\lfloor$Com$\rfloor$, and are straightforward. We consider now the base cases. If the last reduction is by rules $\lfloor$Link$\rfloor$, $\lfloor$Delay$\rfloor$, $\lfloor$Sel$\rfloor$, $\lfloor$Bra$\rfloor$, $\lfloor$Del$\rfloor$, $\lfloor$SRec$\rfloor$, $\lfloor$IfT$\rfloor$, or $\lfloor$IfF$\rfloor$, then $P' \neq \texttt{error}$ since (1) the rules do not introduce error processes, and (2) error processes are a run-time process which cannot appear as a continuation of P. Furthermore, we can proceed similarly to Theorem 3.5.6 and show that $\Gamma' \vdash P' \triangleright \Delta'$ for some Γ' and Δ' (obtained by some reduction from Γ and Δ). This yields the result for these cases.

The only reductions that introduce error processes are $\lfloor$ESel$\rfloor$, $\lfloor$EBra$\rfloor$, $\lfloor$EDel$\rfloor$, $\lfloor$ESRec$\rfloor$. We consider $\lfloor$ESel$\rfloor$ (the others proceed similarly). Assume $\lfloor$ESel$\rfloor$ can be applied to P which is therefore of the following form: $s[\mathtt{q}] \triangleleft \{\delta, \lambda\} l\langle e \rangle; P'$. Then by $\lfloor$ESel$\rfloor$

$$\neg \nu \models \delta \tag{3.43}$$

By hypothesis P is well typed. By inspecting the validation rules, the derivation of $s[\mathtt{q}] \triangleleft \{\delta, \lambda\} l\langle e \rangle; P'$ is done by applying the rule $\lfloor$VSel$\rfloor$

$$\frac{\Gamma \vdash e : S_j \quad \Gamma \vdash P' \mid (s, \nu') \triangleright \Delta, s : (\nu', T') \quad \nu' = [\lambda_j \mapsto 0]\nu \qquad \nu \models \delta}{\Gamma \vdash s[\mathtt{q}] \triangleleft \{\delta, \lambda\} l\langle e \rangle; P' \mid (s, \nu) \triangleright \Delta, s : (\nu, \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i'\}_{i \in I})}$$

with the condition $\nu \models \delta$, which contradicts (3.43) and the fact that $\lfloor$ESel$\rfloor$ can be applied. The fact that none of the reductions that introduce errors can be applied (together with the fact that $P \not\equiv \texttt{error}$) yields the theorem. □

$$\frac{\forall i \in \{1,..,n\} \quad s \notin \mathsf{fn}P_i}{\overline{a}[\mathtt{n}](y).P_1 \mid \prod_{i\in\{2,..,n\}} a[\mathtt{i}](y).P_i \longrightarrow (\nu s)(\prod_{i\in\{1,..,n\}}(P_i[s[\mathtt{i}]/y] \mid (s[\mathtt{i}],\nu_0)) \mid s:\emptyset)} \lfloor \textsc{Link} \rfloor$$

$$\frac{e \downarrow v \qquad \nu' = [\lambda \mapsto 0]\nu \qquad \delta \models \nu}{s[\mathtt{q}] \triangleleft \{\delta,\lambda\} l\langle e\rangle; P \mid s:h \mid (s,\nu) \longrightarrow P \mid s:h\cdot(\mathtt{p},\mathtt{q},l\langle v\rangle) \mid (s,\nu')} \lfloor \textsc{Sel} \rfloor$$

$$\frac{j \in J \qquad \nu' = [\lambda_j \mapsto 0]\nu \qquad \delta_j \models \nu}{s[\mathtt{q}] \triangleright \{\{\delta_i,\lambda_i\} l_i(z_i).P_i\}_{i\in J} \mid s:(\mathtt{p},\mathtt{q},l_j\langle v\rangle)\cdot h \mid (s,\nu) \longrightarrow P_j[v/z_j] \mid s:h \mid (s,\nu')} \lfloor \textsc{Bra} \rfloor$$

$$\frac{\nu' = [\lambda \mapsto 0]\nu \qquad \delta \models \nu}{s[\mathtt{q}] \triangleleft \{\delta,\lambda\}\langle\langle s'[\mathtt{p}'],\nu''\rangle\rangle; P \mid s:h \mid (s,\nu) \mid (s'[\mathtt{p}'],\nu'') \longrightarrow P \mid s:h\cdot(\mathtt{p},\mathtt{q},(s'[\mathtt{p}'],\nu'')) \mid (s[\mathtt{p}],\nu')} \lfloor \textsc{Del} \rfloor$$

$$\frac{\nu' = [\lambda \mapsto 0]\nu \qquad \delta \models \nu}{s[\mathtt{q}] \triangleright \{\delta,\lambda\}((y)).P \mid s:(\mathtt{p},\mathtt{q},(s'[\mathtt{p}'],\nu''))\cdot h \mid (s,\nu) \longrightarrow P[s'[\mathtt{p}']/y] \mid s:h \mid (s[\mathtt{p}],\nu') \mid (s'[\mathtt{p}'],\nu'')} \lfloor \textsc{SRec} \rfloor$$

$$\mathtt{delay}(t).P \mid \prod_{j\in J}(s_j:h_j \mid \prod_{k\in K_j}(s_j[\mathtt{p}_k],\nu_k)) \mid \longrightarrow P \mid \prod_{j\in J}(s_j:h_j \mid \prod_{k\in K_j}(s_j[\mathtt{p}_k],\nu_k+t)) \lfloor \textsc{Delay} \rfloor$$

$$\frac{j \in J \qquad \neg\delta_j \models \nu}{s[\mathtt{q}] \triangleright \{\{\delta_i,\lambda_i\} l_i(z_i).P_i\}_{i\in J} \mid s:h \mid (s,\nu) \longrightarrow \mathtt{error} \mid s:h \mid (s,\nu)} \lfloor \textsc{EBra} \rfloor$$

$$\frac{\neg\delta \models \nu}{s[\mathtt{q}] \triangleleft \{\delta,\lambda\} l\langle e\rangle; P \mid s:h \mid (s,\nu) \longrightarrow \mathtt{error} \mid s:h \mid (s,\nu)} \lfloor \textsc{ESel} \rfloor$$

$$\frac{\neg\delta \models \nu}{s[\mathtt{q}] \triangleleft \{\delta,\lambda\}\langle\langle s'[\mathtt{p}'],\nu\rangle\rangle; P \mid s:h \mid (s,\nu) \mid (s'[\mathtt{p}'],\nu'') \longrightarrow \mathtt{error} \mid s:h \mid (s,\nu) \mid (s'[\mathtt{p}'],\nu'')} \lfloor \textsc{EDel} \rfloor$$

$$\frac{\neg\delta \models \nu}{s[\mathtt{q}] \triangleright \{\delta,\lambda\}((y)).P \mid s:h \mid (s,\nu) \longrightarrow \mathtt{error} \mid s:h \mid (s,\nu)} \lfloor \textsc{ESRec} \rfloor$$

Figure 3.9.: Modified/Added Rules of Extended Reductions

$$\frac{\Gamma, u : G \vdash P \mid (y[1], \nu_0) \triangleright \Delta, y[1] : (\nu_0, G\downarrow_1) \quad dom(\nu_0) = \{x_1\} \quad \nu_0(x_1) = 0}{\Gamma, u : G \vdash \overline{u}[\mathtt{n}](y).P \triangleright \Delta} \lfloor \text{VREQ} \rfloor$$

$$\frac{\Gamma, u : G \vdash P \mid (y[\mathtt{i}], \nu_0) \triangleright \Delta, y[\mathtt{i}] : (\nu_0, G\downarrow_{\mathtt{i}}) \quad dom(\nu_0) = \{x_{\mathtt{i}}\} \quad \nu_0(x_{\mathtt{i}}) = 0 \quad \mathtt{i} \neq 1}{\Gamma, u : G \vdash u[\mathtt{i}](y).P \triangleright \Delta} \lfloor \text{VACC} \rfloor$$

$$\frac{j \in I \quad \Gamma \vdash e : S_j \quad \nu' = [\lambda_j \mapsto 0]\nu \quad \delta_j \models \nu \quad \Gamma \vdash P \mid (c, \nu') \mid \prod(c_i, \nu_i) \triangleright \Delta, c : (\nu', T_j)}{\begin{array}{c}\Gamma \vdash c[\mathtt{p}] \triangleleft \{\delta, \lambda\} l_j \langle e \rangle; P \mid (c, \nu) \mid \prod(c_i, \nu_i) \\ \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})\end{array}} \lfloor \text{VSEL} \rfloor$$

$$\frac{\nu' = [\lambda \mapsto 0]\nu \quad \nu \models \delta \quad \nu_d \models \delta_d \quad \Gamma \vdash P \mid (c, \nu') \mid \prod(c_i, \nu_i) \triangleright \Delta, c : (\nu', T') \quad \Gamma \vdash P \mid (c, \nu') \mid \prod(c_i, \nu_i) \triangleright \Delta, c : (\nu', T')}{\begin{array}{c}\Gamma \vdash c[\mathtt{p}] \triangleleft \{\delta, \lambda\} \langle\langle c', \nu_d \rangle\rangle; P \mid (c, \nu) \mid (c', \nu_d) \mid \prod(c_i, \nu_i) \\ \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T'), c' : (\nu_d, T_d)\end{array}} \lfloor \text{VDEL} \rfloor$$

$$\frac{\forall i \in I \quad \nu'_i = [\lambda_i \mapsto 0]\nu \quad \nu \models \delta_i \quad \Gamma, z_i : S_i \vdash P_i \mid (c, \nu'_i) \mid \prod(c_i, \nu_i) \triangleright \Delta, c : (\nu_i, T_i)}{\begin{array}{c}\Gamma \vdash \ c[\mathtt{p}] \triangleright \{\{\delta_i, \lambda_i\} l_i(z_i).P_i\}_{i \in I} \mid (c, \nu) \mid \prod(c_i, \nu_i) \\ \triangleright \Delta, c : (\nu, \mathtt{p} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})\end{array}} \lfloor \text{VBRA} \rfloor$$

$$\frac{\nu' = [\lambda \mapsto 0]\nu \quad \nu \models \delta \quad \nu_d \models \delta_d \quad \Gamma \vdash P \mid (c, \nu') \mid \prod(c_i, \nu_i) \triangleright \Delta, c : (\nu', T'), y : (\nu_d, T_d)}{\begin{array}{c}\Gamma \vdash \ c[\mathtt{q}] \triangleright \{\delta, \lambda\}((y)).P \mid (c, \nu) \mid (c', \nu_d) \mid \prod(c_i, \nu_i) \\ \triangleright \Delta, c : (\nu, \mathtt{p} \& \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T')\end{array}} \lfloor \text{VSREC} \rfloor$$

$$\frac{\Gamma \vdash P \mid (s, \nu + t) \mid \prod(c_i, \nu_i + t) \triangleright \{c_i : (\nu_i + t, T_i)\}_{i \in I}}{\Gamma \vdash \mathtt{delay}(t).P \mid (s, \nu) \mid \prod(c_i, \nu_i) \triangleright \{c_i : (\nu_i, T_i)\}_{i \in I}} \lfloor \text{VTIME} \rfloor$$

$$\frac{dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \quad \Gamma \vdash P_i \triangleright \Delta_i \quad P_i \equiv P'_i \mid (c, \nu) \Rightarrow c \notin dom(\Delta_j) \quad i \neq j \in \{1, 2\}}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \lfloor \text{VPAR} \rfloor$$

$$\frac{\forall c \in dom(\Delta) \quad \Delta(c) = (\nu, \mathtt{end})}{\Gamma \vdash 0 \mid \prod_i (c_i, \nu_i) \triangleright \Delta} \lfloor \text{VEND} \rfloor$$

Figure 3.10.: Extended Reduction Rules with Error Processes

1. $\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} > 3,\ _\ , x_{\mathtt{q}} = 4,\ _\ \}$
2. $\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} > 3 \wedge x_{\mathtt{p}} \leq 4,\ _\ , x_{\mathtt{q}} = 4,\ _\ \}$
3. $\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} > 3,\ _\ , x_{\mathtt{q}} \geq 4,\ _\ \}$
4. $\mathtt{q} \to \mathtt{r} : \{ l_1 : \{ x_{\mathtt{q}} > 3,\ _\ , \mathsf{true},\ _\ \},\ l_2 : \{ x_{\mathtt{q}} < 3,\ _\ , \mathsf{true},\ _\ \} \}$
5. $\mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} < 1,\ _\ , x_{\mathtt{q}} = 2, x_{\mathtt{q}} \}.\mathtt{p} \to \mathtt{r} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} < 5, x_{\mathtt{p}}, \mathsf{true}, x_{\mathtt{r}} \}.\mathbf{t}$
6. $\mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} < 1,\ _\ , x_{\mathtt{q}} = 2, x_{\mathtt{q}} \}.\mathtt{p} \to \mathtt{r} : \langle \mathsf{int} \rangle \{ x_{\mathtt{p}} < 1, x_{\mathtt{p}}, \mathsf{true}, x_{\mathtt{r}} \}.\mathbf{t}$

Figure 3.11.: Examples of non-feasible global types and their amended feasible versions

3.6. Time-Progress of Timed Processes

This section studies a subclass of timed global types characterised by two properties, *feasibility* and *global satisfiability*, which ensure progress in terms of time. Then we show the theorem of time-progress to verify the progresses of timed processes.

3.6.1. Feasibility and Wait-freedom

A *TG* G_0 is *feasible* iff $(\nu_0, G_0) \to^* (\nu, G)$ implies $(\nu, G) \to^* (\nu', \mathsf{end})$. Intuitively, G is feasible if every partial execution can be extended to an infinite one (note that executions reaching `end` are infinite). The specified protocol may get stuck because a constraint is unsatisfiable, for example it is false, or the restrictions posed by previously occurred constraints are too strong. In Figure 3.11: in (1), if $\mathtt{p}$ sends $\langle \mathsf{int} \rangle$ at time 5 (which satisfies $x_{\mathtt{p}} > 3$) then there exists no $x_{\mathtt{q}}$ satisfying $x_{\mathtt{q}} = 4$ ($x_{\mathtt{q}}$ must be greater than or equal to 5 to respect the global flowing of time); (2) amends (1) by restricting an earlier constraint; (3) amends (1) by relaxing the unsatisfiable constraint. In branching and selection at least one constraint associated to the branches must be satisfiable, e.g., we accept (4). In recursive *TG*s, a constraint may become unsatisfiable by the restrictions posed by constraints that occur after, syntactically, in the same recursion body. In the second iteration of (5) $x_{\mathtt{q}} = 2$ is made unsatisfiable by the restriction $x_{\mathtt{p}} < 5$ from the first iteration (e.g., $\mathtt{p}$ may send $\mathtt{q}$ the message when $x_{\mathtt{q}} > 2$); in (6) this problem is solved by restricting the second constraint on $x_{\mathtt{p}}$.

Global Satisfiability. Timed global type G_0 is *wait-free* iff $(\nu_0, G_0) \to^* \xrightarrow{\mathtt{pq}!l\langle S \rangle} (\nu, G)$ and $\mathtt{p} \rightsquigarrow \mathtt{q} : \{ l : \langle S \rangle \{ \delta_O, \lambda_O, \delta_I, \lambda_I \}.G' \} \in G$ imply $\delta_I(x) \supset \nu(x) \leq x$.

Intuitively, G_0 is wait-free when in all its distributed implementations, a receiver checking the queue at any time satisfying the constraint δ_I, never has to wait for the message. We give below that a process (i.e., $P \mid Q$) can get stuck despite being a correct implementation (with respect to the validation rules) of a feasible TG.

$$
\begin{array}{lcl}
G & = & \mathtt{p} \to \mathtt{q} : \langle \mathsf{int} \rangle \{x_{\mathtt{p}} < 3 \vee x_{\mathtt{p}} > 3,\ _\ , x_{\mathtt{q}} < 3 \vee x_{\mathtt{q}} > 3,\ _\ \}. \\
 & & \mathtt{q} \to \mathtt{p} : \{l_1 : \{x_{\mathtt{q}} > 3,\ _\ , x_{\mathtt{p}} > 3,\ _\ \},\ l_2 : \{x_{\mathtt{q}} < 3,\ _\ , x_{\mathtt{p}} < 3,\ _\ \}\} \\
G\downarrow_{\mathtt{q}} & = & \mathtt{p}\&\langle \mathsf{int} \rangle \{x_{\mathtt{q}} < 3 \vee x_{\mathtt{q}} > 3,\ _\ \}.\mathtt{p} \oplus \{l_1 : \{x_{\mathtt{q}} > 3,\ _\ \},\ l_2 : \{x_{\mathtt{q}} < 3,\ _\ \}\} \\
G\downarrow_{\mathtt{p}} & = & \mathtt{q} \oplus \langle \mathsf{int} \rangle \{x_{\mathtt{p}} < 3 \vee x_{\mathtt{p}} > 3,\ _\ \}.\mathtt{q}\&\{l_1 : \{x_{\mathtt{p}} > 3,\ _\ \},\ l_2 : \{x_{\mathtt{p}} < 3,\ _\ \}\} \\
Q & = & s[\mathtt{p},\mathtt{q}] \triangleright (x).s[\mathtt{q},\mathtt{p}] \triangleleft l_2\langle\rangle;0 \\
P & = & \mathtt{delay}(6).s[\mathtt{p},\mathtt{q}] \triangleleft \langle 10 \rangle; s[\mathtt{q}][\mathtt{p}] \triangleright \{l_1.0,\ l_2.0\}
\end{array}
$$

Q implements $G\downarrow_{\mathtt{q}}$: it receives a message from $\mathtt{p}$ and then selects label l_2; both interactions occur at time 0 which satisfies the clock constraints. P implements $G\downarrow_{\mathtt{p}}$: it waits 6 units of time, then sends $\mathtt{q}$ a message and waits for the reply. We can validate $P \mid Q$ against $\{s[\mathtt{p}] : (\nu_0, G\downarrow_{\mathtt{p}}), s[\mathtt{q}] : (\nu_0, G\downarrow_{\mathtt{q}})\}$ which by Theorem 3.5.8 all interactions conform to the specification. However the execution of $P \mid Q$ will get stuck, i.e., the session cannot proceed as prescribed.

3.6.2. Auxiliary Definitions for Reasoning on Clocks

We extend the definition of clock constraints by allowing them to be defined on more than one clock. This allows us to compose constraints, and to determine the possible times at which an action in G can happen on the basis of the constraints that occur earlier of that action in G. The aim is to check for the satisfiability of a constraint in all possible execution scenarios.

Definition 3.6.1 (Extended clock constraints)**.** *The set of extended clock constraints* δ *on* X *is:*

$$
\begin{array}{lcl}
\delta & ::= & \mathsf{true} \mid x > e \mid x = e \mid \neg\delta \mid \delta_1 \wedge \delta_2 \mid \\
e & ::= & x \mid c \mid e + e
\end{array}
$$

The extension of $fc(\delta)$ to constraints on multiple clocks is straightforward. Hereafter we implicitly assume constraints are extended constraints.

Definition 3.6.2 ($\delta + \tilde{x}$)**.** *The sum of a constraint with a clock, written* $\delta + x$ *is defined inductively as follows:* $\mathsf{true} + x = \mathsf{true}$, $(x' \ \mathtt{bop}\ e) + x = x' \ \mathtt{bop}\ (e + x)$ *with*

$\mathtt{bop} \in \{\wedge, \vee, =\}$, $(\neg\delta) + x = \neg(\delta + x)$. *The sum of a constraint with a vector of clocks is* $\delta + \tilde{x} = (\delta + x_0) + \tilde{x}'$ *with* $\tilde{x} = x_0 + \tilde{x}'$.

3.6.3. Time Graphs

To make dependencies between constraints explicit when checking feasibility and wait-freedom, we create a representation of global timed types as *time graphs*.

Let G be a timed global type; a subterm G' of G, written $G' \in G$, is a timed global type that appears in G. To build the time graph of G we first annotate each $G' \in G$ with a distinguished name, ranged over by $\overline{\mathtt{n}}$. We denote the nodes $\mathtt{n}$ of a timed graph in the form of: $(\overline{\mathtt{n}}, i, !)$ for an output action in the branch i of an interaction; $(\overline{\mathtt{n}}, i, ?)$ for an input action of branch i; $(\overline{\mathtt{n}}, \mu\mathbf{t})$ for a recursive type; $(\overline{\mathtt{n}}, \mathbf{t})$ for a type variable and the special node $\mathtt{end}$. We build the set of nodes from an annotated timed global type as follows:

$$
\begin{array}{lcl}
\mathtt{nodes}(\overline{\mathtt{n}} : \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_I \rangle \{A_i\}.G_i\}_{i \in I}) & = & \{(\overline{\mathtt{n}}, i, \#) \mid i \in I,\ \# \in \{!, ?\}\} \cup \\
 & & \bigcup_{i \in I} \mathtt{nodes}(G_i) \\
\mathtt{nodes}(\overline{\mathtt{n}} : \mu\mathbf{t}.G') & = & \{(\overline{\mathtt{n}}, \mu\mathbf{t})\} \cup \mathtt{nodes}(G') \\
\mathtt{nodes}(\mathbf{t}) & = & \{(\overline{\mathtt{n}}, \mathbf{t})\} \\
\mathtt{nodes}(\overline{\mathtt{n}} : \mathtt{end}) & = & \{\mathtt{end}\}
\end{array}
$$

The set of nodes deriving from G but not from the subterms of G, written $\mathtt{t_nodes}(G)$, is defined as $\mathtt{nodes}(G) \setminus \{\mathtt{n} \mid \mathtt{n} \in \mathtt{nodes}(G') \text{ and } G' \in G\}$. Vice versa, we use the function $\mathtt{term}_G(\mathtt{n})$ to return the subterm of annotated type G corresponding to node $\mathtt{n}$. We will use the following auxiliary functions that can be defined straightforwardly from $\mathtt{term}_G(\mathtt{n})$. Let $\mathtt{n} \in \mathtt{nodes}(G)$ and

$$\mathtt{term}_G(\mathtt{n}) = \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_i \rangle \{\delta_{Oi}, \lambda_{Oi}, \delta_{Ii}, \lambda_{Ii}\}.G_i\}_{i \in I}$$

then:

- $subj(\mathtt{n})$ returns the participant that is performing the action associated to $\mathtt{n}$. More precisely: let $\mathtt{n} = (\overline{\mathtt{n}}, i, \#)$; then $subj(\mathtt{n}) = \mathtt{p}$ if $\# = !$ and $subj(\mathtt{n}) = \mathtt{q}$ otherwise.
- $\mathtt{const}(\mathtt{n})$ returns: δ_{Oi} if $\mathtt{n} = (\overline{\mathtt{n}}, i, !)$, and δ_{Ii} if $\mathtt{n} = (\overline{\mathtt{n}}, i, ?)$.
- $\mathtt{resInfo}(\mathtt{n})$ returns: λ_{Oi} if $\mathtt{n} = (\overline{\mathtt{n}}, i, !)$, and λ_{Ii} if $\mathtt{n} = (\overline{\mathtt{n}}, i, ?)$.

All functions above return $\emptyset$ if $\mathtt{term}_G(\mathtt{n})$ is not an interaction type.

Definition 3.6.3 (Time graph of a *TG*)**.** *The time graph of G is a pair* (N,E) *where the set of nodes* $N = \mathtt{nodes}(G)$ *and E is defined as follows:*

- *(I/O dependency) for each* $(\overline{\mathtt{n}},i,!)$ *and* $(\overline{\mathtt{n}},i,?)$ *in N,* $((\overline{\mathtt{n}},i,!),(\overline{\mathtt{n}},i,?)) \in E$*,*
- *(single participant dependency) for each pair of nodes* $\mathtt{n}_1 = (\overline{\mathtt{n}},i,\#)$ *and* $\mathtt{n}_2 = (\overline{\mathtt{n}},i',\#')$ *in N, if* $\mathtt{term}_G(\mathtt{n}_2) \in \mathtt{term}_G(\mathtt{n}_1)$*,* $subj(\mathtt{n}_1) = subj(\mathtt{n}_2)$ *and* $\overline{\mathtt{n}} \neq \overline{\mathtt{n}}'$ *then* $(\mathtt{n}_1,\mathtt{n}_2) \in E$*.*
- *(syntactic dependency and recursion) for each node* $\mathtt{n} \in N$*:*
  1. *if* $\mathtt{term}_G(\mathtt{n}) = \mu \boldsymbol{t}.G'$ *then* $(\mathtt{n},\mathtt{t_nodes}(G')) \in E$*, and*
  2. *if* $\mathtt{n} = (\overline{\mathtt{n}},i,\#)$ *and* $\mathtt{term}_G(\mathtt{n}) = \mathtt{p} \to \mathtt{q} : \{l_j : \langle S_j \rangle \{\delta_{Oj},\lambda_{Oj},\delta_{Ij},\lambda_{Ij}\}.G_j\}_{j \in J}$ *then for all j such that* $G_j \in \{\boldsymbol{t},\mathtt{end}\}$*,* $(\mathtt{n},\mathtt{t_nodes}(G_j)) \in E$*.*

The edges of the time graph of G define an immediate successor relation (i.e., $(\mathtt{n},\mathtt{n}') \in N$) between nodes written $\mathtt{n} <_G \mathtt{n}'$. We refer to a path in the time graph as $\mathtt{n}_0 <_G \ldots <_G \mathtt{n}_m$ or more concisely $\mathtt{n}_0,\ldots,\mathtt{n}_m$. We say that a path $\mathtt{n}_0,\ldots,\mathtt{n}_m$ has postfix $\mathtt{n}_i,\ldots,\mathtt{n}_m$ for $0 < i < m$. A path from $\mathtt{n}$ to $\mathtt{n}'$ is a path of the form $\mathtt{n},\ldots,\mathtt{n}'$. We write $\mathtt{n},p$ for the path obtained by appending path p to $\mathtt{n}$. Similarly, we write $\mathtt{p},\mathtt{n}$ for the path obtained appending node $\mathtt{n}$ to p. We write $p \setminus p'$ for the path obtained by removing from p all the nodes in p' (assuming node names unique within a path).

3.6.4. Checking Feasibility and Wait-Freedom

Feasibility

To check feasibility for G we need to check for the satisfiability of each constraint in G, in every possible scenario that satisfies the previously occurred constraints. We also need to take care that each constraint occurring in the body of a recursive global timed type can be satisfied when the body is executed infinitely many times.

If global types contain recursion, we must consider that the action associated to each node $\mathtt{n}$ in a recursion body can be executed in two different scenarios: (1) in the first iteration, (2) in successive iterations. In (2) we must consider that the time scenario in successive iterations is affected by time constraints occurring syntactically after $\mathtt{n}$ in the recursion body. To account for this, we check feasibility on the one-time unfolding of *all* the recursions in a *TG*. We will show

the conditions by which only one unfolding is sufficient to guarantee that $\mathtt{n}$ will be also satisfiable in any successive unfolding (hence iteration instance).

Step 1 : all-unfolding. The first step for checking feasibility of G is to make the one-time unfolding of all recursions in G proceeding bottom-up, that is starting from the latest occurrences of recursion definitions. Let $\mathbf{t}_1,\ldots,\mathbf{t}_n$ be the type variables occurring in G where if $i<j$ then $\mu\mathbf{t}_i$ does not occur after $\mu\mathbf{t}_j$ in G. The all-unfolding of G wrt $\mathbf{t}_1,\ldots,\mathbf{t}_{n-1},\mathbf{t}_n$ is defined recursively as the all unfolding of $G[G'_n/\mu\mathbf{t}_n.G_n]$ where $G'_n = G_n[\mu\mathbf{t}_{n-1}.G_{n-1}/\mathbf{t}_{n-1}]$ wrt $\mathbf{t}_1,\ldots,\mathbf{t}_{n-1}$. Hereafter we leave the vector of process variable implicit and simply refer to the all-unfolding of G.

Step 2 : the time graph. A time graph is generated on the all-unfolding of G.

Step 3 : the dependency constraints We define for each node of the time graph of G a dependency constraint that models the possible time scenarios in which the action associated to that node is executed (i.e., a virtual time). This virtual time is calculated by considering that the current instant in time must not precede the time of past actions. The time of past actions is determined considering the past constraints (as reset may have occurred, the virtual time must be appropriately shifted forward wrt constraints referring to reset clocks).

We model the time scenario of node i as $\overline{\delta}_{\mathtt{n}}$ and use a function R that takes a node and a participant and returns the sum of clock states in which this participant has reset the clock.

Let us set default initial values to recursively calculate the time scenario of a node: $\overline{\delta}_0 = \mathsf{true}$ and $\overline{R}(\mathtt{n}_0,\mathtt{p}) = 0$ for all $\mathtt{p}\in\mathcal{P}(G)$. Let $\mathtt{n}$ be a node in the time graph of the all-unfolding of G with $\delta = \mathtt{const}(\mathtt{n})$, $\mathit{subj}(\mathtt{n}) = \mathtt{p}$, and $M = \{\mathtt{n}' \mid \mathtt{n}' \in N \text{ and } \mathtt{n}' <_G \mathtt{n}\}$.

$$\overline{\delta}_{\mathtt{n}} \stackrel{\text{def}}{=} \wedge_{\mathtt{n}'\in M}\overline{\delta}_{\mathtt{n}'}(\tilde{x}) \wedge (\delta(x_{\mathtt{p}})[x^{\mathtt{n}}_{\mathtt{p}}/x_{\mathtt{p}}] + \overline{R}(\mathtt{n}',\mathtt{p})) \wedge (\tilde{x} \le x^{\mathtt{n}}_{\mathtt{p}})$$

$$\overline{R}(\mathtt{n},\mathtt{p}) := \begin{cases} \sum_{\mathtt{n}'\in M}\overline{R}(\mathtt{n}',\mathtt{p}) + x^{\mathtt{n}}_{\mathtt{p}} & \text{if } \mathtt{resInfo}(\mathtt{n}) = \{x_{\mathtt{p}}\} \\ \sum_{\mathtt{n}'\in M}\overline{R}(\mathtt{n}',\mathtt{p}) & \text{otherwise} \end{cases}$$

Definition 3.6.4 (Dependency constraint and dependency reset)**.** *Let* $\mathtt{n}$ *be a node in* G *and* $M = \{\mathtt{n}' \mid \mathtt{n}' \in N$ *and* $\mathtt{n}' <_G \mathtt{n}\}$. *The dependency constraint of* $\mathtt{n}$ *is defined*

as follows:

$$\delta_{\mathtt{n}} \stackrel{def}{=} \wedge_{\mathtt{n}' \in M} \overline{\delta}_{\mathtt{n}'}$$

The dependency reset of $\mathtt{n}$ *is defined as follows:*

$$R(\mathtt{n},\mathtt{p}) \stackrel{def}{=} \sum_{\mathtt{n}' \in M} \overline{R}(\mathtt{n}',\mathtt{p})$$

Step 4 : checking satisfiability of time graph We define time graph satisfiability (Definition 3.6.6) in terms of node satisfiability (Definition 3.6.5).

Definition 3.6.5 (Satisfiable node)**.** *Let* $\mathtt{n}$ *be a node of the (all-unfolding) time graph of G,* $subj(\mathtt{n}) = \mathtt{p}$*. We say that* $\mathtt{n}$ *is* satisfiable *if:*

$$\begin{cases} \delta_n(\tilde{x}) \supset \exists x.(\delta(x) + R(\mathtt{n},\mathtt{p})) \wedge (\tilde{x} \leq x) & \text{if } \mathtt{const}(\mathtt{n}) = \delta \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Definition 3.6.6 (Satisfiable time graph)**.** *The time graph of G is satisfiable if*

1. *all nodes* $\mathtt{n}$ *that are not of the form* $(\overline{\mathtt{n}}, i, !)$ *are satisfiable, and*
2. *for all* $(\overline{\mathtt{n}}, i, !)$ *(with* $\mathtt{term}_G((\overline{\mathtt{n}}, j, !)) = \mathtt{p} \to \mathtt{q} : \{l_i : \langle S_I \rangle \{A_i\}.G_i\}_{i \in I}$*) there exists* $j \in I$ *and node* $(\overline{\mathtt{n}}, j, !)$ *in the graph tree of G that is satisfiable and such that* $\mathsf{end} \in G_j$*.*

Condition (2) ensures that, in case of branching, at least one branch is satisfiable, and that there is always at least one satisfiable path to end. Note that j can be equal to i. Also note that if a branch i cannot be taken then the node associated with the receive/branching $(\overline{\mathtt{n}}, i, ?)$ is trivially satisfiable as its node constraint is false.

Step 5 : checking for infinite satisfiability of nodes in recursion bodies Although in the chapter we assume infinite satisfiability, we provide here a method to explicitly check it. Namely, we check that all recursions in G are infinitely satisfiable (see Definition 3.6.8).

First, we give a general characterisation of constraints that make recursive types possibly unsatisfiable, if clocks are not reset. These constraints, which we will call *bad constraints*, are either unrepeatable constraints (e.g., equalities) or upper-binding constraints.

Definition 3.6.7 (Bad constraint)**.** *A constraint* δ *is bad, written* $\mathcal{B}(\delta)$*, if one of the following conditions hold:*

$$\begin{array}{l}\delta \text{ is of the form } x = e \text{ or of the form } x < e\\ \delta = \delta_1 \wedge \delta_2 \text{ and } \mathcal{B}(\delta_i) \text{ for some } i \in \{1,2\}\\ \delta = \delta_1 \vee \delta_2 \text{ and } \mathcal{B}(\delta_i) \text{ for both } i \in \{1,2\}\end{array}$$

We require that, in presence of bad constraints in a recursion loop, the clock is reset at least once within each possible computation in the recursive scope.

Definition 3.6.8 (Infinitely satisfiable G)**.** *Let* $p = \mathtt{n}, p'$ *be a path in* G *such that* $\mathtt{term}_G(\mathtt{n}) = G'$*. We say that* p *is infinitely satisfiable if either of the following holds:*

1. *for all* $\mathtt{n}' \in p$, $\mathtt{resInfo}(\mathtt{n}') = \emptyset$ *and* $\neg\mathcal{B}(\mathtt{const}(\mathtt{n}'))$*,*
2. *for all* $\mathtt{p} \in \mathcal{P}(G')$ *there exists* $\mathtt{n}' \in p$ *such that subj*$(\mathtt{n}') = \mathtt{p}$ *and* $\mathtt{resInfo}(\mathtt{n}') \neq \emptyset$

We say that all recursions in G are infinitely satisfiable if each path $\mathtt{nodes}(G'), \ldots, \mathbf{t}$ such that $\mu\mathbf{t}.G' \in G$ is infinitely satisfiable.

Definition 3.6.9 (Feasibility checker)**.** *To check for feasibility of a G: (1-2) generates the time graph for the all-unfolding of G, (3) annotates the nodes of the graph with their dependency constraints and resets, (4) checks that the time graph is satisfiable, and (5) checks that all recursions in G are infinitely satisfiable. We write* $\vdash_{pe} G$ *if G satisfies the checker.*

Wait-freedom. Global Satisfiability can be checked following a similar approach as for Progress Enablement(steps 1,2,3,4) but substituting Definition 3.6.5 (satisfiable node) with Definition 3.6.10 (wait-free node).

Definition 3.6.10 (Wait-free node)**.** *Let* $\mathtt{n}_! = (\overline{\mathtt{n}}, i, !)$ *and* $\mathtt{n}_? = (\overline{\mathtt{n}}, i, ?)$ *be nodes of the (all-unfolding) time graph of G, subj*$(\mathtt{n}_?) = \mathtt{p}$*. We say that* $\mathtt{n}_?$ *is* wait-free *if:*

$$\begin{cases}\delta_{\mathtt{n}_?}(\tilde{x}) \wedge (\delta(x) + R(\mathtt{n}_?, \mathtt{p})) \supset (\tilde{x} \leq x) & \text{if } \mathtt{const}(\mathtt{n}_?) = \delta\\ \mathsf{true} & \text{otherwise}\end{cases}$$

3.6.5. Time-Progress for Processes

We study the conditions under which a validated program is guaranteed to proceed until the completion of all activities of the protocols it implements. We define an *erasure* function for timed processes: The erasure $\texttt{erase}(P)$ is defined inductively by removing the delays in P (i.e., $\texttt{erase}(\texttt{delay}(t).P) = \texttt{erase}(P)$), while leaving unchanged the untimed parts (e.g., $\texttt{erase}(\overline{u}[\texttt{n}](y).P) = \overline{u}[\texttt{n}](y).\texttt{erase}(P)$). The other rules are homomorphic. Under the assumption that $\texttt{erase}(P)$ can progress, we investigate under which conditions P is also ensured to progress.

Definition 3.6.11 (Time erasure)**.** *The erasure* $\texttt{erase}(P)$ *of time from a process* P *is defined inductively as follows:*

$$
\begin{array}{lcl}
\texttt{erase}(\overline{u}[\texttt{n}](y).P) & = & \overline{u}[\texttt{n}](y).\texttt{erase}(P) \\
\texttt{erase}(u[\texttt{i}](y).P) & = & u[\texttt{i}](c).\texttt{erase}(P) \\
\texttt{erase}(c[\texttt{p}] \triangleleft l\langle e\rangle; P) & = & c[\texttt{p}] \triangleleft l\langle e\rangle; \texttt{erase}(P) \\
\texttt{erase}(c[\texttt{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I}) & = & c[\texttt{p}] \triangleright \{l_i(z_i).\texttt{erase}(P_i)\}_{i\in I} \\
\texttt{erase}(c[\texttt{p}] \triangleleft \langle\langle c'\rangle\rangle; P) & = & c[\texttt{p}] \triangleleft \langle\langle c'\rangle\rangle; \texttt{erase}(P) \\
\texttt{erase}(c[\texttt{p}] \triangleright ((y)).P) & = & c[\texttt{p}] \triangleright ((y)).\texttt{erase}(P) \\
\texttt{erase}(\texttt{delay}(t).P) & = & \texttt{erase}(P) \\
\texttt{erase}(\texttt{if}\, e\, \texttt{then}\, P\, \texttt{else}\, Q) & = & \texttt{if}\, e\, \texttt{then}\, \texttt{erase}(P)\, \texttt{else}\, \texttt{erase}(Q) \\
\texttt{erase}(P \mid Q) & = & \texttt{erase}(P) \mid \texttt{erase}(Q) \\
\texttt{erase}(0) & = & 0 \\
\texttt{erase}((\nu n)P) & = & (\nu n)\texttt{erase}(P) \\
\texttt{erase}(\mu X.P) & = & \mu X.\texttt{erase}(P) \\
\texttt{erase}(X) & = & X \\
\texttt{erase}(s : h) & = & s : h
\end{array}
$$

Definition 3.6.12 (Non-deferrable process)**.** *A process is non-deferrable if no action ready to execute in any of its threads is a time action, namely if* $\texttt{defr}(P) =$

true *with* $\texttt{defr}()$ *defined as follows:*

$$
\begin{array}{lcl}
\texttt{defr}(\overline{a}[\texttt{n}](y).P) & = & \text{false} \\
\texttt{defr}(a[\texttt{i}](y).P) & = & \text{false} \\
\texttt{defr}(s[\texttt{q}] \triangleleft l\langle e\rangle; P) & = & \text{false} \\
\texttt{defr}(s[\texttt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I}) & = & \text{false} \\
\texttt{defr}(s[\texttt{q}] \triangleleft \langle\langle s'[\texttt{p}']\rangle\rangle; P) & = & \text{false} \\
\texttt{defr}(s[\texttt{q}] \triangleright ((y)).P) & = & \text{false} \\
\texttt{defr}(\texttt{delay}(t).P) & = & \textit{if } t > 0 \textit{ then } \text{true} \textit{ else } \texttt{defr}(P) \\
\texttt{defr}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) & = & \text{false} \\
\texttt{defr}(P \mid Q) & = & \texttt{defr}(P) \vee \texttt{defr}(Q) \\
\texttt{defr}(0) & = & \text{true} \\
\texttt{defr}((\nu n)P) & = & \texttt{defr}(P) \\
\texttt{defr}(\mu X.P) & = & \text{false} \\
\texttt{defr}(X) & = & \text{false} \\
\texttt{defr}(s:h) & = & \text{true}
\end{array}
$$

We say that P is non deferrable *if* $\texttt{defr}(P) = \text{false}$.

Lemma 3.6.13. *If* $\texttt{erase}(P) \longrightarrow$ *then* $\texttt{erase}(P)$ *is of one of the following forms:*

1. $\overline{a}[\texttt{n}](y).P_1 \mid \prod_{i\in\{2,\ldots,n\}} a[\texttt{i}](y).P_i$
2. $s[\texttt{q}] \triangleleft l\langle e\rangle; P \mid s : h$
3. $s[\texttt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \mid s : (\texttt{p}, \texttt{q}, l_j\langle S_j\rangle) \cdot h \quad (j \in I)$
4. $s[\texttt{q}] \triangleleft \langle\langle s'[\texttt{p}']\rangle\rangle; P \mid s : h$
5. $s[\texttt{q}] \triangleright ((z)).P \mid s : (\texttt{p}, \texttt{q}, s'[\texttt{p}']) \cdot h$
6. $\texttt{if } e \texttt{ then } P \texttt{ else } Q$
7. $P \mid Q$ *with* P *and* Q *non-deferrable*
8. $(\nu n)P$

Lemma 3.6.14. *1. if* $\texttt{erase}(P) = \overline{a}[\texttt{n}](y).P_1 \mid \prod_{i\in\{2,\ldots,n\}} a[\texttt{i}](y).P_i$, *then* $\exists P'_1, \ldots, P'_n$, $t_1 \geq 0, \ldots, t_n \geq 0$ *such that* $P \equiv \texttt{delay}(t_1).\overline{a}[\texttt{n}](y).P'_1 \mid \prod_{i\in\{2,\ldots,n\}} \texttt{delay}(t_i).a[\texttt{i}](y).P'_i$ *and* $P_i = \texttt{erase}(P'_i)$ *for all* $i \in 1, \ldots, n$.

2. *if* $\texttt{erase}(P) = s[\mathtt{q}] \triangleleft l\langle e\rangle; P_1 \mid s : h$ *then* $\exists P_1', t \geq 0$ *such that* $P \equiv \texttt{delay}(t).s[\mathtt{q}] \triangleleft l\langle e\rangle; P_1' \mid s : h$ *and* $P_1 = \texttt{erase}(P_1')$.

3. *if* $\texttt{erase}(P) = s[\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle v\rangle) \cdot h \quad (j \in I)$ *then* $\exists P_i', t \geq 0$ *for* $i \in I$ *such that* $P \equiv \texttt{delay}(t).s[\mathtt{q}] \triangleright \{l_i(z_i).P_i'\}_{i\in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle v\rangle)$ *and* $P_i = \texttt{erase}(P_i')$ *for all* $i \in I$.

4. *if* $\texttt{erase}(P) = s[\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P_1 \mid s : h$ *then* $\exists P_1', t \geq 0$ *such that* $P \equiv \texttt{delay}(t).s[\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P_1' \mid s : h$ *and* $P_1 = \texttt{erase}(P_1')$.

5. *if* $\texttt{erase}(P) = s[\mathtt{q}] \triangleright ((z)).P_1 \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h$ *then* $\exists P_1', t \geq 0$ *such that* $P \equiv \texttt{delay}(t).s[\mathtt{q}] \triangleright ((z)).P_1' \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h$ *and* $P_1 = \texttt{erase}(P_1')$.

6. *if* $\texttt{erase}(P) = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ *then* $\exists P_1', P_2', t \geq 0$ *such that* $P \equiv \texttt{delay}(t).\texttt{if}\ e\ \texttt{then}\ P_1'\ \texttt{else}\ P_2'$ *and* $P_i = \texttt{erase}(P_i')$ *for* $i \in \{1,2\}$.

7. *if* $\texttt{erase}(P) = P_1 \mid P_2$ *then* $\exists P_1', P_2', t_1 \geq 0, t_2 \geq 0$ *such that* $P \equiv \texttt{delay}(t_1).P_1' \mid \texttt{delay}(t_2).P_2'$ *with* $P_i \equiv \texttt{erase}(P_i')$ *and* P_i' *non-deferrable for* $i \in \{1,2\}$.

8. *if* $\texttt{erase}(P) = (\nu n)P'$ *then* $\exists P'', t_{\geq} 0$ *such that* $P \equiv \texttt{delay}(t).(\nu)P''$ *and* $\texttt{erase}(P'') = P'$.

Proof. The proof is mechanical by Lemma 3.6.13. □

In case (7) of Lemma 3.6.14 observe that we can always find non deferrable continuations P_1' and P_2' as we can collect the delays in front of the process by structural equivalence rule $\texttt{delay}(t).\texttt{delay}(t').P_i \equiv \texttt{delay}(t+t').P_i$.

Lemma 3.6.15. *If P is non-deferrable and* $\texttt{erase}(P) \longrightarrow P'$ *then* $P \longrightarrow P''$ *with* $\texttt{erase}(P'') = P'$.

Proof. Mechanical, by induction on the derivation of $\texttt{erase}(P)$, proceeding by case analysis using Lemma 3.6.13 and Lemma 3.6.14. □

Proposition 3.6.16 (Conformance)**.** *If* $P \longrightarrow P'$ *then* $\texttt{erase}(P) \longrightarrow^* \texttt{erase}(P')$.

We say that: (1) P is a *deadlock process* if $P \longrightarrow^* P'$ where $P' \not\longrightarrow$ and $P' \not\equiv 0$, and (2) P is session delay if for each delay process of the form $\texttt{delay}(t).P'$ occurring in P, there are no session request and session accept in P'. We say that Γ is feasible (resp. wait-free) if for all $u \in dom(\Gamma)$, $\Gamma(u)$ is feasible (resp. wait-free).

Lemma 3.6.17 (Time progress - parallelism and interleaved sessions)**.** *Let* Γ *be a* feasible *and* wait free *mapping,* $\Gamma \vdash P_0 \triangleright \emptyset$*, and* $P_0 \longrightarrow^+ P$ *where* P *is session delay and*

$$P \equiv \texttt{delay}(t).P \mid R \qquad \textit{for some } t > 0 \textit{ and non-deferrable } R$$

If $\texttt{erase}(P)$ *is not a deadlock process and* $\texttt{erase}(P) \longrightarrow$ *then* $R \longrightarrow$.

Proof. Without loss of generality we assume that for some L

$$R = R' \mid \prod_{l \in L} s_l : h_l$$

We proceed by induction on the complexity of R', proceeding by case analysis on the possible form of process R'.

The complex cases are those for receive and session receive, (i.e., R' is stuck waiting for messages). In all other cases one can immediately show that R' reduces. In fact,

- R' cannot be a of the form $\overline{u}[\texttt{n}](y).R''$ or $u[\texttt{i}](y).R''$ since (1) $\texttt{erase}(P)$ is deadlock free hence all the other corresponding session requests or accepts to start session y are in P, and (2) P is session delay hence none of the corresponding session accepts/delay can be in P_i.
- R' cannot be a delay process as P it is non deferrable (Definition 5.6.3) by hypothesis,
- if R' is of the form $s[\texttt{q}] \triangleleft l\langle e \rangle; R''$ or $s[\texttt{q}] \triangleleft \langle\langle c' \rangle\rangle; R''$ then one can immediately show that R can reduce by $\lfloor \textsc{Sel} \rfloor$ or $\lfloor \textsc{Srec} \rfloor$.
- in the inductive cases (e.g., for hiding and parallel processes) the result follows by induction.

We will show the case for R' being an input process. The case for session receive is similar hence omitted.

Case $R' \equiv s[\texttt{p}][\texttt{q}] \triangleright \{l_k(z_k).R_k\}_{k \in I}$. In this case:

$$R \equiv s[\texttt{p}][\texttt{q}] \triangleright \{l_k(z_k).R_k\}_{k \in I} \mid \prod_{l \in L} s_l : h_l$$

We proceed by contradiction assuming Γ feasible and wait-free, $\texttt{erase}(P)$ deadlock free, $\texttt{erase}(P) \longrightarrow$ and $R \not\longrightarrow$. If $R \not\longrightarrow$ we have the following cases

depending on where the corresponding output of $s[\mathtt{p}][\mathtt{q}] \triangleright \{l_k(z_k).R_k\}_{k\in I}$ is:

1. the corresponding output appears (immediately or after some other input) in some continuation R_k,
2. the corresponding output appears in P_i for some $i \in I$
3. the corresponding output does not appear.

If (1) or (3) then we contradict that $\mathtt{erase}(P)$ is deadlock free. If (2) then we contradict the fact P_0 was validated against a feasible and wait-free Γ in fact: feasibility ensures that there exists at least one execution of the types of sessions of s_i where a selection/delegate happens before or at the same time of the corresponding receive/session receive, and by wait-freedom since in one execution the output does not precede the input then in all correct implementations the output does not precede the input, hence the output for a non deferrable (Definition 5.6.3) input does not occur after $t_i > 0$ time which contradicts fact that the output appears in P_i. □

Theorem 3.6.18 (Timed progress in interleaved sessions)**.** *Let* Γ *be a* feasible *and* wait-free *mapping,* $\Gamma \vdash P_0 \triangleright \emptyset$*, and* $P_0 \longrightarrow^+ P$*. If* P *is session delay,* $\mathtt{erase}(P)$ *is not a deadlock process and if* $\mathtt{erase}(P) \longrightarrow$ *then* $P \longrightarrow$.

Proof. We prove Theorem 3.6.18 by induction on the proof of P, proceeding by case analysis on the possible forms of P according to Lemma 3.6.13.

Case Lemma 3.6.13 (1): Since we assume that P is session delay then this case can only occur if $P = \mathtt{erase}(P)$, hence the thesis follows immediately by the hypothesis.

Cases Lemma 3.6.13 (2), (3), (4), and (5) proceed similarly. We show the case for Lemma 3.6.13 (3): $\mathtt{erase}(P) = s[\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle S_j\rangle) \cdot h \quad (j \in I)$. By Lemma 3.6.14 (3)

$$P \equiv \mathtt{delay}(t).\mathtt{erase}(P)[P_i'/P_i] \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle S_j\rangle) \cdot h \quad (j \in I) \tag{3.44}$$

By two reductions by rule $\lfloor$Str$\rfloor$ with premises by rules $\lfloor$Time$\rfloor$ and then $\lfloor$Bra$\rfloor$ on (3.44)

$$P \longrightarrow \mathtt{erase}(P)[P_i'/P_i] \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle S_j\rangle) \cdot h \longrightarrow P_j \mid s : h \tag{3.45}$$

and by reduction rule $\lfloor$Bra$\rfloor$ on $\mathtt{erase}(P)$

$$\mathtt{erase}(P) \longrightarrow \mathtt{erase}(P_j) \mid s : h \tag{3.46}$$

Finally, by (3.45) and (3.46), since

$$\texttt{erase}(P_j) \mid s : h = \texttt{erase}(P_j \mid s : h)$$

we have the thesis for this case.

Case Lemma 3.6.13 (6): immediate considering that $e \downarrow \mathsf{true}$ in both P and $\texttt{erase}(P)$ (the erasure does not affect the conditions of conditional statements as such conditions do not involve delays).

Case Lemma 3.6.13 (7): $\texttt{erase}(P) = P_1 \mid P_2$ and $\texttt{erase}(P) \longrightarrow P_1' \mid P_2$. By reduction rule $\lfloor$Com$\rfloor$

$$P_1 \longrightarrow P_1' \tag{3.47}$$

By Lemma 3.6.14 (7)

$$P \equiv \texttt{delay}(t_1).Q_1 \mid \texttt{delay}(t_2).Q_2$$

$$\texttt{erase}(Q_i) = P_i,\ i \in \{1,2\} \tag{3.48}$$

and

$$Q_i \textit{ non-deferrable},\ i \in \{1,2\} \tag{3.49}$$

If $t_1 = t_2$ then $\texttt{delay}(t_1).Q_1 \mid \texttt{delay}(t_2).Q_2 \longrightarrow Q_1 \mid Q_2$ by a $\lfloor$Time$\rfloor$ reduction. By (3.48), (3.49) and (3.47) $Q_1 \longrightarrow$ hence by $\lfloor$Com$\rfloor$

$$Q_1 \mid Q_2 \longrightarrow Q_1' \mid Q_2$$

hence done.

If $t_1 \neq t_2$, assume without loss of generality that $t_1 > t_2$. After a $\lfloor$Time$\rfloor$ reduction

$$\texttt{delay}(t_1).Q_1 \mid \texttt{delay}(t_2).Q_2 \longrightarrow \texttt{delay}(t_1 - t_2).Q_1 \mid Q_2$$

with Q_2 *non-deferrable*. This case holds by Lemma 3.6.17

Case Lemma 3.6.13 (8): immediate by induction.

□

Note that if P implements a single session then $\texttt{erase}(P)$ is deadlock free [CDCYP15]; this yields the following corollary. We characterise processes implementing single sessions, or *simple*, as follows: a process P is simple if $P_0 \longrightarrow^* P$ for some P_0 such that $a : \langle G \rangle \vdash P_0 \triangleright \emptyset$, and $P_0 = \overline{a}[\mathtt{n}](y).P_1 \mid$

$\prod_{i\in\{2,..,n\}} a[\mathtt{i}](y).P_i$ where $P_1,\ldots,P_n$ do not contain any name hiding, request/accept, and session receive/delegate.

Corollary 3.6.19 (Time progress in single sessions)**.** *Let G be* feasible *and* wait-free*, and P be a* simple process *with* $a : \langle G\rangle \vdash P \triangleright \emptyset$*. If* $\mathtt{erase}(P) \longrightarrow$*, then there exists P' such that* $\mathtt{erase}(P) \longrightarrow P'$ *and* $P \longrightarrow^+ P''$ *such that* $\mathtt{erase}(P'') = P'$.

Proof. Since P is simple, by the Proposition of progress for a *simple* process in [HYC08], P has no delegation input or output. Thus P is a *session delay* process. By Theorem 3.6.18, the corollary is proved. □

3.7. Related Works

Calculi with Time

A few proposed timed π-calculi or other kinds of timed calculus are reviewed. In this section, we list some of the typical references. Since we focus on how to model time into session types/session calculus, here we present the ways in which these researchers model time/time constraints in their works.

The authors of [LZ02] propose a process algebra for modelling real-time specifications of mobile systems, which is called π RT-calculus. They combine existing research on static, real-time processes with Milner's π-calculus, and they aim to use the π RT-calculus to specify and reason about dynamic temporal behaviour and dynamic configuration of concurrent systems as simply and expressively as possible so as to achieve a minimal extension to the original π-calculus. The only change is that it introduces a *'timeout operator'*, denoted as $P \rhd^t Q$, which behaves as P before t time units and as Q at and after t time units. They also distinguish when an action is possible (*enabled*) and when an action happens (*fired*), because of possible delay between these two states.

The work in [FFW$^+$04] presents a dense-time extension of π-calculus, on the basis of time consuming transitions. Similar to the previous one, the authors try to minimise the changes to the original π-calculus while maintaining simplicity and expressiveness for modelling concurrent real-time systems. In the proposed semantics, the authors introduce a *'time guard'* denoted as $\triangle$ that specifies a time interval before a process can proceed. The time guard has the format $[t_{min}, t_{max}]$, where t_{min} and t_{max} are the time lower and upper bound for reaching the target state. Also, the authors define a timed labelled transition system (TLTS), whose

solutions result in two time sets **ACTIVE** (the set of incoming transitions of state S) and **ARRIVE** (the set of time at which the target state of a transition can be reached).

In the paper [LZ05], the authors define a timed extension of asynchronous π-calculus with a transaction construct, called the calculus webπ, for describing protocols and interfaces of web services. Despite the lack of an agreement on web transaction standard, the authors propose webπ that models the main issues of web transactions, that is, interruptible processes, failure handler and time. The important part in the syntax of webπ is the web transaction construct, denoted as $\lhd P;Q \rhd_x^n$, where P is the body, Q is a compensation, n means time steps and x is the abort message, which indicates the transaction is discarded. The construct means that it will behave as P if it does not terminate and it will perform as Q after n time steps or if an abort message x is received.

The authors [BY07] propose types of a distributed timed π-calculus and an approximate bisimulation that could be used as a tool for analysing probabilistic processes. In their typing, the time information is modeled as a 'timer construct' denoted as $\text{timer}^t\langle a(\tilde{x}).P,Q\rangle$, with $t > 0$. It has two functions: (1) timeout, which means that after t time units it turns into Q, unless (2) it has been stopped, that is, a message has been received by the timer at a. The flow of time is achieved by a timestepper function φ, which acts on processes. It models the implicit broadcast of time passing and its two main rules are:

$$\varphi(\text{timer}^{t+1}\langle a(\tilde{x}).P,Q\rangle) = \text{timer}^t\langle a(\tilde{x}).P,Q\rangle \qquad \varphi(\text{timer}^1\langle a(\tilde{x}).P,Q\rangle) = Q$$

This typing semantics enables all existing typing rules unchanged without additional typing. Later the authors incorporate probabilities of message loss to calculus and propose approximate bisimulation, which ignores the transitions that are below the probability threshold. This bisimulation is proved to be a congruence and can be used for verification of protocols in distributed systems, for example, verifying an RFC protocol as presented in the paper.

The previously stated works are similar that they are all extended π-calculi with time constructs. Compared with above research, we directly achieve the extension on multiparty session types. In this way, we pass the responsibility of expressing timing actions to global types, not in asynchronous π-calculus as the above works do; and our timed session calculus only shows the time passing. The benefits of directly extending multiparty session types are: (1) we can check feasibility

and wait-freedom easily from timed global types; (2) we can guarantee the type-safety and time error-freedom for timed processes and (3) we can describe real-time protocols in timed global types in a simple and elegant way. In addition, reset operations in our time assertions enable the timed session types to be more expressive than those without resets.

The authors in [LP11] propose the basic semantics of C^3, an extension of the Conversation Calculus (CC) [VCS08, Vie10] that provides an explicit treatment of time issues and exceptional behaviour in structured communications. The CC is a process calculus that addresses issues in service-oriented computations, such as distribution, context sensitiveness and loose coupling. It is observed that in the specification of practical instances of structured communications, timed behaviour is often accompanied with exceptional behaviour; thus it is necessary to include the behavioural pattern that consists of a default behaviour and a compensational behaviour in case of unexpected conditions. Furthermore, the authors extend the original CC by replacing the simple conversation context with a timed, compensable conversation context denoted as $n \blacktriangleleft [P;Q]^t_\kappa$. Here n denotes the identity of conversation context and process P is the default behaviour of n. P is observable as long as the duration t is greater than 0; as soon as $t = 0$, the compensating behaviour Q will replace the default behaviour. Name κ means an abort mechanism; receiving the *kill* signal sets t to zero at once. Through examples, the authors prove that C^3 achieves the function of implicit conversation abortion related with time passing. One of the drawbacks, as authors have mentioned, is that the encoding of this method is untyped; thus the design should be more careful.

COWS (*Calculus for Orchestration of Web Services*) [LPT07] is a new language for modelling different and typical aspects in Service Oriented Computing (SOC), such as receive conflict and routeing of correlated messages. However, it turns out that timeout, which is not implemented by COWS, is important for dealing with service transactions or message losses. Motivated by closing this gap, the authors extend COWS with timed orchestration constructs to model the timed-related activities in SOC. In their extension language, the basic actions are instantaneous and time passing is modelled as explicit actions, called *wait activity*. The wait activity $\oplus_e$ gives the time interval, whose value is given by the evaluation e, the execution service has to wait for. The authors illustrate the effectiveness of the semantics through three application examples. The problem of this proposed extended language is that it lacks the expressions of its corresponding labelled transition systems, behavioural equivalence and typing systems, thus it makes the

language seem incomplete for practical use.

In [LOP09], the authors propose a unified framework as a declarative approach for modelling and reasoning about SOC systems. This approach takes a set of time condition components into consideration instead of only focusing on the specification of flow controls within processes. The authors base their research on *utcc*, a timed Concurrent Constraint Programming process algebra [OV08], in which time is divided into time units or intervals. The syntax of processes includes some timed-related operations: $(\textbf{local}\ \tilde{x}, c)P$ means binding the variables $\tilde{x}$ in P under a constraint c and *unit delay* **next** P indicates execution of P in the next time interval. The authors introduce their framework by combining *utcc* with language primitives by Honda [HVK] and providing the encoding of the original session calculus in [HVK] into *utcc*. Then they describe a timed extension of the calculus and present the encoding of this timed extension into *utcc*. The case study of electronic booking illustrates the usage of proposed framework. One disadvantage of this framework is that time extension only involves session engagement processes; the time requirements within the session cannot be modelled in this framework.

The above three research works extend other kinds of calculi with time. Compared with timed multiparty session types, though practical implementations of their theory are presented, the above extensions suffer similar problems as those based on π-calculus, such as less expressiveness and no guarantee for type-safety. In addition, recent work [NY17] shows that timed multiparty session types can be applied to build a dynamic verification framework for real-time systems based on Scribble.

Feasibility

Feasibility was first introduced in a different context in [AFK87], as a necessary requirement of the definition of fairness in the study of nondeterministic systems. This paper gives a concrete definition in the context of real-time interactions and states its decidability for infinitely satisfiable *TG*s. [Tri99] models distributed timed systems as a parallel combination of timed automata, and gives an algorithm to check deadlock freedom and time progress for timed automata. The algorithm, based on syntactic conditions on the states relying on invariant annotations, is not directly applicable to checking feasibility e.g., on the timed automaton derived from a *TG*.

3.8. Summary

This chapter presents the theory of multiparty session types extended with time. By introducing time constraints and reset as predicates, timed global types can model real-time communicating protocols with choices and recursion. Timed local types are the projection of global types to each participant in the communication systems. Labelled transition systems for global types and local types are defined to describe the behaviours and traces of the program. Then the theory of timed session types is attested in π-calculus. We propose a decidable proof system for π-calculus processes that ensures type safety and time-error freedom. Two conditions, feasibility and wait-freedom, are defined and with the theorem of time progress, we can verify progress property of processes.

4. Choreographical Multiparty Session Types

In this chapter, we introduce new choreographical multiparty (global and local) session types that solve open problems of expressiveness and algorithmic projection posed in [CDCP11]. We develop a new typing system for multiparty session mobile processes generalised with choice, fork, merge and join constructs, and prove that typed processes conform to the type safety property. We also define the conditions for well-formedness of choreographical global types to ensure that global types are well-behaved.

4.1. Motivation: Fork and Join

The theory of multiparty session types proposed in [CDCYP15] describes the value/channel exchange, branching and recursion behaviours for communicating protocols. However, in concurrent systems, especially in business use cases, it is common that one participant could multitask, communicating with multiple participants at one time. For example, a web server should be able to handle multiple requests from different clients simultaneously. This requirement of concurrent operations exposes the problem of expressiveness of multiparty session types; we lack constructs for describing parallel operations in multiparty session types. To address this problem, we present a new global type syntax which encompasses previous systems [HYC08, CDCYP15] with extended constructs (join and merge) and generalised graph syntax.

The main feature of these choreographical session types is to explicitly distinguish the branching points from the forking points (where concurrent, interleaved interaction can take place). Such a distinction is critical to avoid state explosion and to directly and efficiently type session-based languages and processes. In process calculus, the parallel operation can be decomposed into a choice: $a \mid b = a.b + b.a$. From this equation, we find that one state is expressed as

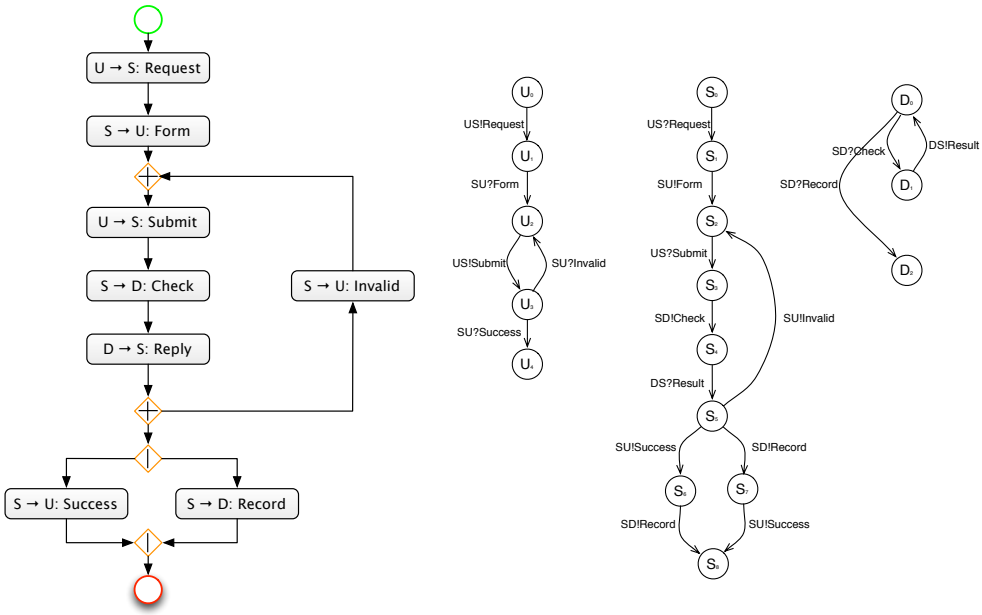


Figure 4.1.: Register Example: Global View in Activity Diagram (Left) v.s. Local View in State Transition Diagram (Right)

two states. If we have three parallel actions, then the decomposition is:

$$a \mid b \mid c = a.b.c + a.c.b + b.a.c + b.c.a + c.a.b + c.b.a$$

and then we summarise that if n actions are parallel, then the number of decomposed states is the permutation number $P(n) = n!$, which grows exponentially with n. Therefore, distinguishing fork/join with choice/merge could help avoid state explosion and describe parallel processes in a precise manner.

We illustrate with a simple scenario, named Register, shown in Figure 4.1. We show the example in two ways: from the global view, the example is represented as a BPMN-like [BPM] activity diagram, where '+' stands for exclusive gateways and '|' for fork/join, and interactions are presented in session type conventions; from the local view, we illustrate the scenario in state transition diagrams for each participant in the protocol (state transition diagrams are only used for illustration and their connection with local types is not discussed in this thesis).

This scenario comprehensively combines recursion, fork, join, choice and merge. It models a protocol that describes a membership registration process. This process involves three participants: a user `U` who wants to register as a member

of the website, a server S of the website that deals with the requests from users, and a remote database D that keeps records of membership. The user sends a registration request by clicking the *register* button on the web page, and the server, upon receiving the request, replies with a form for the user to fill in. After the user finishes and submits the form, the user information is sent to the website server and then the server communicates with the remote database to ask whether the application is valid or not. If the database reports the application with problems, such as the email address has been registered or the name has been occupied, then the server will inform the user that this registration is not valid and report where the problem is. Then the registration loops until the submitted application is valid. In that case, the server notifies the user that the registration is successful and simultaneously requires the database to store the user information.

4.2. Global Types for Choreographical Multiparty Sessions

In this section, we present the syntax of global choreographical session types and illustrate with several examples. We also present the well-formedness conditions, which ensure the global types are well-behaved.

4.2.1. Syntax of Global Types

We introduce new *choreographical global types*, which improve the expressiveness of multiparty session types.

Definition 4.2.1 (Choreographical Global Types)**.** The syntax of global types is defined below.

$\mathbf{G}$	::=	$\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}$	Global type
G	::=	$\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle ; \mathbf{x}'$	Labelled messages
	|	$\mathbf{x} = \mathbf{x}' \mid \mathbf{x}''$	Fork
	|	$\mathbf{x} = \mathbf{x}' + \mathbf{x}''$	Choice
	|	$\mathbf{x} \mid \mathbf{x}' = \mathbf{x}''$	Join
	|	$\mathbf{x} + \mathbf{x}' = \mathbf{x}''$	Merge
	|	$\mathbf{x} = \mathsf{end}$	End
U	::=	$\langle \mathbf{G} \rangle \mid \mathsf{bool} \mid \mathsf{nat} \mid \cdots$	Sorts

Although it is based on multiparty session frameworks, choreographical global types are different from the basic syntax of global types in three aspects.

1. We define flexible fork, choice, merge and join operations in global session types. Distinguishing fork and choice in global expressions brings two benefits: when analysing the behaviours of participants by transition diagram, we could save state space state and present the interactions in clear and neat graphs; we gain precise thread management because we can distinguish mutually exclusive threads, which are presented in choices, and independent threads, which are shown in pairs of fork/join.

2. We introduce the *state variables* $\mathbf{x}$ in global types to represent the successive states of the interactions. These state variables not only help express flexible operations (so that we could write complex protocols that are hard to describe in the syntax of basic session types), but also facilitate the process of generating state transition diagrams in the global view.

3. Recursion notation is removed but we could express recursion in flexible choice/merge pairs.

A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ describes a set of interactions among the participants in the protocol. It is noted that $=$ in the global syntax represents a state equivalence relation, not equality. The communication starts from $\mathbf{x}_0$, denoted as the *initial state*, and continues according to the interactions specified in $\widetilde{G}$. The interactions are modelled as *labelled message exchanges* $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle;\mathbf{x}'$. This transition describes that the sender $\mathtt{p}$ can go from state $\mathbf{x}$ to the continuation $\mathbf{x}'$ by sending message l with type U, and the receiver $\mathtt{p}'$ goes from $\mathbf{x}$ to $\mathbf{x}'$ upon receiving it. All other uninvolved participants can go from $\mathbf{x}$ to $\mathbf{x}'$ for free. Sort types S include channel types $\langle\mathbf{G}\rangle$ or base data types, and message types U are either sort types S or local types $\mathcal{T}$ for delegation. $\mathbf{x} = \mathbf{x}' + \mathbf{x}''$ represents the branching made by the sending participant, between continuation with state $\mathbf{x}'$ or $\mathbf{x}''$. The merging construct $\mathbf{x}' + \mathbf{x}'' = \mathbf{x}$ shows that the choices are closed, that is, two mutually exclusive branches share the same continuation. $\mathbf{x} = \mathbf{x}' \mid \mathbf{x}''$ expresses forking the two concurrent interactions, and allowing the interleaved actions in $\mathbf{x}'$ and $\mathbf{x}''$. Similarly, these forking threads are combined by the joining operation $\mathbf{x}' \mid \mathbf{x}'' = \mathbf{x}$. $\mathbf{x} = \mathsf{end}$ denotes session termination.

Example 4.2.2 (Global Types of the Register Example)**.** We show the global types of Register example in Section 4.1 below.

$$
\begin{array}{rcl}
\mathbf{G}_{\text{Register}} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0 & & \\
\mathbf{x}_0 & = & \mathtt{U} \to \mathtt{S} : \mathit{Request}\langle \mathsf{nat} \rangle ; \mathbf{x}_1 \\
\mathbf{x}_1 & = & \mathtt{S} \to \mathtt{U} : \mathit{Form}\langle \mathsf{unit} \rangle ; \mathbf{x}_2 \\
\mathbf{x}_2 + \mathbf{x}_9 & = & \mathbf{x}_3 \\
\mathbf{x}_3 & = & \mathtt{U} \to \mathtt{S} : \mathit{Submit}\langle \mathsf{string} \rangle ; \mathbf{x}_4 \\
\mathbf{x}_4 & = & \mathtt{S} \to \mathtt{D} : \mathit{Check}\langle \mathsf{string} \rangle ; \mathbf{x}_5 \\
\mathbf{x}_5 & = & \mathtt{D} \to \mathtt{S} : \mathit{Reply}\langle \mathsf{nat} \rangle ; \mathbf{x}_6 \\
\mathbf{x}_6 & = & \mathbf{x}_7 + \mathbf{x}_8 \\
\mathbf{x}_7 & = & \mathtt{S} \to \mathtt{U} : \mathit{Invalid}\langle \mathsf{nat} \rangle ; \mathbf{x}_9 \\
\mathbf{x}_8 & = & \mathbf{x}_{10} \mid \mathbf{x}_{11} \\
\mathbf{x}_{10} & = & \mathtt{S} \to \mathtt{U} : \mathit{Success}\langle \mathsf{bool} \rangle ; \mathbf{x}_{12} \\
\mathbf{x}_{11} & = & \mathtt{S} \to \mathtt{D} : \mathit{Record}\langle \mathsf{string} \rangle ; \mathbf{x}_{13} \\
\mathbf{x}_{12} \mid \mathbf{x}_{13} & = & \mathbf{x}_{14} \\
\mathbf{x}_{14} & = & \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

We illustrate the syntax in the Register example shown in Example 4.2.2. $\mathbf{G}_{\text{Register}}$, the *global type*, represents the protocol in the form of choreography that describes the interactions between participants and not merely a collection of local behaviours. Initially, we write $\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ where $\widetilde{G}$ represents the transitions between state variables, and where $\mathbf{x}_0$ is denoted as the initial state of all the participants. A transition of the form $\mathbf{x}_0 = \mathtt{U} \to \mathtt{S} : \mathit{Request}\langle \mathsf{nat} \rangle ; \mathbf{x}_1$ corresponds to an activity that U sends S a message *Request* carrying a value of type nat, which informs the websites that request is for registration. Then this activity is followed by the interactions that happen in $\mathbf{x}_1$. A transition $\mathbf{x}_6 = \mathbf{x}_7 + \mathbf{x}_8$ denotes a choice that is done by S. If the server finds that the application is invalid based on the *Reply* message from the database, then the server will inform the user about this invalid message. Then $\mathbf{x}_2 + \mathbf{x}_9 = \mathbf{x}_3$ denotes a merging point, which indicates the recursion by continuing with $\mathbf{x}_3$. If the registration request is passed, then the server is supposed to forward messages to user and database simultaneously. A transition $\mathbf{x}_8 = \mathbf{x}_{10} \mid \mathbf{x}_{11}$ describes such an interaction that should continue concurrently with the actions of $\mathbf{x}_{10}$ and of $\mathbf{x}_{11}$. In a symmetric way, the transition $\mathbf{x}_{12} \mid \mathbf{x}_{13} = \mathbf{x}_{14}$ corresponds to the join of two concurrent threads that reach points $\mathbf{x}_{12}$ and $\mathbf{x}_{13}$ respectively. After joining into a single thread $\mathbf{x}_{14}$, the protocol reaches the end point.

Example 4.2.3 (Choreographical Global Types)**.** We give more complicated examples in Figure 4.2, with their graph representation for the purpose of

illustrating the expressiveness of choreographical session types. More examples are available in Appendix A.1.

1. $\mathbf{G}_1$ shows a protocol that `Alice` recursively sends messages to `Bob` This example illustrates that the recursion is presented using merge construct.

2. $\mathbf{G}_2$ describes a choice between directly communicating with `Carol` or with `Bob` first. It is noted that the last interaction from `Carol` to `Bob` in $\mathbf{x}_6$ is important; without this transition, `Bob` could reach a deadlock by waiting forever for a message from `Alice`, if $\mathbf{x}_3$ is chosen.

3. $\mathbf{G}_3$ presents a use case called *the Alternating Bit Protocol*. `Alice` repeatedly sends `Bob` the alternating messages Msg_1 and Msg_2 but will concurrently wait for the acknowledgement Ack_i before sending another Msg_i. This protocol structure requires a general graph syntax and cannot be represented in existing session type framework. But it is representable in the choreographical global types.

4. $\mathbf{G}_4$ shows a simple example of the usage of forking followed by immediate joining. The possible traces of $\mathbf{G}_4$ are $\alpha\gamma\beta\delta$, $\beta\alpha\gamma\delta$, $\alpha\beta\gamma\delta$, $\beta\alpha\delta\gamma$ and $\alpha\beta\delta\gamma$. The traces represented by this global type are not easily representable by tree-based representation: it is easy to write with a standard syntax of the relation between messages α, β and δ: $(\alpha \mid \beta);\delta$; but γ should follow β without linking with α or δ. Only a graph-like syntax can represent such set of traces without introducing a choice-based enumeration.

5. $\mathbf{G}_5$ follows the structural shape of the $\mathbf{G}_3$, replaces forks by choices and joins by merges.

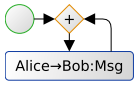
4.2.2. Well-formed Global Types

This subsection defines well-formedness of global types, which consists of four conditions: sanity, well-threadedness, local choice and linearity.

Sanity Conditions for global types should prevent possible syntactic confusions about which continuations to follow at given point. A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ satisfies the *sanity* conditions if the following three conditions are met.

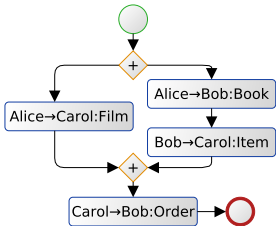
1. (**Unambiguity**) Every state variable $\mathbf{x}$ except $\mathbf{x}_0$ should appear exactly once on the left-hand side and once on the right-hand side of the transitions in $\widetilde{G}$.

1. $\mathbf{G}_1 = \text{def } \mathbf{x}_0 + \mathbf{x}_2 = \mathbf{x}_1$
$\mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}\langle \text{string} \rangle ; \mathbf{x}_2 \text{ in } \mathbf{x}_0$



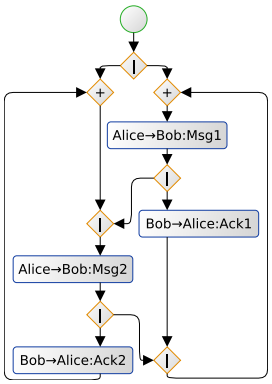
---

2. $\mathbf{G}_2 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_3$
$\mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Book}\langle \text{string} \rangle ; \mathbf{x}_2$
$\mathbf{x}_2 = \texttt{Bob} \to \texttt{Carol} : \mathit{Item}\langle \text{nat} \rangle ; \mathbf{x}_4$
$\mathbf{x}_3 = \texttt{Alice} \to \texttt{Carol} : \mathit{Film}\langle \text{string} \rangle ; \mathbf{x}_5$
$\mathbf{x}_4 + \mathbf{x}_5 = \mathbf{x}_6$
$\mathbf{x}_6 = \texttt{Carol} \to \texttt{Bob} : \mathit{Order}\langle \text{string} \rangle ; \mathbf{x}_7$
$\mathbf{x}_7 = \text{end in } \mathbf{x}_0$



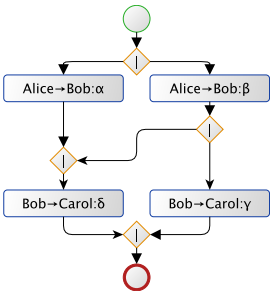
---

3. $\mathbf{G}_3 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$
$\mathbf{x}_1 + \mathbf{x}_3 = \mathbf{x}_4$
$\mathbf{x}_2 + \mathbf{x}_5 = \mathbf{x}_6$
$\mathbf{x}_4 = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_1\langle \text{string} \rangle ; \mathbf{x}_7$
$\mathbf{x}_7 = \mathbf{x}_8 \mid \mathbf{x}_9$
$\mathbf{x}_8 = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_1\langle \text{unit} \rangle ; \mathbf{x}_{10}$
$\mathbf{x}_6 \mid \mathbf{x}_9 = \mathbf{x}_{11}$
$\mathbf{x}_{11} = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_2\langle \text{string} \rangle ; \mathbf{x}_{12}$
$\mathbf{x}_{12} = \mathbf{x}_{13} \mid \mathbf{x}_{14}$
$\mathbf{x}_{13} = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_2\langle \text{unit} \rangle ; \mathbf{x}_5$
$\mathbf{x}_{10} \mid \mathbf{x}_{14} = \mathbf{x}_3 \text{ in } \mathbf{x}_0$



---

4. $\mathbf{G}_4 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$
$\mathbf{x}_1 = \texttt{A} \to \texttt{B} : \alpha\langle U \rangle ; \mathbf{x}_3$
$\mathbf{x}_2 = \texttt{A} \to \texttt{B} : \beta\langle U \rangle ; \mathbf{x}_4$
$\mathbf{x}_3 = \mathbf{x}_5 \mid \mathbf{x}_6$
$\mathbf{x}_4 \mid \mathbf{x}_5 = \mathbf{x}_7$
$\mathbf{x}_6 = \texttt{A} \to \texttt{B} : \gamma\langle U \rangle ; \mathbf{x}_8$
$\mathbf{x}_7 = \texttt{A} \to \texttt{B} : \delta\langle U \rangle ; \mathbf{x}_9$
$\mathbf{x}_8 \mid \mathbf{x}_9 = \mathbf{x}_{10}$
$\mathbf{x}_{10} = \text{end in } \mathbf{x}_0$



---

5. $\mathbf{G}_5 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2$
$\mathbf{x}_1 + \mathbf{x}_3 = \mathbf{x}_4$
$\mathbf{x}_2 + \mathbf{x}_5 = \mathbf{x}_6$
$\mathbf{x}_4 = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_1\langle \text{string} \rangle ; \mathbf{x}_7$
$\mathbf{x}_7 = \mathbf{x}_8 + \mathbf{x}_9$
$\mathbf{x}_8 = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_1\langle \text{unit} \rangle ; \mathbf{x}_{10}$
$\mathbf{x}_6 + \mathbf{x}_9 = \mathbf{x}_{11}$
$\mathbf{x}_{11} = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_2\langle \text{string} \rangle ; \mathbf{x}_{12}$
$\mathbf{x}_{12} = \mathbf{x}_{13} + \mathbf{x}_{14}$
$\mathbf{x}_{13} = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_2\langle \text{unit} \rangle ; \mathbf{x}_5$
$\mathbf{x}_{10} + \mathbf{x}_{14} = \mathbf{x}_3 \text{ in } \mathbf{x}_0$

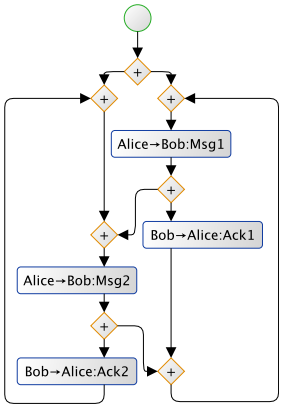


Figure 4.2.: Examples of Choreographical Global Types

2. (**Unique start**) $\mathbf{x}_0$ appears exactly once, on the left-hand side.

3. (**Unique end**) end appears at most once.

The conditions (1–3) are self-explanatory.

Thread correctness aims at verifying connexity, the ability to reach end (liveness) and that global types should always join states that occur concurrently and only them: this prevents both deadlocks and state explosion.

Inspired by [FHY14], we find that both choreographical session types and Petri nets can express flexible fork/join and choice/merge operations. And there exist well-established theories of reachability, liveness and boundedness for Petri nets [RR98]. For simplicity, our approach is to translate global types to Petri nets first and then to verify thread correctness using theorems of Petri nets. Before defining well-threadedness, we present a translation from choreographical global types to Petri nets. Recall the basic definitions of Petri nets in Section 2.3.2: a Petri net is a quadruple (P,R,F,m_0) with P a set of places, R a set of transitions, $F \subseteq P \times R \cup R \times P$ the flow relation, and m_0 the initial marking.

Definition 4.2.1 (Petri net translation)**.** *The Petri net* $\mathbb{P}(\mathbf{G}) = (P,R,F,m_0)$ *of global type* $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ *is defined as:*

$$
\begin{aligned}
P = \quad & \{\mathbf{x}_k | \mathbf{x}_k \in \widetilde{G}\} \\
R = \quad & \{r_{\mathtt{i}\to\mathtt{j}} | \mathbf{x}_i = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}_j \in \widetilde{G} \vee \mathbf{x}_i = \mathbf{x}_j + \mathbf{x}_k \in \widetilde{G} \vee \mathbf{x}_i = \mathbf{x}_k + \mathbf{x}_j \in \widetilde{G} \vee \\
& \qquad \mathbf{x}_i + \mathbf{x}_k = \mathbf{x}_j \in \widetilde{G} \vee \mathbf{x}_k + \mathbf{x}_i = \mathbf{x}_j \in \widetilde{G}\} \cup \\
& \{r_{\mathtt{i}\to\mathtt{j},\mathtt{k}} | \mathbf{x}_i = \mathbf{x}_j \mid \mathbf{x}_k \in \widetilde{G}\} \cup \{r_{\mathtt{i},\mathtt{j}\to\mathtt{k}} | \mathbf{x}_i \mid \mathbf{x}_j = \mathbf{x}_k \in \widetilde{G}\} \\
F = \quad & \{(\mathbf{x}_i, r_{\mathtt{i}\to\mathtt{j}}), (r_{\mathtt{i}\to\mathtt{j}}, \mathbf{x}_j) | \mathbf{x}_i = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}_j \in \widetilde{G}\} \cup \\
& \{(\mathbf{x}_i, r_{\mathtt{i}\to\mathtt{j}}), (r_{\mathtt{i}\to\mathtt{j}}, \mathbf{x}_j) | \mathbf{x}_i = \mathbf{x}_j + \mathbf{x}_k \in \widetilde{G} \vee \mathbf{x}_i = \mathbf{x}_k + \mathbf{x}_j \in \widetilde{G}\} \cup \\
& \{(\mathbf{x}_i, r_{\mathtt{i}\to\mathtt{j}}), (r_{\mathtt{i}\to\mathtt{j}}, \mathbf{x}_j) | \mathbf{x}_i + \mathbf{x}_k = \mathbf{x}_j \in \widetilde{G} \vee \mathbf{x}_k + \mathbf{x}_i = \mathbf{x}_j \in \widetilde{G}\} \cup \\
& \{(\mathbf{x}_i, r_{\mathtt{i}\to\mathtt{j},\mathtt{k}}), (r_{\mathtt{i}\to\mathtt{j},\mathtt{k}}, \mathbf{x}_j), (r_{\mathtt{i}\to\mathtt{j},\mathtt{k}}, \mathbf{x}_k) | \mathbf{x}_i = \mathbf{x}_j \mid \mathbf{x}_k \in \widetilde{G}\} \cup \\
& \{(\mathbf{x}_i, r_{\mathtt{i},\mathtt{j}\to\mathtt{k}}), (\mathbf{x}_j, r_{\mathtt{i},\mathtt{j}\to\mathtt{k}}), (r_{\mathtt{i},\mathtt{j}\to\mathtt{k}}, \mathbf{x}_k) | \mathbf{x}_i \mid \mathbf{x}_j = \mathbf{x}_k \in \widetilde{G}\} \\
m_0 = \quad & \{\mathbf{x}_0\}
\end{aligned}
$$

We now state well-threadedness as a Petri net property. These properties are classically decidable.

Definition 4.2.2 (Well-threadedness)**.** *A global type* $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ *satisfies the well-threadedness condition if* $\mathbb{P}(\mathbf{G})$ *is:*

1. *weakly-connected, and*

$$
\begin{array}{rcl}
\mathbf{G}_{\neg\text{thr}} & = & \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2 \\
\mathbf{x}_1 & = & \mathtt{A} \to \mathtt{B} : \mathit{Msg}_1\langle \mathsf{string}\rangle ; \mathbf{x}_3 \\
\mathbf{x}_2 & = & \mathtt{A} \to \mathtt{B} : \mathit{Msg}_2\langle \mathsf{string}\rangle ; \mathbf{x}_4 \\
\mathbf{x}_3 \mid \mathbf{x}_4 & = & \mathbf{x}_5 \\
\mathbf{x}_5 & = & \mathtt{B} \to \mathtt{A} : \mathit{Msg}_3\langle \mathsf{nat}\rangle ; \mathbf{x}_6 \\
\mathbf{x}_6 & = & \mathsf{end\ in}\ \mathbf{x}_0
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{G}_{\neg\text{loc}} & = & \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2 \\
\mathbf{x}_1 & = & \mathtt{A} \to \mathtt{B} : \mathit{Msg}_1\langle \mathsf{string}\rangle ; \mathbf{x}_3 \\
\mathbf{x}_2 & = & \mathtt{B} \to \mathtt{A} : \mathit{Msg}_2\langle \mathsf{string}\rangle ; \mathbf{x}_4 \\
\mathbf{x}_3 + \mathbf{x}_4 & = & \mathbf{x}_5 \\
\mathbf{x}_5 & = & \mathsf{end\ in}\ \mathbf{x}_0
\end{array}
$$

Figure 4.3.: Incorrect Global Types

2. *safe (i.e.* 1*-bounded), and*

3. *deadlock-free, and*

4. *if a transition* $\mathbf{x} = \mathsf{end}$ *is in* $\widetilde{G}$*, then the marking where all places are empty except for* $\mathbf{x}$ *with one token, is reachable.*

$\mathbf{G}_{\neg\text{thr}}$ in Figure 4.3 shows an illegal join waits for two mutually exclusive choice messages. As a result, B at $\mathbf{x}_5$ reaches deadlock, since he is expecting both Msg_1 and Msg_2 from A.

Local Choice is important for the consistency of a choreographical global type with respect to branching. For a global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$, we need to ensure that each choice is clearly labelled and is local to a participant. To realise this effect, we define a function $\mathit{Rcv}(\widetilde{G})(\mathbf{x})$ in Figure 4.4, which returns the set of all the participants that expect at least one message starting from $\mathbf{x}$. Also, $\mathit{Rcv}(\widetilde{G})(\mathbf{x})$ returns the label l of the received message and the encountered merging points $\tilde{\mathbf{x}}$. The equality $\mathit{Rcv}(\widetilde{G})(\mathbf{x}_1) = \mathit{Rcv}(\widetilde{G})(\mathbf{x}_2)$ holds if $\forall(\mathtt{p} : l_1 : \tilde{\mathbf{x}}_1) \in \mathit{Rcv}(\widetilde{G})(\mathbf{x}_1), \forall(\mathtt{p} : l_2 : \tilde{\mathbf{x}}_2) \in \mathit{Rcv}(\widetilde{G})(\mathbf{x}_2), l_1 \neq l_2 \vee \tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2$ share a non-null suffix.
To guarantee that choices are local to one participant, we define a function that asserts that a unique sender $\mathtt{p}$ is active in each branch $\mathbf{x}_1$ and $\mathbf{x}_2$ for a choice $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 \in \widetilde{G}$. This is written $\mathit{ASend}(\widetilde{G})(\mathbf{x}) = \mathtt{p}$ and is undefined if more than one active sender exists, that is, the choice is not local to a unique participant $\mathtt{p}$ (The function $\mathit{ASend}(\widetilde{G})(\mathbf{x}) = \mathtt{p}$ is defined in Appendix A.1). $\mathbf{G}_{\neg\text{loc}}$ in Figure 4.3 shows an illegal global type that violates this condition since the active senders of branches $\mathbf{x}_1$ and $\mathbf{x}_2$ are `Alice` and `Bob` and choice is not localised to either of them.

$$
\begin{array}{ll}
Rcv(\widetilde{G})(\mathbf{x}) = Rcv(\widetilde{G},\emptyset,\emptyset)(\mathbf{x}) & \text{(remembers recursive calls and receivers)} \\
Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}') & \\
\qquad\qquad \text{if } \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}' \in \widetilde{G} \wedge \mathtt{p}' \in \tilde{\mathtt{p}} \text{ or if } \mathbf{x} \mid \mathbf{x}'' = \mathbf{x}' \in \widetilde{G} & \\
Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = \{\mathtt{p}' : l : \tilde{\mathbf{x}}\} \cup Rcv(\widetilde{G},\tilde{\mathbf{x}},\mathtt{p}'\tilde{\mathtt{p}})(\mathbf{x}') \qquad \text{if } \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}' \in \widetilde{G} \wedge \mathtt{p}' \notin \tilde{\mathtt{p}} & \\
Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}') \cup Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}'') \text{ if } \mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G} \text{ or } \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G} & \\
Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = \emptyset & \\
\qquad\qquad \text{if } \mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \in \tilde{\mathbf{x}} \text{ or if } \mathbf{x} = \mathsf{end} \in \widetilde{G} & \\
Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = Rcv(\widetilde{G},\tilde{\mathbf{x}}\mathbf{x}'',\tilde{\mathtt{p}})(\mathbf{x}'') \qquad \text{if } \mathbf{x}' + \mathbf{x} = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \notin \tilde{\mathbf{x}} &
\end{array}
$$

Figure 4.4.: Receiver Computation (up to permutation of $\mid$ and $+$)

Definition 4.2.3 (Local Choice)**.** A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ satisfies the local choice conditions if for every transition $\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G}$, we have **(1)** (**Choice awareness**) $Rcv(\widetilde{G})(\mathbf{x}') = Rcv(\widetilde{G})(\mathbf{x}'')$; and **(2)** (**Unique sender**) $\exists \mathtt{p}, ASend(\widetilde{G})(\mathbf{x}) = \mathtt{p}$.

Linearity is to avoid processes from race-conditions, To realise linearity we require that two concurrent branches with the same messages label should not appear. This condition, *linearity*, is enforced by comparing the results of $Lin(\widetilde{G})(\mathbf{x}_1)$ and $Lin(\widetilde{G})(\mathbf{x}_2)$ when a forking transition $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ occurs in $\widetilde{G}$. The *Lin* function is defined in Figure 4.5. This function works in a similar way on labels as the *Rcv* function on message receivers. As with *Rcv*, the equality $Lin(\widetilde{G})(\mathbf{x}_1) = Lin(\widetilde{G})(\mathbf{x}_2)$ holds if $\forall(\mathtt{pp}' : l_1 : \tilde{\mathbf{x}}_1) \in Lin(\widetilde{G})(\mathbf{x}_1), \forall(\mathtt{pp}' : l_2 : \tilde{\mathbf{x}}_2) \in Lin(\widetilde{G})(\mathbf{x}_2), l_1 \neq l_2 \vee \tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2$ share a non-null suffix. For example, linearity condition would prevent the two parallel branches with labels Msg_1 and Msg_2 from both being renamed Msg_0 in $\mathbf{G}$, since they can be received concurrently and thus receiver will confuse.

Definition 4.2.4 (Linearity)**.** *A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ satisfies the linearity condition if, for every transition $\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G}$, we have $Lin(\widetilde{G})(\mathbf{x}') = Lin(\widetilde{G})(\mathbf{x}'')$.*

Well-formedness We say that a global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ is *well-formed*, if it satisfies the *sanity*, *thread correctness*, *local choice* and *linearity* conditions. We can easily check that global types from Ex. 4.2.3 are well-formed. It is noted that well-formedness of classic multiparty session types [CDCYP15] also requires sanity conditions and local choice conditions. By adding fork/join constructs, we also need to ensure thread correctness and linearity conditions.

$$
\begin{aligned}
&\mathit{Lin}(\widetilde{G})(\mathbf{x}) = \mathit{Lin}(\widetilde{G},\emptyset,\emptyset)(\mathbf{x}) && \text{(remembers merges and joins)}\\
&\mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \emptyset && \text{if } \mathbf{x}+\mathbf{x}' = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \in \tilde{\mathbf{x}} \text{ or } \mathbf{x} = \mathsf{end} \in \widetilde{G}\\
&\mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \{\mathtt{p}\mathtt{p}' : l : \tilde{\mathbf{x}}_j\} \cup \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}')\\
&\qquad && \text{if } \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}' \in \widetilde{G} \wedge \mathtt{p}' \notin \tilde{\mathtt{p}}\\
&\mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}') \cup \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}'') && \text{if } \mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G}\\
&\mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}') \cup \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}'') && \text{if } \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G}\\
&\mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m\mathbf{x}'',\tilde{\mathbf{x}}_j)(\mathbf{x}'') && \text{if } \mathbf{x}'+\mathbf{x} = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \notin \tilde{\mathbf{x}}\\
&\mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \mathit{Lin}(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j\mathbf{x}'')(\mathbf{x}'') && \text{if } \mathbf{x}' \mid \mathbf{x} = \mathbf{x}'' \in \widetilde{G}
\end{aligned}
$$

Figure 4.5.: Linearity Computation (up to permutation of $\mid$ and $+$)

4.3. Local Types and Projections

Local types, defined in Definition 4.3.1, describe the actions of session from end-point view of participants and each process implementation should follow the projected local types. Similar to global types, a local type $\mathcal{T}$ follows the shape of a state machine definition, and are of the form $\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_0$.

Definition 4.3.1 (Syntax of Choreographical Local Types)**.**

$$
\begin{array}{rcll}
\mathcal{T} & ::= & \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \text{local type}\\
T & ::= & \mathbf{x} = !\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' & \text{send}\\
& \mid & \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' & \text{receive}\\
& \mid & \mathbf{x} = \mathbf{x}' & \text{indirection}\\
& \mid & \mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' & \text{internal choice}\\
& \mid & \mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}'' & \text{external choice}\\
& \mid & \mathbf{x} + \mathbf{x}' = \mathbf{x}'' & \text{merge}\\
& \mid & \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' & \text{fork}\\
& \mid & \mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' & \text{join}\\
& \mid & \mathbf{x} = \mathsf{end} & \text{end}
\end{array}
$$

The local type of send $!\langle \mathtt{p}, l\langle U\rangle\rangle$ expresses an action of sending $\mathtt{p}$ a message with label l and type U, and receive $?\langle \mathtt{p}, l\langle U\rangle\rangle$ is the symmetric action of receiving from $\mathtt{p}$ a message with label l and type U. Other behaviours, like fork, join and end, are same as global types. Note that merge is used for collecting both internal and external choices.

We define the projection in Definition 4.3.2 to project a well-formed global type $\mathbf{G}$ to the local type of participant $\mathtt{p}$, denoted as $\mathbf{G} \upharpoonright \mathtt{p}$. The projection is straightforward: the interaction $\mathbf{x} = \mathtt{p} \to \mathtt{q} : l\langle U \rangle; \mathbf{x}'$ is an output action from $\mathtt{p}$'s viewpoint and an input action from $\mathtt{q}$'s viewpoint; otherwise it marks as an indirection link from $\mathbf{x}$ to $\mathbf{x}'$ for other participants. Choice $\mathbf{x} = \mathbf{x}' + \mathbf{x}''$ is projected as the internal choice if $\mathtt{p}$ is the participant who decides which branch to choose, and this is guaranteed by well-formedness of global types; otherwise, the projection is noted as an external choice. For local types, we also define a congruence relation $\equiv$ over $\widetilde{T}$ in Figure 4.7 which removes the indirections, locally irrelevant choices, and the unused local threads. Appendix A.3 includes examples illustrating how local congruence relation is applied to simplify graphs.

Definition 4.3.2 (Projection Algorithm)**.**

$$
\begin{array}{rcl}
\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x} \upharpoonright \mathtt{p} & = & \mathsf{def}\ \widetilde{G} \upharpoonright_{\widetilde{G}} \mathtt{p}\ \mathsf{in}\ \mathbf{x} \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = !\langle \mathtt{p}', l\langle U \rangle \rangle . \mathbf{x}' \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p}' & = & \mathbf{x} = ?\langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}' \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p}'' & = & \mathbf{x} = \mathbf{x}' \ (\mathtt{p} \notin \{\mathtt{p}, \mathtt{p}'\}) \\
\mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \\
\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \\
\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' \\
 & & (\text{if } \mathtt{p} = \mathit{ASend}(\widetilde{G})(\mathbf{x})) \\
\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathbf{x}' \ \& \ \mathbf{x}'' \\
 & & (\text{otherwise}) \\
\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} + \mathbf{x}' = \mathbf{x}'' \\
\mathbf{x} = \mathsf{end} \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathsf{end}
\end{array}
$$

Example 4.3.1 (Register Example)**.** We illustrate our projection algorithm by showing the result of the projection of the global type $\mathbf{G}_{\mathrm{Register}}$ to the three local types of the user $\mathcal{T}_{\mathrm{RegisterU}}$, the server $\mathcal{T}_{\mathrm{RegisterS}}$ and the client $\mathcal{T}_{\mathrm{RegisterD}}$ in Figure 4.6. Local type congruence rules are used to simplify the result. When comparing with the state transition diagram in Figure 4.1, we can observe the similarities in syntactic structure.

We define Local State Equivalence Relation as $X[\ ]$ in Figure 4.8. The local states are considered as equivalent if no message exchange occurs. This equivalence relation is used in the proof of subject congruence in Section 4.5. We

$$
\begin{array}{rl}
\mathcal{T}_{\mathtt{RegisterU}} = & \mathsf{def}\ \mathbf{x}_0 = !\langle \mathtt{S}, \mathit{Request}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_1 \\
& \quad \mathbf{x}_1 = ?\langle \mathtt{S}, \mathit{Form}\langle \mathsf{unit}\rangle\rangle.\mathbf{x}_2 \\
& \mathbf{x}_2 + \mathbf{x}_9 = \mathbf{x}_3 \\
& \quad \mathbf{x}_3 = !\langle \mathtt{S}, \mathit{Submit}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_4 \\
& \quad \mathbf{x}_4 = \mathbf{x}_7 \ \& \ \mathbf{x}_8 \\
& \quad \mathbf{x}_7 = ?\langle \mathtt{S}, \mathit{Invalid}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_9 \\
& \quad \mathbf{x}_8 = ?\langle \mathtt{S}, \mathit{Success}\langle \mathsf{bool}\rangle\rangle.\mathbf{x}_{14} \\
& \quad \mathbf{x}_{14} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0 \\
\mathcal{T}_{\mathtt{RegisterS}} = & \mathsf{def}\ \mathbf{x}_0 = ?\langle \mathtt{U}, \mathit{Request}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_1 \\
& \quad \mathbf{x}_1 = !\langle \mathtt{U}, \mathit{Form}\langle \mathsf{unit}\rangle\rangle.\mathbf{x}_2 \\
& \mathbf{x}_2 + \mathbf{x}_9 = \mathbf{x}_3 \\
& \quad \mathbf{x}_3 = ?\langle \mathtt{U}, \mathit{Submit}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_4 \\
& \quad \mathbf{x}_4 = !\langle \mathtt{D}, \mathit{Check}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_5 \\
& \quad \mathbf{x}_5 = ?\langle \mathtt{D}, \mathit{Reply}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_6 \\
& \quad \mathbf{x}_6 = \mathbf{x}_7 \oplus \mathbf{x}_8 \\
& \quad \mathbf{x}_7 = !\langle \mathtt{U}, \mathit{Invalid}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_9 \\
& \quad \mathbf{x}_8 = \mathbf{x}_{10} \mid \mathbf{x}_{11} \\
& \quad \mathbf{x}_{10} = !\langle \mathtt{U}, \mathit{Success}\langle \mathsf{bool}\rangle\rangle.\mathbf{x}_{12} \\
& \quad \mathbf{x}_{11} = !\langle \mathtt{D}, \mathit{Record}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_{13} \\
& \mathbf{x}_{12} \mid \mathbf{x}_{13} = \mathbf{x}_{14} \\
& \quad \mathbf{x}_{14} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0 \\
\mathcal{T}_{\mathtt{RegisterD}} = & \mathsf{def}\ \mathbf{x}_0 + \mathbf{x}_9 = \mathbf{x}_3 \\
& \quad \mathbf{x}_3 = ?\langle \mathtt{S}, \mathit{Check}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_5 \\
& \quad \mathbf{x}_5 = !\langle \mathtt{S}, \mathit{Reply}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_6 \\
& \quad \mathbf{x}_6 = \mathbf{x}_9 \ \& \ \mathbf{x}_{11} \\
& \quad \mathbf{x}_{11} = ?\langle \mathtt{S}, \mathit{Record}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_{14} \\
& \quad \mathbf{x}_{14} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

Figure 4.6.: Register Example: Local Types

$$
\begin{array}{rcl}
\widetilde{T}, \mathbf{x} = \mathbf{x}' & \equiv & \widetilde{T}[\mathbf{x}/\mathbf{x}'] \\
\mathbf{x}_1 = \mathbf{x}_2 \oplus \mathbf{x}_3, \mathbf{x}_2 + \mathbf{x}_3 = \mathbf{x}_4 & \equiv & \mathbf{x}_1 = \mathbf{x}_4 \\
\mathbf{x}_1 = \mathbf{x}_2 \ \& \ \mathbf{x}_3, \mathbf{x}_2 + \mathbf{x}_3 = \mathbf{x}_4 & \equiv & \mathbf{x}_1 = \mathbf{x}_4 \\
\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}_3, \mathbf{x}_3 = \mathbf{x}_2 \ \& \ \mathbf{x}_4 & \equiv & \mathbf{x}_1 = \mathbf{x}_4 \\
\mathbf{x}_1 = \mathbf{x}_2 \mid \mathbf{x}_3, \mathbf{x}_2 \mid \mathbf{x}_3 = \mathbf{x}_4 & \equiv & \mathbf{x}_1 = \mathbf{x}_4
\end{array}
$$

Figure 4.7.: Local Type Congruence (up to commutativity of | and +)

$$X \mid X' \equiv_{\widetilde{T}} X' \mid X \qquad X \mid (X' \mid X'') \equiv_{\widetilde{T}} (X \mid X') \mid X''$$

$$\frac{\mathbf{x} = \mathbf{x}' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}']} \qquad \frac{\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}' \mid \mathbf{x}'']} \qquad \frac{\mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x} \mid \mathbf{x}'] \equiv_{\widetilde{T}} X[\mathbf{x}'']}$$

$$\frac{\mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}']} \qquad \frac{\mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}'']} \qquad \frac{\mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}']}$$

$$\frac{\mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}'']} \qquad \frac{\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}] \equiv_{\widetilde{T}} X[\mathbf{x}'']} \qquad \frac{\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{T}}{X[\mathbf{x}'] \equiv_{\widetilde{T}} X[\mathbf{x}'']}$$

Figure 4.8.: Local State Equivalence

$$\frac{\mathcal{T} \equiv \mathcal{T}_0 \quad \mathcal{T}_0 \;\to_{\widetilde{T}}\; \mathcal{T}_0' \quad \mathcal{T}_0' \equiv \mathcal{T}'}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}'} \qquad \frac{\mathbf{x} = \mathsf{end} \in \widetilde{T}}{\widetilde{T} \vdash \mathbf{x} \triangleright \mathbf{0}}$$

$$\frac{\mathbf{x} = \mathbf{x}' \in \widetilde{T}}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}'} \qquad \frac{\mathbf{x} = !\,\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' \in \widetilde{T}}{\mathcal{T} \xrightarrow{\mathtt{p}!l\langle U\rangle}_{\widetilde{T}} \mathcal{T}'} \qquad \frac{\mathbf{x} = ?\,\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' \in \widetilde{T}}{\mathcal{T} \xrightarrow{\mathtt{p}?l\langle U\rangle}_{\widetilde{T}} \mathcal{T}'}$$

$$\frac{\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2 \in \widetilde{T}}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}_{\mathbf{x}_1 \mid \mathbf{x}_2}} \qquad \frac{\mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x} \in \widetilde{T}}{\mathcal{T}_{\mathbf{x}_1 \mid \mathbf{x}_2} \;\to_{\widetilde{T}}\; \mathcal{T}}$$

$$\frac{\mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2 \in \widetilde{T}}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}_{\mathbf{x}_1 + \mathbf{x}_2}} \qquad \frac{\mathbf{x} = \mathbf{x}_1 \,\&\, \mathbf{x}_2 \in \widetilde{T}}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}_{\mathbf{x}_1 + \mathbf{x}_2}} \qquad \frac{\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x} \in \widetilde{T}}{\mathcal{T}_1 \;\to_{\widetilde{T}}\; \mathcal{T} \quad \mathcal{T}_2 \;\to_{\widetilde{T}}\; \mathcal{T}}$$

$$\frac{\mathbf{x}_1 + \mathbf{x} = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}' \oplus \mathbf{x}_2' \in \widetilde{T} \quad \mathcal{T}_2 \;\to^*_{\widetilde{T}}\; \mathcal{T}_1' \quad \mathcal{T}_2' \;\to^*_{\widetilde{T}}\; \mathcal{T}_1}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}'}$$

$$\frac{\mathbf{x}_1 + \mathbf{x} = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}_2' \oplus \mathbf{x}' \in \widetilde{T} \quad \mathcal{T}_2 \;\to^*_{\widetilde{T}}\; \mathcal{T}_1' \quad \mathcal{T}_2' \;\to^*_{\widetilde{T}}\; \mathcal{T}_1}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}'}$$

$$\frac{\mathbf{x} + \mathbf{x}_1 = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}' \,\&\, \mathbf{x}_2' \in \widetilde{T} \quad \mathcal{T}_2 \;\to^*_{\widetilde{T}}\; \mathcal{T}_1' \quad \mathcal{T}_2' \;\to^*_{\widetilde{T}}\; \mathcal{T}_1}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}'}$$

$$\frac{\mathbf{x} + \mathbf{x}_1 = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}_2' \,\&\, \mathbf{x}' \in \widetilde{T} \quad \mathcal{T}_2 \;\to^*_{\widetilde{T}}\; \mathcal{T}_1' \quad \mathcal{T}_2' \;\to^*_{\widetilde{T}}\; \mathcal{T}_1}{\mathcal{T} \;\to_{\widetilde{T}}\; \mathcal{T}'}$$

Figure 4.9.: Transition Rules for Local Choreographical Types

define the transition rules for local types in Figure 4.9 as auxiliary theorems for proving type safety.

4.4. General Multiparty Session Processes

This section introduces *general multiparty session processes*. We give the syntax of choreographical session calculus and show the operational semantics to allow processes to reduce. Also, we define the typing system for processes for verification. The extension to run-time processes is defined to show the transitions of session environment.

4.4.1. Syntax of Choreographical Session Calculus

Similar to local types, process calculus also follows the shape of the choreographical global types defined in Section 4.2.1: new external and internal choice operators are added to allow branching with different receivers and merging with different senders; and forking and joining threads are included in the syntax.

Definition 4.4.1 (Process and Network Syntax)**.** The syntax of processes and network for choreographical session calculus is defined below.

$P ::= \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}$	definition
$P ::=$	process transition
$\mid \mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e})$	init
$\mid \mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e})$	request
$\mid \mathbf{x}(\tilde{x}) = x!\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e})$	send
$\mid \mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e})$	receive
$\mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z})$	parallel
$\mid \mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'')$	conditional
$\mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x})$	external choice
$\mid \mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x})$	join
$\mid \mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x})$	merge
$\mid \mathbf{x}(\tilde{x}) = (\nu a)\ \mathbf{x}'(a\tilde{x})$	new name
$\mid \mathbf{x}(\tilde{x}) = \mathbf{0}$	null

$\mathbf{X} ::=$	state
$\quad \mid \mathbf{x}(\tilde{v})$	thread
$\quad \mid \mathbf{X} \mid \mathbf{X}$	parallel
$\quad \mid (\nu a)\mathbf{X}$	restriction
$\quad \mid \mathbf{0}$	null
$\mathbf{N} ::=$	network
$\quad \mid P$	def
$\quad \mid \mathbf{N} \mid\mid \mathbf{N}$	parallel
$\quad \mid (\nu a)\mathbf{N}$	new name
$\quad \mid \mathbf{0}$	null
$\quad \mid (\nu s)\mathbf{N}$	new session
$\quad \mid s : h$	queue
$\quad \mid a\langle s \rangle[\mathtt{p}]$	invitation
$v ::= a \mid s[\mathtt{p}] \mid \mathtt{true} \mid \mathtt{false} \mid \ldots$	values
$e ::= v \mid x \mid e \wedge e \mid \ldots$	expression
$h ::= \emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)$	messages

A process starts from a definition $P = \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}(\tilde{v})$, and $\tilde{v}$ instantiates the parameters of $\mathbf{x}$ in $\widetilde{P}$. The form of process actions $\widetilde{P}$ follows the shape of global and local types and processes is based on a functional style to pass values around continuations. Variables $\tilde{x}$ in $\mathbf{x}(\tilde{x})$ on the left-hand side of a process action are binding variables on the the $\tilde{x}$ appearing in the right-hand side. y in request and receive are also binding variable.

Session initiation is of the form $\mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e})$ where $\mathbf{G}$ is a global type. It follows an interaction pattern defined in $\mathbf{G}$ to the shared channel a that x is substituted to. The variables in $\tilde{e}$ are all bound by $\tilde{x}$. After a session initialisation, participants can accept the session with $\mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e})$, and start the interaction, where the variables in $\tilde{e}$ are bound by $\tilde{x}$ and by y, which receives the session channel.

The sending action $x!\langle \mathtt{p}, l\langle e \rangle \rangle$ allows sending to $\mathtt{p}$ a value e labelled by a constant l in session x. The reception $x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e})$ is the symmetric action to receive a message with a label l from $\mathtt{p}$. It is noted that the message is then received in variable y that binds in $\mathbf{x}'(\tilde{e})$.

$\mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z})$ express forking threads and $\tilde{y}$ and $\tilde{z}$ are subsets of $\tilde{x}$.

The join action combines parallel threads. The conditional process `if` e `then` $\mathbf{x}'(\tilde{e}')$ `else` $\mathbf{x}''(\tilde{e}'')$ for internal choice and the $(\mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}))$ for external choices represent traditional selection actions in session types. The merge action collects choices. Note that fork, join, external choice and merge only allow a restricted use of bound variables in continuations. $\mathbf{x}(\tilde{x}) = (\nu a)\mathbf{x}'(a\tilde{x})$ is name creation of a new shared name a. $\mathbf{0}$ is an inactive agent.

The process states $\mathbf{X}$ are defined from the state variables defined in $\widetilde{P}$. The network $\mathbf{N}$, with restrictions of the form $(\nu a)\mathbf{N}$, means a parallel composition of definition agents.

Session instances are represented by session restriction $(\nu s)P$. The message queue $s : h$ stores the messages in transit for the session instance s. A session invitation, denoted as $a[\mathtt{p}]\langle s\rangle$, invites participant $\mathtt{p}$ to start the session s on channel a.

We say that a network is *initial* if it only consists of shared name restrictions and parallel compositions of $\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}(\vec{v})$.

Operational Semantics We define the operational semantics for processes and networks in Figure 4.10. The labels used in the reduction of processes are defined as below.

$$\alpha,\beta ::= \ \tau \ \mid\ s[\mathtt{p},\mathtt{q}]!l\langle v\rangle \ \mid\ s[\mathtt{p},\mathtt{q}]?l\langle v\rangle \ \mid\ a\langle\mathbf{G}\rangle \ \mid\ a\langle\mathtt{p}\rangle[s]$$

The rules consist of two parts. The first part shows the transitions of the form $\widetilde{P}\vdash \mathbf{X}\xrightarrow{\alpha}\mathbf{X}'$ representing that a process in a state $\mathbf{X}$ can transit to state $\mathbf{X}'$ with action α. The second part defines reductions within networks with transitions $\mathbf{N}\ \rightarrow\ \mathbf{N}'$. $e\downarrow v$ corresponds to the evaluation of expression e to v.

In the first part, rule [Init] initiates a session, and rule [Acc] signifies the process's acceptance for participate in a session. Rule [Send] outputs a message from $\mathtt{p}$ to $\mathtt{q}$, with $\tilde{x}$ substituting by $\tilde{v}$ and evaluating e to v. Rule [Rcv] receives a message and instantiates variable y to the received value v. Rule [New] represents name creation. A branch with the value of e is internally chosen in Rule [IfT]. Rule [Ext] shows the external choice, and depending on which label α is received, either the left or right state variable is invoked.

For the second part, rules [Def] and [Tau] raise to the network level. [Init$_N$] is used in combination with [Init] to creates an empty queue $s : \varepsilon$ together with invitations for participant. Rule [Acc$_N$] consumes an invitation in the session if someone signals as ready by [Acc]. Other contextual rules are standard. We write

$$\frac{x[\tilde{v}/\tilde{x}] = a \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{a\langle \mathbf{G} \rangle} \mathbf{x}'(\tilde{v}')} \quad [\textsc{Init}]$$

$$\frac{x[\tilde{v}/\tilde{x}] = a \qquad \tilde{e}[\tilde{v}/\tilde{x}][s/y] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{a\langle s \rangle[\mathtt{p}]} \mathbf{x}'(\tilde{v}')} \quad [\textsc{Acc}]$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad e[\tilde{v}/\tilde{x}] \downarrow v \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x\,!\,\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v \rangle} \mathbf{x}'(\tilde{v}')} \quad [\textsc{Send}]$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad \tilde{e}[\tilde{v}/\tilde{x}][v/y] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v' \rangle} \mathbf{x}'(\tilde{v}')} \quad [\textsc{Rcv}]$$

$$\frac{a \notin \tilde{v}}{\mathbf{x}(\tilde{x}) = (\nu a)\mathbf{x}'(a\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} (\nu a)\mathbf{x}'(a\tilde{v})} \quad [\textsc{New}]$$

$$\frac{e[\tilde{v}/\tilde{x}] \downarrow \mathsf{true} \qquad \tilde{e}'[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'') \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} \mathbf{x}'(\tilde{v}')} \quad [\textsc{IfT}]$$

$$\frac{e[\tilde{v}/\tilde{x}] \downarrow \mathsf{false} \qquad \tilde{e}''[\tilde{v}/\tilde{x}] \downarrow \tilde{v}''}{\mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{x})\ \mathtt{else}\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} \mathbf{x}''(\tilde{v}'')} \quad [\textsc{IfF}]$$

$$\frac{\widetilde{P}, \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}'(\tilde{v}) \xrightarrow{\alpha} \mathbf{X}}{\widetilde{P}, \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\alpha} \mathbf{X}} \quad [\textsc{Ext}]$$

$$\frac{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}{\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \xrightarrow{\alpha} \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}'} \quad [\textsc{Def}]$$

$$\frac{P \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v \rangle} P'}{P \mid\mid s : h \to P' \mid\mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)} \quad [\textsc{Put}]$$

$$\frac{P \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v \rangle} P'}{P \mid\mid s : (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \cdot h \to P' \mid\mid s : h} \quad [\textsc{Get}]$$

$$\frac{P \xrightarrow{a\langle \mathbf{G} \rangle} P' \quad \mathtt{p}_0, \ldots, \mathtt{p}_k \in \mathbf{G} \quad s \notin \mathsf{fn}(P')}{P \to (\nu s)(P' \mid\mid s : \varepsilon \mid\mid a\langle s \rangle[\mathtt{p}_0] \mid\mid \ldots \mid\mid a\langle s \rangle[\mathtt{p}_k])} \quad [\textsc{Init}_N]$$

$$\frac{P \xrightarrow{a\langle s \rangle[\mathtt{p}]} P'}{P \mid\mid a\langle s \rangle[\mathtt{p}] \to P'} [\textsc{Acc}_N] \qquad \frac{P \xrightarrow{\tau} P'}{P \to P'} [\textsc{Tau}] \qquad \frac{\widetilde{P} \vdash \mathbf{X}_0 \xrightarrow{\alpha} \mathbf{X}_0'}{\widetilde{P} \vdash \mathbf{X}_0 \mid \mathbf{X} \xrightarrow{\alpha} \mathbf{X}_0' \mid \mathbf{X}} \quad [\textsc{Par}]$$

$$\frac{\widetilde{P}' \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}{\widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'} [\textsc{Weak}] \qquad \frac{\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}_0 \xrightarrow{\alpha} \mathbf{X}_0' \equiv \mathbf{X}'}{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'} [\textsc{Str}] \qquad \frac{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}' \quad a \notin \alpha}{\widetilde{P} \vdash (\nu a)\mathbf{X} \xrightarrow{\alpha} (\nu a)\mathbf{X}'} \quad [\textsc{Res}]$$

$$\frac{\mathbf{N} \to \mathbf{N}'}{(\nu n)\ \mathbf{N} \to (\nu n)\ \mathbf{N}'} [\textsc{Res}_N] \qquad \frac{\mathbf{N} \to \mathbf{N}'}{\mathbf{N} \mid\mid \mathbf{N}'' \to \mathbf{N}' \mid\mid \mathbf{N}''} [\textsc{Par}_N] \qquad \frac{\mathbf{N} \equiv \mathbf{N}_0 \to \mathbf{N}_0' \equiv \mathbf{N}'}{\mathbf{N} \to \mathbf{N}'} \quad [\textsc{Str}_N]$$

Figure 4.10.: Operational Semantics of Session Calculus

$\longrightarrow^*$ for the multi-step reduction.

Example 4.4.2 (Register Example)**.** We write here an implementation of the Register example from Section 4.2.1. The global and local types are referred to Figure 4.1 and Figure 4.6.

$P_{\mathtt{U}}$, $P_{\mathtt{S}}$ and $P_{\mathtt{D}}$, respectively correspond to the user, server and remote database. $P_{\mathtt{U}}$ initiates the session by announcing $\mathbf{G}_{\text{Trade}}$ on shared name a. According to rule [INIT$_N$], it creates a session name s, a message buffer and invitations for U, S and D. $P_{\mathtt{U}}$ then joins the session as the user U, the variable z being used to contain the session name. $P_{\mathtt{U}}$ proceeds with $\mathbf{x}_0(y_0, z)$ where y_0 is the integer 1 to represent the number for request, and z the session name. The execution of $\mathbf{x}_0(y_0, z)$ sends a message *Request* with payload integer 1 in the message buffer and user continues interactions as described in the process. $P_{\mathtt{S}}$ starts in $\mathbf{x}(a, form)$ where a is the shared name and $form$ is a generated registration form. It joins the session as S, gets in variable z the session name s and continues with $\mathbf{x}_0(i_0, z)$. The registration information i_2 from user is checked recursively with D until valid information is obtained and S notifies user with the success message number i_3 and notifies database to record this user information with command number i_4. Similarly, $P_{\mathtt{D}}$ begins with $\mathbf{x}(a)$ by joining the session as D, helps server to check the uniqueness of user with message d_1, and sends the result back with correct/error indicator d_2. The information will be eventually recorded with command number d_3 from the server.

4.4.2. Typing Multiparty Interactions

in this subsection, we define the typing system for choreographical session calculus. What is different from existing multiparty typing systems is that we need to gather the typing constraints of the transitions $\widetilde{P}$ in P, and keep track of associations such as $\mathbf{x}_1 = !\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}_2$. The typing system relies on matching between local types and inferred transitions in order to keep the typing system for processes simple.

Environments We write u as a shared channel a and its variable x and c as a session channel $s[\mathtt{p}]$ or its variable. The environments are defined as:

$$\Gamma ::= \emptyset \mid \Gamma, u : U \qquad \Delta ::= \emptyset \mid \Delta, c : \mathcal{T} \qquad \Sigma ::= \emptyset \mid \Sigma, \mathbf{x} : \tilde{U}$$

Γ is the *standard environment* that associates variables to sort types and shared names to global types. Δ is the *session environment* that associates channels to

$$
\begin{aligned}
P_{\mathtt{U}} = \mathsf{def}\quad \mathbf{x}(x,y_0) &= x\langle \mathbf{G}_{\text{Register}}\rangle.\mathbf{x}'(x,y_0)\\
\mathbf{x}'(x,y) &= x[\mathtt{U}](z).\mathbf{x}_0(y_0,z)\\
\mathbf{x}_0(y_0,z) &= z\,!\,\langle \mathtt{S}, \mathit{Request}\langle y_0\rangle\rangle.\mathbf{x}_1(z)\\
\mathbf{x}_1(z) &= z\,?\,\langle \mathtt{S}, \mathit{Form}(y_1)\rangle.\mathbf{x}_2(y_1,z)\\
\mathbf{x}_2(y_1,z) + \mathbf{x}_9(y_3,z) &= \mathbf{x}_3(y_2,z)\\
\mathbf{x}_3(y_2,z) &= z\,!\,\langle \mathtt{S}, \mathit{Submit}\langle y_1\rangle\rangle.\mathbf{x}_4(z)\\
\mathbf{x}_4(z) &= \mathbf{x}_7(z)\ \&\ \mathbf{x}_8(z)\\
\mathbf{x}_7(z) &= z\,?\,\langle \mathtt{S}, \mathit{Invalid}(y_3)\rangle.\mathbf{x}_9(y_3,z)\\
\mathbf{x}_8(z) &= z\,?\,\langle \mathtt{S}, \mathit{Success}(y_4)\rangle.\mathbf{x}_{14}(y_4,z)\\
\mathbf{x}_{14}(y_4,z) &= \mathbf{0} \qquad \mathsf{in}\ \mathbf{x}(a,1)
\end{aligned}
$$

$$
\begin{aligned}
P_{\mathtt{S}} = \mathsf{def}\quad \mathbf{x}(x,i_0) &= x[\mathtt{S}](z).\mathbf{x}_0(i_0,z)\\
\mathbf{x}_0(i_0,z) &= z\,?\,\langle \mathtt{U}, \mathit{Request}(i_1)\rangle.\mathbf{x}_1(i_0,i_1,z)\\
\mathbf{x}_1(i_0,i_1,z) &= z\,!\,\langle \mathtt{U}, \mathit{Form}\langle i_0\rangle\rangle.\mathbf{x}_2(z)\\
\mathbf{x}_2(z) + \mathbf{x}_9(z) &= \mathbf{x}_3(z)\\
\mathbf{x}_3(z) &= z\,?\,\langle \mathtt{U}, \mathit{Submit}(i_2)\rangle.\mathbf{x}_4(i_2,z)\\
\mathbf{x}_4(i_2,z) &= z\,!\,\langle \mathtt{D}, \mathit{Check}\langle i_2\rangle\rangle.\mathbf{x}_5(z)\\
\mathbf{x}_5(z) &= z\,?\,\langle \mathtt{D}, \mathit{Reply}(i_3)\rangle.\mathbf{x}_6(i_3,z)\\
\mathbf{x}_6(i_3,z) &= \mathit{if}\ i_3 = 0\ \mathit{then}\ \mathbf{x}_7(i_3,z)\ \mathit{else}\ \mathbf{x}_8(i_3,z)\\
\mathbf{x}_7(i_3,z) &= z\,!\,\langle \mathtt{U}, \mathit{Invalid}\langle i_3\rangle\rangle.\mathbf{x}_9(z)\\
\mathbf{x}_8(i_3,z) &= \mathbf{x}_{10}(i_3,z) \mid \mathbf{x}_{11}(i_4,z)\\
\mathbf{x}_{10}(i_3,z) &= z\,!\,\langle \mathtt{U}, \mathit{Success}\langle i_3\rangle\rangle.\mathbf{x}_{12}(z)\\
\mathbf{x}_{11}(i_4,z) &= z\,!\,\langle \mathtt{D}, \mathit{Record}\langle i_4\rangle\rangle.\mathbf{x}_{13}(z)\\
\mathbf{x}_{12}(z) \mid \mathbf{x}_{13}(z) &= \mathbf{x}14(z)\\
\mathbf{x}_{14}(z) &= \mathbf{0} \qquad \mathsf{in}\ \mathbf{x}(a,\mathit{form})
\end{aligned}
$$

$$
\begin{aligned}
P_{\mathtt{D}} = \mathsf{def}\quad \mathbf{x}(x) &= x[\mathtt{D}](z).\mathbf{x}_0(z)\\
\mathbf{x}_0(z) + \mathbf{x}_9(z) &= \mathbf{x}_3(z)\\
\mathbf{x}_3(z) &= z\,?\,\langle \mathtt{S}, \mathit{Check}(d_1)\rangle.\mathbf{x}_5(d_1,z)\\
\mathbf{x}_5(d_1,z) &= z\,!\,\langle \mathtt{S}, \mathit{Reply}\langle d_2\rangle\rangle.\mathbf{x}_6(z)\\
\mathbf{x}_6(z) &= \mathbf{x}_9(z)\ \&\ \mathbf{x}_{11}(z)\\
\mathbf{x}_{11}(z) &= z\,?\,\langle \mathtt{S}, \mathit{Record}(d_3)\rangle.\mathbf{x}_{14}(d_3,z)\\
\mathbf{x}_{14}(d_3,z) &= \mathbf{0} \qquad \mathsf{in}\ \mathbf{x}(a)
\end{aligned}
$$

Figure 4.11.: Register Example: Processes

session types. What is different from multiparty session types is that we use Σ to keep tracking state variable associations in the typing system, since we introduce state variables in local types and processes. We say $\Gamma, u : U$ only if $u \notin dom(\Gamma)$, and it is similar for other variables.

Judgements The different judgements are used:

$\Gamma \vdash e : U$	Expression e has type U under Γ
$\Gamma \vdash P \triangleright \Sigma \parallel \Sigma'$	Left/right variables in P have types Σ/Σ' under Γ
$\Gamma \vdash P$	Process P is typed under Γ
$\Gamma \vdash \mathbf{N}$	Network $\mathbf{N}$ is typed under Γ

Typing Rules We define the typing system for processes in Figure 4.12. Apart from inferring local types through gathered constraint, the essence of the typing system is the same as that of [HYC08, CDCYP15]. In the rules, $\tilde{y}$ and $\tilde{z}$ correspond to sorts and session types, respectively.

Rule [INIT] types the initialisation, where $\tilde{y}$ should contain x and variables in $\tilde{e}$ appearing in the right hand side. In the type system we record that $\mathbf{x} = \mathbf{x}'$ at the head of $\mathcal{T}$ and every z_i should have type $\mathcal{T} \uplus \mathbf{x} = \mathbf{x}'$. Rule [REQ] is similar but we record the introduced projected local type $\mathcal{T} = \mathbf{G} \upharpoonright \mathtt{p}$. Rule [SEND] types the send action with recording z_i ($T_i = \mathbf{x} =! \langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}'$) and $\mathbf{x} = \mathbf{x}'$ for all other sessions. Rule [RECV] for input is its symmetric rule. It records $\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =? \langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}'$ for z_i and $\mathbf{x} = \mathbf{x}'$ for all other sessions. Rule [DELEG] types the delegation by sending a session name where we record the type $\mathcal{T}$ of the sent session name y as the argument of $\mathbf{x}$. Rule [CATCH] is its symmetric rule with recording $\mathcal{T}$ for y as the final argument $\mathbf{x}'$.

Rule [PAR] types the parallel composition. The operation $(\mathcal{T}_1 \cup \mathcal{T}_2) \uplus \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ is defined as $\mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2, \widetilde{T_1} \cup \widetilde{T_2}\ \mathsf{in}\ \mathbf{x}$ where $\mathcal{T}_i = \mathsf{def}\ \widetilde{T_i}\ \mathsf{in}\ \mathbf{x}_i$ and we record $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ at the head of a union of $\mathcal{T}_1$ and $\mathcal{T}_2$. Rule [JOIN] is the symmetric rule.

Rule [IF] describes the internal choice and variables in e and $\tilde{e}$ are covered by $\tilde{y}$. Rule [CHOICE] types the external choice. Rule [MERGE] collects the branching. In [DEF], we write $comp(\{\Sigma_i ; \Sigma_i'\}_i)$ if (1) there exists unique $\mathbf{x}_0 \in (\cup_i \Sigma_i) \setminus (\cup_i \Sigma_i')$; (2) for all $\mathbf{x} \neq \mathbf{x}_0$, $\mathbf{x}$ appears exactly once in Σ_i and Σ_j' for a unique i, j. (3) if $\Sigma_i(\mathbf{x}) = \tilde{U}; \tilde{\mathcal{T}}$ and $\Sigma_j'(\mathbf{x}) = \tilde{U}'; \tilde{\mathcal{T}}'$, then $\tilde{U} = \tilde{U}'$ and $\tilde{\mathcal{T}} = \tilde{\mathcal{T}}'$. The conditions (1,2) corresponds to the Unambiguity condition in the well-formed global types defined in Section 4.2.2, and (3) checks the matching of the argument types of each state variable appears in right Σ_i and left Σ_j'. The rules for expressions ([TRUE, NAMES, NOT]) are standard. In rule [RES], we record type $\langle \mathbf{G} \rangle$ for a.

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \tilde{y}:\tilde{U}\vdash x:\langle\mathbf{G}\rangle \quad \forall i,\mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=x\langle\mathbf{G}\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}':\tilde{U}'\,\widetilde{\mathcal{T}'}}\ [\textsc{Init}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \tilde{y}:\tilde{U}\vdash x:\langle\mathbf{G}\rangle \quad \forall i,\mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=\mathbf{x}' \quad \mathcal{T}=\mathbf{G}\upharpoonright\mathtt{p}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y)\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}':\tilde{U}'\,\widetilde{\mathcal{T}'}\mathcal{T}}\ [\textsc{Req}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash e:U \quad \tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=!\,\langle\mathtt{p},l\langle U\rangle\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=z_i\,!\,\langle\mathtt{p},l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}':\tilde{U}'\,\widetilde{\mathcal{T}'}}\ [\textsc{Send}]$$

$$\frac{\tilde{y}:\tilde{U},y:U\vdash\tilde{e}:\tilde{U}' \quad \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=?\,\langle\mathtt{p},l\langle U\rangle\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=z_i\,?\,\langle\mathtt{p},l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}':\tilde{U}'\,\widetilde{\mathcal{T}'}}\ [\textsc{Recv}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=!\,\langle\mathtt{p},l\langle\mathcal{T}\rangle\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z}y)=z_i\,!\,\langle\mathtt{p},l\langle y\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\mathcal{T}\ \|\ \mathbf{x}':\tilde{U}'\,\widetilde{\mathcal{T}'}}\ [\textsc{Deleg}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=?\,\langle\mathtt{p},l\langle\mathcal{T}\rangle\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=z_i\,?\,\langle\mathtt{p},l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z}y)\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}':\tilde{U}'\,\widetilde{\mathcal{T}'}\mathcal{T}}\ [\textsc{Catch}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{y}_1:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{y}_2:\tilde{U}_2 \quad \forall i,\mathcal{T}_i=(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})\uplus\mathbf{x}=\mathbf{x}_1\mid\mathbf{x}_2}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=\mathbf{x}_1(\tilde{y}_1\tilde{z})\mid\mathbf{x}_2(\tilde{y}_2\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}_1:\tilde{U}_1\,\widetilde{\mathcal{T}_1},\mathbf{x}_2:\tilde{U}_2\,\widetilde{\mathcal{T}_2}}\ [\textsc{Par}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{y}_1:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{y}_2:\tilde{U}_2 \quad \forall i,(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})=\mathcal{T}_i'\uplus\mathbf{x}_1\mid\mathbf{x}_2=\mathbf{x}}{\vdash\mathbf{x}_1(\tilde{y}_1\tilde{z})\mid\mathbf{x}_2(\tilde{y}_2\tilde{z})=\mathbf{x}(\tilde{y}\tilde{z})\triangleright\ \mathbf{x}_1:\tilde{U}_1\,\widetilde{\mathcal{T}_1},\mathbf{x}_2:\tilde{U}_2\,\widetilde{\mathcal{T}_2}\ \|\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}'}}\ [\textsc{Join}]$$

$$\frac{\tilde{y}:\tilde{U}\vdash e:U \quad \tilde{y}:\tilde{U}\vdash\tilde{e}_1:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{e}_2:\tilde{U}_2 \quad \forall i,\mathcal{T}_i=(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})\uplus\mathbf{x}=\mathbf{x}_1\oplus\mathbf{x}_2}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=\mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}_1(\tilde{e}_1\tilde{z})\ \mathtt{else}\ \mathbf{x}_2(\tilde{e}_2\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}_1:\tilde{U}_1\,\widetilde{\mathcal{T}_1},\mathbf{x}_2:\tilde{U}_2\,\widetilde{\mathcal{T}_2}}\ [\textsc{If}]$$

$$\frac{\forall i,\mathcal{T}_i=(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})\uplus\mathbf{x}=\mathbf{x}_1\ \&\ \mathbf{x}_2}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=\mathbf{x}_1(\tilde{y}\tilde{z})\ \&\ \mathbf{x}_2(\tilde{y}\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\ \mathbf{x}_1:\tilde{U}\,\widetilde{\mathcal{T}_1},\mathbf{x}_2:\tilde{U}\,\widetilde{\mathcal{T}_2}}\ [\textsc{Choice}]$$

$$\frac{\forall i,\mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}_1+\mathbf{x}_2=\mathbf{x}}{\vdash\mathbf{x}_1(\tilde{y}\tilde{z})+\mathbf{x}_2(\tilde{y}\tilde{z})=\mathbf{x}(\tilde{y}\tilde{z})\triangleright\ \mathbf{x}_1:\tilde{U}\,\widetilde{\mathcal{T}},\mathbf{x}_2:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}'}}\ [\textsc{Merge}]$$

$$\frac{\vdash P_i\triangleright\Sigma_i\ \|\ \Sigma_i' \quad comp(\{\Sigma_i\ \|\ \Sigma_i'\}_i) \quad \mathbf{x}_0:\tilde{U}=(\cup\Sigma_i)\setminus(\cup\Sigma_i') \quad \Gamma\vdash\tilde{v}:\tilde{U}}{\Gamma\vdash\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}_0(\tilde{v})}\ [\textsc{Def}]$$

$$\frac{}{\Gamma\vdash\mathsf{true}:\mathsf{bool}}[\textsc{True}] \quad \frac{u:U\in\Gamma}{\Gamma\vdash u:U}[\textsc{Name}] \quad \frac{\Gamma\vdash e:\mathsf{bool}}{\Gamma\vdash\mathsf{not}(e):\mathsf{bool}}\ [\textsc{Not}]$$

$$\frac{\forall i,\widetilde{\mathcal{T}_i'}=\widetilde{\mathcal{T}_i}\uplus\mathbf{x}=\mathbf{x}' \qquad \widetilde{\mathcal{T}_i}=\widetilde{\mathcal{T}_i'}\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=(\nu a)\mathbf{x}'(a\tilde{y}\tilde{z})\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|\,\mathbf{x}':\langle\mathbf{G}\rangle\tilde{U}\,\widetilde{\mathcal{T}'}}[\textsc{Res}] \quad \frac{\forall i,\mathcal{T}_i=\mathsf{def}\ \mathbf{x}=\mathsf{end}\ \mathsf{in}\ \mathbf{x}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=\mathbf{0}\triangleright\mathbf{x}:\tilde{U}\,\widetilde{\mathcal{T}}\ \|}\ [\textsc{Nil}]$$

$$\frac{\Gamma\vdash\mathbf{N}_1 \quad \Gamma\vdash\mathbf{N}_2}{\Gamma\vdash\mathbf{N}_1\ ||\ \mathbf{N}_2}[\textsc{NPar}] \qquad \frac{\Gamma,a:\langle\mathbf{G}\rangle\vdash\mathbf{N}}{\Gamma\vdash(\nu a)\mathbf{N}}[\textsc{NRes}] \qquad \frac{}{\Gamma\vdash\mathbf{0}}\ [\textsc{NNil}]$$

Figure 4.12.: Typing System for State Processes

$$
\begin{array}{rcl}
\mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z}) & \vdash & \mathbf{x}(\tilde{v}) \equiv \mathbf{x}'(\tilde{y}[\tilde{v}/\tilde{x}]) \mid \mathbf{x}''(\tilde{z}[\tilde{v}/\tilde{x}]) \\
\mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x}) & \vdash & \mathbf{x}(\tilde{y}[\tilde{v}/\tilde{x}]) \mid \mathbf{x}'(\tilde{z}[\tilde{v}/\tilde{x}]) \equiv \mathbf{x}''(\tilde{v}) \\
\mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) & \vdash & \mathbf{x}(\tilde{v}) \equiv \mathbf{x}''(\tilde{v}) \\
\mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) & \vdash & \mathbf{x}'(\tilde{v}) \equiv \mathbf{x}''(\tilde{v}) \\
\mathbf{x}(\tilde{x}) = \mathbf{0} & \vdash & \mathbf{x}(\tilde{v}) \equiv \mathbf{0} \\
\widetilde{P} & \vdash & \mathbf{X} \mid (\nu a)\mathbf{X}' \equiv (\nu a)(\mathbf{X} \mid \mathbf{X}') \qquad (a \notin \mathbf{X}) \\
\widetilde{P} & \vdash & \mathbf{X} \mid \mathbf{X}' \equiv \mathbf{X}' \mid \mathbf{X}
\end{array}
$$

$$
\frac{\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}''}{\widetilde{P} \vdash \mathbf{X} \mid \mathbf{X}' \equiv \mathbf{X}'' \mid \mathbf{X}'} \qquad \frac{\widetilde{P}' \vdash \mathbf{X} \equiv \mathbf{X}'}{\widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X} \equiv \mathbf{X}'}
$$

$$
\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} \qquad \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ (\nu a)\ \mathbf{X} \equiv (\nu a)\ \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}
$$

$$
\mathbf{N} \,||\, \mathbf{0} \equiv \mathbf{N} \qquad \mathbf{N} \,||\, \mathbf{N}' \equiv \mathbf{N}' \,||\, \mathbf{N} \qquad \mathbf{N} \,||\, (\mathbf{N}' \,||\, \mathbf{N}'') \equiv (\mathbf{N} \,||\, \mathbf{N}') \,||\, \mathbf{N}''
$$

$$
\mathbf{N} \,||\, (\nu n)\mathbf{N}' \equiv (\nu n)(\mathbf{N} \,||\, \mathbf{N}') \quad (n \notin \mathbf{N}) \qquad (\nu n)\ \mathbf{0} \equiv \mathbf{0} \qquad (\nu s)(s : h) \equiv \mathbf{0}
$$

Figure 4.13.: Structural Congruence for Processes and Networks

We define structural congruence for process states, processes and networks in Figure 4.13 and omit the same ones as in [CDCYP15].

4.4.3. Typing for Run-time Processes

During runtime, we also need queues to buffer the messages being sent; just pure processes are not enough to model and verify the communication system. Therefore we extend and define the typing systems for runtime processes. We use the following abbreviations for message types.

$$
\begin{array}{rcl}
\mathcal{T} & ::= & \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ X \\
\mathtt{T} & ::= & !\langle \mathrm{p}_1, l_1\langle U_1\rangle\rangle; !\langle \mathrm{p}_2, l_2\langle U_2\rangle\rangle; \cdots !\langle \mathrm{p}_n, l_n\langle U_n\rangle\rangle
\end{array}
$$

Type $\mathtt{T}$ denotes as a *message type* that contains a sequence of outputs which represents a sequence of messages stored in a queue [CDCYP15]. Note that we write single $x = \varepsilon$ instead of $x = \mathsf{end}$ for message type $\mathtt{T}$.

$$
\begin{array}{rl}
\Delta \mid \Delta' = & \Delta \backslash dom(\Delta') \cup \Delta' \backslash dom(\Delta) \\
& \cup \{c : \Delta(c) | \Delta'(c) \mid c \in dom(\Delta) \cap dom(\Delta')\}
\end{array}
$$

where

$$
(\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1) \mid (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_2) = \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ (\mathbf{x}_1 \mid \mathbf{x}_2)
$$

$$
!\langle \mathrm{p}_1, l_1\langle U_1\rangle\rangle; \cdots !\langle \mathrm{p}_n, l_n\langle U_n\rangle\rangle \mid (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}) = \mathsf{def}\ \widetilde{T}'\ \mathsf{in}\ \mathbf{x}
$$

with $\widetilde{T}' = \mathbf{x} = !\langle \mathrm{p}_1, l_1\langle U_1\rangle\rangle.\mathbf{x}_1, \cdots \mathbf{x}_n = !\langle \mathrm{p}_2, l_2\langle U_2\rangle\rangle.\mathbf{x}_{n+1}, \widetilde{T}[\mathbf{x}_{n+1}/\mathbf{x}]$.

$$\frac{\vdash P_i \triangleright \Sigma_i \parallel \Sigma_i' \quad comp(\{\Sigma_i;\Sigma_i'\}_i) \quad \mathbf{x}:\tilde{U}\tilde{\mathcal{T}} \in \Sigma_i \quad \Gamma \vdash \tilde{v}:\tilde{U}}{\Gamma,\widetilde{P} \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c}:\tilde{\mathcal{T}}}\ [\textsc{State}]$$

$$\frac{\Gamma,\widetilde{P} \vdash \mathbf{X}_i \triangleright \Delta_i \quad (i=1,2)}{\Gamma,\widetilde{P} \vdash \mathbf{X}_1 \mid \mathbf{X}_2 \triangleright \Delta_1 \mid \Delta_2}\ [\textsc{Par}] \qquad \frac{\Gamma,a:\langle \mathbf{G}\rangle,\widetilde{P} \vdash \mathbf{X} \triangleright \Delta}{\Gamma,\widetilde{P} \vdash (\nu a)\mathbf{X} \triangleright \Delta}\ [\textsc{Res}]$$

$$\frac{\vdash P_i \triangleright \Sigma_i;\Sigma_i' \quad comp(\{\Sigma_i;\Sigma_i'\}_i)}{\Gamma,\widetilde{P} \vdash \mathbf{0} \triangleright \tilde{c}:\mathsf{def}\ \mathbf{x} = \mathsf{end}\ \mathsf{in}\ \mathbf{x}}\ [\textsc{Nil}] \qquad \frac{\Gamma,\widetilde{P} \vdash \mathbf{X} \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \triangleright \Delta}\ [\textsc{RDef}]$$

Figure 4.14.: Typing Rules for Run-Time State Processes

$$\frac{}{\Gamma \vdash_{\{s\}} s:\varepsilon \triangleright \emptyset}\ [\textsc{QInit}]$$

$$\frac{\Gamma \vdash_{\{s\}} s:h \triangleright \Delta \qquad \Gamma \vdash \tilde{v}:U}{\Gamma \vdash_{\{s\}} s:h\cdot(\mathtt{q},\Pi,l\langle v\rangle) \triangleright \Delta;\{s[\mathtt{q}]:!\langle \mathtt{p},l\langle U\rangle\rangle\}}\ [\textsc{QVal}]$$

$$\frac{\Gamma \vdash_{\{s\}} s:h \triangleright \Delta}{\Gamma \vdash s:h\cdot(\mathtt{q},\mathtt{p},l\langle s'[\mathtt{p}']\rangle) \triangleright (\Delta,s'[\mathtt{p}']:\mathcal{T}');\{s[\mathtt{q}]:!\langle \mathtt{p},l\langle \mathcal{T}'\rangle\rangle\}}\ [\textsc{QDeleg}]$$

Figure 4.15.: Typing System for Queues

$\mid$ denotes the parallel composition of two definition types. or concatenating a message type to the top of the definition type.

We define choreographical session types which have both message and completed types.

$$\begin{array}{rrcll} \text{Generalised} & \mathsf{T} & ::= & T & \textit{session} \\ & & \mid & \mathtt{T} & \textit{message} \\ & & \mid & \mathtt{T};T & \textit{continuation} \end{array}$$

Then ; is defined by:

$$\Delta;\{s[\mathtt{q}:r]:\mathtt{T}\} = \begin{cases} \Delta',s[\mathtt{q}:r]:\mathtt{T}';\mathtt{T} & \text{if } \Delta = \Delta',s[\mathtt{q}:r]:\mathtt{T}', \\ \Delta,s[\mathtt{q}:r]:\mathtt{T} & \text{otherwise.} \end{cases}$$

Typing Systems The typing of queues and runtime state are similar with the definition agent, shown in Figure 4.15 and Figure 4.16. The typing for network processes and for queues are essentially the same as the typing system

$$\frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash_{\varnothing} P \triangleright \Delta}\ [\textsc{Prom}] \qquad \frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta \quad \Delta \approx \Delta'}{\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta'}\ [\textsc{Eq}]$$

$$\frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta \quad \Gamma \vdash_{\mathcal{S}'} \mathbf{N}' \triangleright \Delta'}{\Gamma \vdash_{\mathcal{S} \uplus \mathcal{S}'} \mathbf{N} \,||\, \mathbf{N}' \triangleright \Delta \mid \Delta'}\ [\textsc{GPar}] \qquad \frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta \quad co(s, \Delta)}{\Gamma \vdash_{\mathcal{S} \setminus s} (\nu s)\mathbf{N} \triangleright \Delta \setminus s}\ [\textsc{GSRes}]$$

$$\frac{\Gamma, a : \langle \mathbf{G} \rangle \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta}{\Gamma \vdash_{\mathcal{S}} (\nu a)\mathbf{N} \triangleright \Delta}\ [\textsc{GNRes}] \qquad \frac{\Gamma \vdash a : \langle \mathbf{G} \rangle \quad \mathbf{G} \upharpoonright \mathtt{p} = \mathcal{T}}{\Gamma \vdash_{\emptyset} a[\mathtt{p}]\langle s \rangle \triangleright s[\mathtt{p}] : \mathcal{T}}\ [\textsc{GInit}]$$

Figure 4.16.: Typing System for Run-Time State Processes

in [CDCYP15] since at the network level processes can be seen as the same as the original session calculus.

Transition Rules between Environments

We define the transition system between environments in Figure 4.17.

$$\frac{\mathcal{T}_0 \equiv \mathcal{T} \quad \xrightarrow{\ell} \quad \mathcal{T}' \equiv \mathcal{T}_0'}{\mathcal{T}_0 \quad \xrightarrow{\ell} \quad \mathcal{T}_0'}\ \lfloor \textsc{Str} \rfloor$$

$$(\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta) \quad [\textsc{Tau}]$$

$$(\Gamma, a : \langle \mathbf{G} \rangle, \Delta) \xrightarrow{a\langle \mathbf{G} \rangle} (\Gamma, a : \langle \mathbf{G} \rangle, \Delta) \quad [\textsc{Init}]$$

$$(\Gamma, a : \langle \mathbf{G} \rangle, \Delta) \xrightarrow{a\langle s \rangle[\mathtt{p}]} (\Gamma, a : \langle \mathbf{G} \rangle, \Delta, s[\mathtt{p}] : \mathbf{G} \upharpoonright \mathtt{p}) \quad [\textsc{Acc}]$$

$$\frac{\mathcal{T} \xrightarrow{!\langle \mathtt{q}, l\langle U \rangle\rangle} \mathcal{T}' \quad \Gamma \vdash v : U}{(\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}) \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}')}\ [\textsc{Send}]$$

$$\frac{\mathcal{T} \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle\rangle} \mathcal{T}' \quad \Gamma \vdash v : U}{(\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}) \xrightarrow{s[\mathtt{q},\mathtt{p}]?l\langle v \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}')}\ [\textsc{Recv}]$$

$$\frac{\mathcal{T} \xrightarrow{!\langle \mathtt{q}, l\langle \mathcal{T}_s \rangle\rangle} \mathcal{T}'}{(\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}, s'[\mathtt{p}'] : \mathcal{T}_s) \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}')}\ [\textsc{Deleg}]$$

$$\frac{\mathcal{T} \xrightarrow{?\langle \mathtt{p}, l\langle \mathcal{T}_r \rangle\rangle} \mathcal{T}'}{(\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}) \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T}', s'[\mathtt{p}'] : \mathcal{T}_r)}\ [\textsc{Catch}]$$

$$\frac{\mathcal{T}_1 \xrightarrow{!\langle \mathtt{q}, l\langle U \rangle\rangle} \mathcal{T}_1' \qquad \mathcal{T}_2 \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle\rangle} \mathcal{T}_2'}{(s[\mathtt{p}] : \mathcal{T}_1, s[\mathtt{q}] : \mathcal{T}_2, \Delta) \quad \to \quad (s[\mathtt{p}] : \mathcal{T}_1', s[\mathtt{q}] : \mathcal{T}_2', \Delta)}\ [\textsc{Com}]$$

Figure 4.17.: Environment Transition System

$$\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x} \quad \equiv \quad \mathsf{def}\ \widetilde{T}'\ \mathsf{in}\ \mathbf{x} \qquad (\widetilde{T} = \widetilde{T}') \qquad \lfloor \textsc{Eq} \rfloor$$

$$\mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i \quad \equiv \quad \mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \qquad (i = 1 \text{ or } i = 2) \qquad \lfloor \textsc{Merge} \rfloor$$

$$\mathsf{def}\ \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \mid \mathbf{x}_2 \quad \equiv \quad \mathsf{def}\ \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \qquad \lfloor \textsc{Join} \rfloor$$

$$\mathsf{def}\ \mathbf{x} = !\langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x} \xrightarrow{!\langle \mathtt{p}, l\langle U \rangle \rangle} \mathsf{def}\ \mathbf{x} = !\langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}' \qquad \lfloor \textsc{Send}_\ell \rfloor$$

$$\mathsf{def}\ \mathbf{x} = ?\langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x} \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle \rangle} \mathsf{def}\ \mathbf{x} = ?\langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}' \qquad \lfloor \textsc{Recv}_\ell \rfloor$$

$$\mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \xrightarrow{\tau} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i \qquad (i = 1 \text{ or } i = 2) \qquad \lfloor \textsc{Cond} \rfloor$$

$$\frac{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \xrightarrow{\ell} \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1'}{\mathsf{def}\ \mathbf{x} = \mathbf{x}_1\ \&\ \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \xrightarrow{\ell} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1\ \&\ \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1'} \qquad \lfloor \textsc{Choice} \rfloor$$

$$\frac{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \xrightarrow{\ell} \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1'}{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \mid \mathbf{X}_2 \xrightarrow{\ell} \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1' \mid \mathbf{X}_2} \qquad \lfloor \textsc{Par} \rfloor$$

$$\frac{\mathcal{T}_1 \xrightarrow{!\langle \mathtt{q}, l\langle U \rangle \rangle} \mathcal{T}_1' \qquad \mathcal{T}_2 \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle \rangle} \mathcal{T}_2'}{(s[\mathtt{p}] : \mathcal{T}_1, s[\mathtt{q}] : \mathcal{T}_2, \Delta) \rightarrow (s[\mathtt{p}] : \mathcal{T}_1', s[\mathtt{q}] : \mathcal{T}_2', \Delta)} \qquad [\textsc{Com}]$$

Figure 4.18.: Type Transition System, Structure Rules and Environment Communication Rule

In Figure 4.17, rules $\lfloor \textsc{Init,Acc} \rfloor$ are used for the session initiations and receptions. Others shows transitions between local types to transitions between environments. Other rules are straightforward.

4.5. Safety of Typed Multiparty Session Processes

We defines a labelled transition relation between types $\mathcal{T} \xrightarrow{\ell} \mathcal{T}'$ in Figure 4.18, with transition labels ℓ for types: $\ell ::= \tau \mid !\langle \mathtt{p}, l\langle U \rangle \rangle \mid ?\langle \mathtt{p}, l\langle U \rangle \rangle$. Rules $\lfloor \textsc{Send}_l \rfloor$ describes that the sending action occur when the state variable $\mathbf{x}$ points to sending types. Rule $\lfloor \textsc{Recv}_l \rfloor$ is similar. Others are contextual rules. We also use the labelled transition relation between environments, denoted by $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$ and the main rule for transition is $\lfloor \textsc{Com} \rfloor$ in Figure 4.18 which represents the reduction between a message queue and a process at the network level. Other omitted rules are straightforward.

We say P has ***a type error*** if expressions in P contain either a type error for a value or constant in the standard sense, such as $(\mathtt{true} + 7)$, or ***a reception error***, e.g. the sender sends a value with label l_0 while the receiver does not expect label

l_0.

We first state the subject congruence.

Lemma 4.5.1 (Subject Congruence)**.** *Suppose* $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}'$. *Then* $\Gamma, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta$. *Similarly for P and* **N**.

Proof. 1. (Process P) For the axioms in Figure 4.13, we use the type equalities in Figure 4.8. Others are straightforward by induction.

2. (Def agents P) For the case of join and merge, we use the structure rule for local types defined in Figure 4.18.

3. (Network **N**) The same as [CDCYP15].

□

The theorem of subject transition and reduction, which is also called *type soundness*, states that if a process emits a label, or the network performs a reduction, then the environment should do the corresponding action, and after transition, the resulting process and the environment should match.

Theorem 4.5.2 (Subject Transition and Reduction)**.**

1. $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$ *imply* $\Gamma', \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
2. $\Gamma \vdash P \triangleright \Delta$ *and* $P \xrightarrow{\alpha} P'$ *imply* $\Gamma' \vdash P' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
3. $\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta$ *and* $\mathbf{N} \longrightarrow \mathbf{N}'$ *imply* $\Gamma \vdash_{\mathcal{S}} \mathbf{N}' \triangleright \Delta'$ *with* $\Delta \longrightarrow^* \Delta'$.

Proof. We prove the theorem by case analysis approach. **(Process P)**

Case [SEND] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = x\,!\,\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v \rangle} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, $e[\tilde{v}/\tilde{y}] \downarrow v$ and $\tilde{e}[\tilde{v}/\tilde{y}] \downarrow \tilde{w}$ by [SEND]. Then by [STATE] in Figure 4.14,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \widetilde{T}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = x\,!\,\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\,\widetilde{T} \;\; [\!] \;\; \mathbf{x}' : \tilde{U}'\,\widetilde{T}'$$

with

$$\tilde{y} : \tilde{U} \vdash e : U, \quad \tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}',$$

and

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =! \langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U} \vdash e[\tilde{v}/\tilde{y}] : U, e_i[\tilde{v}/\tilde{y}] : U_i$$

On the other hand, by $\lfloor \textsc{Send}_\ell \rfloor$ in Figure 4.18, we have

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =! \langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}' \xrightarrow{! \langle \mathtt{p}, l\langle U \rangle \rangle} \mathcal{T}_i'$$

This implies, by [Send] in Figure 4.17, with $\Gamma \vdash v : U$ and $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, we have:

$$(\Gamma, \Delta) \xrightarrow{s[\mathtt{q},\mathtt{p}]! l\langle v \rangle} (\Gamma, \Delta')$$

where $\Delta = \tilde{c} : \tilde{\mathcal{T}}$ and $\Delta' = \Delta \setminus s[\mathtt{q}] \cup \{s[\mathtt{q}] : \mathcal{T}_i'\}$ as desired.

Case [Recv] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = x ? \langle \mathtt{p}, l(y) \rangle . \mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{s[\mathtt{q},\mathtt{p}]? l\langle v \rangle} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$ and $\tilde{e}[\tilde{v}/\tilde{x}][v/y] \downarrow \tilde{w}$ by [Recv]. Then by [State] in Figure 4.14,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \widetilde{\mathcal{T}}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = x ? \langle \mathtt{p}, l(y) \rangle . \mathbf{x}'(\tilde{e}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\, \widetilde{\mathcal{T}} \;\; [\!] \;\; \mathbf{x}' : \tilde{U}'\, \widetilde{\mathcal{T}}'$$

with

$$\tilde{y} : \tilde{U}, y : U \vdash \tilde{e} : \tilde{U}'$$

and

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =? \langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U}, v : U \vdash e[\tilde{v}/\tilde{y}] : U, e_i[\tilde{v}/\tilde{y}] : U_i$$

On the other hand, by $\lfloor \text{RECV}_\ell \rfloor$ in Figure 4.12, we have

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =? \langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}' \quad \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle \rangle} \quad \mathcal{T}_i'$$

This implies, by [RECV] in Figure 4.17, with $\Gamma \vdash v : U$ and $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, we have:

$$(\Gamma, \Delta) \xrightarrow{s[\mathtt{q},\mathtt{p}]?l\langle v \rangle} (\Gamma, \Delta')$$

where $\Delta = \tilde{c} : \tilde{\mathcal{T}}$ and $\Delta' = \Delta \setminus s[\mathtt{q}] \cup \{s[\mathtt{q}] : \mathcal{T}_i'\}$ as desired.

Case [INIT]

Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{w}\tilde{c}) \quad \xrightarrow{a\langle \mathbf{G} \rangle} \quad \mathbf{x}'(\tilde{v}\tilde{c})$$

with $a = x[\tilde{w}/\tilde{y}]$, $\tilde{e}[\tilde{w}/\tilde{y}] \downarrow \tilde{v}$, By [INIT] in Figure 4.12,

$$\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \triangleright \mathbf{x} : \tilde{U}\, \widetilde{\mathcal{T}} \;\; [\!] \;\; \mathbf{x}' : \tilde{U}'\, \widetilde{\mathcal{T}}'$$

with $\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}'$, $\tilde{y} : \tilde{U} \vdash x : \langle \mathbf{G} \rangle$ and $\forall i, \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}'$. Then by [STATE] in Figure 4.14, we have:

$$\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{w}\tilde{c}) \triangleright \tilde{c} : \tilde{\mathcal{T}}$$

and, by [INIT] in Figure 4.17, we have

$$(\Gamma, \Delta) \quad \xrightarrow{a\langle \mathbf{G} \rangle} \quad (\Gamma, \Delta)$$

By [STATE] in Figure 4.14, we have

$$\Gamma, \widetilde{P} \vdash \mathbf{x}'(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \tilde{\mathcal{T}}$$

as desired, noting $\mathcal{T}_i' \equiv \mathcal{T}_i$.

Case [ACCEPT]

Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \vdash \mathbf{x}(\tilde{w}\tilde{c}) \quad \xrightarrow{a[\mathtt{p}]\langle s \rangle} \quad \mathbf{x}'(\tilde{v}\tilde{c}s)$$

with $a = x[\tilde{w}/\tilde{y}]$, $\tilde{e}[\tilde{w}/\tilde{y}] \downarrow \tilde{v}$, By [REQ] in Figure 4.12,

$$\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \triangleright \mathbf{x} : \tilde{U}\,\widetilde{\mathcal{T}} \;\; [\!] \;\; \mathbf{x}' : \tilde{U}'\,\widetilde{\mathcal{T}}'\mathcal{T}$$

with $\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}'$, $\tilde{y} : \tilde{U} \vdash x : \langle \mathbf{G} \rangle$, $\forall i, \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}'$ and $\mathcal{T} = \mathbf{G} \upharpoonright \mathtt{p}$. Then by [STATE] in Figure 4.14, we have:

$$\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{w}\tilde{c}) \triangleright \tilde{c} : \tilde{\mathcal{T}}$$

and, by [ACC] in Figure 4.17, we have

$$(\Gamma, \Delta) \quad \xrightarrow{a[\mathtt{p}]\langle s \rangle} \quad (\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T})$$

By [STATE] in Figure 4.14, we have

$$\Gamma, \widetilde{P} \vdash \mathbf{x}'(\tilde{v}\tilde{c}s) \triangleright \tilde{c} : \tilde{\mathcal{T}}, s[\mathtt{p}] : \mathcal{T}$$

as desired, noting $\mathcal{T}_i' \equiv \mathcal{T}_i$.

Case [IFT] Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}_1(\tilde{e}_1\tilde{z})\ \mathtt{else}\ \mathbf{x}_2(\tilde{e}_2\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{\tau} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $e[\tilde{v}/\tilde{y}] \downarrow \mathsf{true}$ and $\tilde{e}_1[\tilde{v}/\tilde{y}] \downarrow \tilde{w}$ by [IFT]. Then by [STATE] in Figure 4.14,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \widetilde{\mathcal{T}}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}_1(\tilde{e}_1\tilde{z})\ \mathtt{else}\ \mathbf{x}_2(\tilde{e}_2\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\,\widetilde{\mathcal{T}} \;\; [\!] \;\; \mathbf{x}_1 : \tilde{U}_1\,\widetilde{\mathcal{T}_1}, \mathbf{x}_2 : \tilde{U}_2\,\widetilde{\mathcal{T}_2}$$

with

$$\tilde{y} : \tilde{U} \vdash e : U \quad \tilde{y} : \tilde{U} \vdash \tilde{e}_1 : \tilde{U}_1 \quad \tilde{y} : \tilde{U} \vdash \tilde{e}_2 : \tilde{U}_2$$

and

$$\forall i, \mathcal{T}_i = (\mathcal{T}_{1i} \cup \mathcal{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U} \vdash e_1[\tilde{v}/\tilde{y}] : U_1$$

On the other hand, by $\lfloor$[Cond]$\rfloor$ in Figure 4.18, we have

$$\mathcal{T}_i = (\mathcal{T}_{1i} \cup \mathcal{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2 \quad \xrightarrow{\tau} \quad \mathcal{T}_i'$$

This implies, by [Tau] in Figure 4.17, we have:

$$(\Gamma, \Delta) \quad \xrightarrow{\tau} \quad (\Gamma, \Delta)$$

where $\Delta = \tilde{c} : \tilde{T}$.

Case [New] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = (\nu a)\mathbf{x}'(a\tilde{y}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{\tau} (\nu a)\mathbf{x}'(a\tilde{v}\tilde{c})$$

with $a \notin \tilde{v}$ by [New]. Then by [State] in Figure 4.14,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \widetilde{\mathcal{T}}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = (\nu a)\mathbf{x}'(a\tilde{y}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\, \widetilde{\mathcal{T}} \;\; [\!] \; \mathbf{x}' : \langle \mathbf{G} \rangle \tilde{U}\, \widetilde{\mathcal{T}}'$$

with

$$\forall i, \widetilde{\mathcal{T}_i}' = \widetilde{\mathcal{T}_i} \uplus \mathbf{x} = \mathbf{x}'$$

and $z_i = x$. On the other hand, by $\lfloor$[Eq]$\rfloor$ in Figure 4.18, we have

$$\widetilde{\mathcal{T}_i}' = \widetilde{\mathcal{T}_i} \uplus \mathbf{x} = \mathbf{x}' \quad \xrightarrow{\tau} \quad \mathcal{T}_i'$$

This implies, by [Tau] in Figure 4.17, we have:

$$(\Gamma, \Delta) \quad \xrightarrow{\tau} \quad (\Gamma, \Delta)$$

where $\Delta = \tilde{c} : \tilde{T}$.

Case [EXT] Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{x}_1(\tilde{y}\tilde{z}) \;\&\; \mathbf{x}_2(\tilde{y}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{\alpha} \mathbf{x}_1(\tilde{v}\tilde{c})$$

by [EXT]. Then by [STATE] in Figure 4.14,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \widetilde{T}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{x}_1(\tilde{y}\tilde{z}) \;\&\; \mathbf{x}_2(\tilde{y}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\,\widetilde{T} \;\; [\!] \;\; \mathbf{x}_1 : \tilde{U}\,\widetilde{T_1}, \mathbf{x}_2 : \tilde{U}\,\widetilde{T_2}$$

with

$$\forall i, T_i = (T_{1i} \cup T_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \;\&\; \mathbf{x}_2$$

with $z_i = x$. On the other hand, by $\lfloor$CHOICE$\rfloor$ in Figure 4.18, we have

$$T_i = (T_{1i} \cup T_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \;\&\; \mathbf{x}_2 \quad \xrightarrow{\alpha} \quad T'_{1i}$$

This implies, by [TAU] in Figure 4.17, we have:

$$(\Gamma, \Delta) \quad \xrightarrow{\alpha} \quad (\Gamma, \Delta')$$

where $\Delta = \tilde{c} : \tilde{T}$ and $\Delta' = \Delta \setminus \{\mathbf{x}_2 : \tilde{U}\,\widetilde{T_2}\}$.

Case [PAR]

By the premise of rule [PAR] in Figure 4.10

$$\widetilde{P} \vdash \mathbf{X}_0 \xrightarrow{\alpha} \mathbf{X}'_0$$

, where we could obtain that

$$\Gamma, \widetilde{P} \vdash \mathbf{X}_0 \triangleright \Delta_0$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}'_0 \triangleright \Delta'_0$$

such that $(\Gamma, \Delta_0) \xrightarrow{\alpha} (\Gamma, \Delta'_0)$.

We assume that there exists

$$\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$$

Then, by applying the rule $\lfloor$PAR$\rfloor$ in Figure 4.14, we get

$$\Gamma, \widetilde{P} \vdash \mathbf{X}_0 \mid \mathbf{X} \triangleright \Delta_0 \mid \Delta$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}'_0 \mid \mathbf{X} \triangleright \Delta'_0 \mid \Delta$$

, which satisfies $(\Gamma, \Delta_0 \mid \Delta) \xrightarrow{\alpha} (\Gamma, \Delta'_0 \mid \Delta)$.

Case [WEAK]

By the premise of rule [WEAK], we are given

$$\widetilde{P}' \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$$

, where we could obtain that

$$\Gamma, \widetilde{P}' \vdash \mathbf{X}_0 \triangleright \Delta_0$$
$$\Gamma, \widetilde{P}' \vdash \mathbf{X}'_0 \triangleright \Delta'_0$$

such that $(\Gamma, \Delta_0) \xrightarrow{\alpha} (\Gamma, \Delta'_0)$.

Then, by applying the rule [WEAK] in Figure 4.10, we get

$$\widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$$

, where we have

$$\Gamma, \widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X}_0 \triangleright \Delta_0$$
$$\Gamma, \widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X}'_0 \triangleright \Delta'_0$$

, which also satisfies $(\Gamma, \Delta_0) \xrightarrow{\alpha} (\Gamma, \Delta'_0)$.

Case [STR]

By the premise of rule [STR], we are given

$$\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}_0 \xrightarrow{\alpha} \mathbf{X}'_0 \equiv \mathbf{X}'$$

, where we could obtain that

$$\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}_0 \triangleright \Delta_0$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}'_0 \triangleright \Delta'_0$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$$

with the transition $(\Gamma, \Delta_0) \xrightarrow{\alpha} (\Gamma, \Delta'_0)$.

By applying Lemma 5.5.1, we could get that

$$\Gamma, \widetilde{P} \vdash \mathbf{X}_0 \triangleright \Delta \qquad \Delta \equiv \Delta_0$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'_0 \qquad \Delta'_0 \equiv \Delta'$$

Then, by applying the rule $\lfloor$STR$\rfloor$ in section 4.4.3, we get

$$\mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$$

Then we could obtain

$$\Gamma, \widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$$

with the transition $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma, \Delta')$.

Case [RES]

By the premise of rule [RES] in Figure 4.10

$$\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$$

, where we could obtain that

$$\Gamma, a : \langle \mathbf{G} \rangle, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$$
$$\Gamma, a : \langle \mathbf{G} \rangle, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$$

with the transition $(\Gamma, \Delta_0) \xrightarrow{\alpha} (\Gamma, \Delta'_0)$.

Then, by applying the rule $\lfloor$PAR$\rfloor$ in Figure 4.14, we get

$$\Gamma, \widetilde{P} \vdash (\nu a)\mathbf{X} \triangleright \Delta$$
$$\Gamma, \widetilde{P} \vdash (\nu a)\mathbf{X}' \triangleright \Delta'$$

, which satisfies $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma, \Delta')$.

(Def agent P)

Case [DEF]

Given the premise of the rule [DEF] in Figure 4.10, we have

$$\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$$

Assume that

$$\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$$
$$\Gamma, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$$

and we have the transition $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma, \Delta')$.

Then, by applying the rule [STATE] in Figure 4.14 to the above, we obtain that

$$\Delta \text{ is the set of } \{\Sigma_i \parallel \Sigma_i'\} \text{ for each } P_i$$
$$\Delta' \text{ is the set of } \{\Sigma_i \parallel \Sigma_i'\} \text{ for each } P_i$$

Again, we assume that

$$\Gamma, \widetilde{P} \vdash \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \triangleright \Delta_{def}$$
$$\Gamma, \widetilde{P} \vdash \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}' \triangleright \Delta'_{def}$$

By the premise of the rule $\lfloor$DEF$\rfloor$ in Figure 4.12, we have

$$\Delta_{def} \text{ is the set of } \{\Sigma_i \parallel \Sigma_i'\} \text{ for each } P_i$$
$$\Delta'_{def} \text{ is the set of } \{\Sigma_i \parallel \Sigma_i'\} \text{ for each } P_i$$

This implies that $\Delta \equiv \Delta_{def}$, $\Delta' \equiv \Delta'_{def}$.

By the reduction rule $\lfloor$DEF$\rfloor$ in Figure 4.10, we obtain

$$\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \xrightarrow{\alpha} \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}'$$

, which we have $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma, \Delta')$.

Case [TAU]

By the rules $\lfloor$TAU$\rfloor$ in Figure 4.10 and $\lfloor$TAU$\rfloor$ in Figure 4.17, we obtain that for every transition, $P \xrightarrow{\tau} P$, there exists an environment transition $(\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta)$.

(Network N)

Case [PUT] Assume

$$P \mid\mid s : h \to P' \mid\mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)$$

with

$$P \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v \rangle} P'$$

Since there is a typed transition, $s[\mathtt{p}]$ in P has a type

$$\mathcal{T}_1 \stackrel{\text{def}}{=} (\mathsf{def}\ \mathbf{x} = !\langle \mathtt{p}_n, l_1\langle U_n \rangle \rangle.\mathbf{x}_1, \widetilde{T}\ \mathsf{in}\ \mathbf{x})$$

and $s[\mathtt{p}]$ in $s : h$ has a type

$$\mathtt{T}_1 \stackrel{\text{def}}{=} !\langle \mathtt{p}_1, l_1\langle U_1 \rangle \rangle; \cdots; !\langle \mathtt{p}_{n-1}, l_1\langle U_{n-1} \rangle \rangle$$

On the other hand, $s[\mathtt{p}]$ in P' has a type:

$$\mathcal{T}_2 \stackrel{\text{def}}{=} (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1)$$

and $s[\mathtt{p}]$ in $s : h \cdot\cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)$ has a type

$$\mathtt{T}_2 \stackrel{\text{def}}{=} !\langle \mathtt{p}_1, l_1\langle U_1 \rangle \rangle; \cdots !\langle \mathtt{p}_{n-1}, l_1\langle U_{n-1} \rangle \rangle \cdot !\langle \mathtt{p}_n, l_1\langle U_n \rangle \rangle$$

Then by the parallel composition rule, we have $\mathcal{T}_2 \mid \mathtt{T}_2 = \mathcal{T}_1 \mid \mathtt{T}_1$, as desired.

Case [GET] Similar with [PUT].

Assume

$$P \mid\mid s : (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \cdot h \to P' \mid\mid s : h$$

with

$$P \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v \rangle} P'$$

Since there is a typed transition, $s[\mathtt{p}]$ in P has a type

$$\mathcal{T}_1 \stackrel{\text{def}}{=} (\mathsf{def}\ \mathbf{x} = ?\langle \mathtt{p}_n, l_1\langle U_n \rangle \rangle.\mathbf{x}_1, \widetilde{T}\ \mathsf{in}\ \mathbf{x})$$

and $s[\mathtt{p}]$ in $s : h \cdot\cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)$ has a type

$$\mathtt{T}_1 \stackrel{\text{def}}{=} ?\langle \mathtt{p}_1, l_1\langle U_1 \rangle \rangle; \cdots; ?\langle \mathtt{p}_{n-1}, l_1\langle U_{n-1} \rangle \rangle$$

On the other hand, $s[\mathtt{p}]$ in P' has a type:

$$\mathcal{T}_2 \stackrel{\text{def}}{=} (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1)$$

and $s[\mathtt{p}]$ in $s : h$ has a type

$$\mathtt{T}_2 \stackrel{\text{def}}{=} ?\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle; \cdots ?\langle \mathtt{p}_{n-1}, l_1\langle U_{n-1}\rangle\rangle \cdot ?\langle \mathtt{p}_n, l_1\langle U_n\rangle\rangle$$

Then by the parallel composition rule, we have $\mathcal{T}_2 \mid \mathtt{T}_2 = \mathcal{T}_1 \mid \mathtt{T}_1$, as desired.

Case [INIT$_N$]

Easy using [GINIT], and noting a parallel composition of $a[\mathtt{p}_i]\langle s\rangle$ forms the complete type $co(s,\Delta)$ with $\Delta = \{s[\mathtt{p}_i] : \mathcal{T}_i\}_i$, hence we can close by name hiding.

Case [ACC$_N$] .

By [GINIT] in Figure 4.16, we can assume $\Gamma \vdash P \mid\mid a[\mathtt{p}]\langle s\rangle \triangleright \Delta, s[\mathtt{p}] : \mathbf{G} \upharpoonright \mathtt{p}$ with $\Gamma \vdash a : \langle \mathbf{G}\rangle$. Then by [REQ] in Figure 4.12, $\Gamma \vdash P' \triangleright \Delta, s[\mathtt{p}] : \mathbf{G} \upharpoonright \mathtt{p}$, as required.

Case [RES$_N$]

This is divided into name binding case and session binding case.

- For name binding case, the transition $(\nu a)\ \mathbf{N} \to (\nu a)\ \mathbf{N}'$ is given.

  By applying the rule [GNRES] in Figure 4.15 on $(\nu a)\ \mathbf{N}$ and $(\nu a)\ \mathbf{N}'$ seperately, we get

  $$\frac{\Gamma, a : \langle \mathbf{G}\rangle \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta}{\Gamma \vdash_{\mathcal{S}} (\nu\, a)\mathbf{N} \triangleright \Delta}$$

  $$\frac{\Gamma, a : \langle \mathbf{G}\rangle \vdash_{\mathcal{S}} \mathbf{N}' \triangleright \Delta'}{\Gamma \vdash_{\mathcal{S}} (\nu\, a)\mathbf{N}' \triangleright \Delta'}$$

  Then, we apply the rule [RES$_N$] in Figure 4.10 and get transition $\mathbf{N} \to \mathbf{N}'$.

  Since this is a typed transition, there exists an environment transition as $(\Gamma, a : \langle \mathbf{G}\rangle, \Delta) \to (\Gamma, a : \langle \mathbf{G}\rangle, \Delta')$, which implies the transition $(\Gamma, \Delta) \to (\Gamma, \Delta')$, as desired.

- The session binding case is similar.

  The transition $(\nu s)\ \mathbf{N} \to (\nu s)\ \mathbf{N}'$ is given.

  By applying the rule [GSRES] in Figure 4.15 on $(\nu s)\ \mathbf{N}$ and $(\nu s)\ \mathbf{N}'$ seperately,

we get

$$\frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta \quad co(s,\Delta)}{\Gamma \vdash_{\mathcal{S}\setminus s} (\nu s)\mathbf{N} \triangleright \Delta \setminus s}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N}' \triangleright \Delta' \quad co(s,\Delta')}{\Gamma \vdash_{\mathcal{S}\setminus s} (\nu s)\mathbf{N}' \triangleright \Delta' \setminus s}$$

Then, we apply the rule [RES$_N$] in Figure 4.10 and get transition $\mathbf{N} \to \mathbf{N}'$.

Since this is a typed transition, there exists an environment transition as $(\Gamma,\Delta) \to (\Gamma,\Delta')$, which implies the transition $(\Gamma,\Delta \setminus s) \to (\Gamma,\Delta' \setminus s)$, as desired.

Combine the above two cases, Case **Case**[RES$_N$] is proved.

Case [PAR$_N$]

Given the transition $\mathbf{N} \,||\, \mathbf{N}'' \to \mathbf{N}' \,||\, \mathbf{N}''$, we apply the rule [GPAR] in Figure 4.15 on $\mathbf{N} \,||\, \mathbf{N}''$ and $\mathbf{N}' \,||\, \mathbf{N}''$ seperately, and get

$$\frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta \quad \Gamma \vdash_{\mathcal{S}'} \mathbf{N}'' \triangleright \Delta''}{\Gamma \vdash_{\mathcal{S}\uplus\mathcal{S}'} \mathbf{N} \,||\, \mathbf{N}'' \triangleright \Delta \mid \Delta''}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} \mathbf{N}' \triangleright \Delta' \quad \Gamma \vdash_{\mathcal{S}'} \mathbf{N}'' \triangleright \Delta''}{\Gamma \vdash_{\mathcal{S}\uplus\mathcal{S}'} \mathbf{N}' \,||\, \mathbf{N}'' \triangleright \Delta' \mid \Delta''}$$

Then, by applying the rule [PAR$_N$] in Figure 4.10 on the transition, we obtain $\mathbf{N} \to \mathbf{N}'$.

Since this is a typed transition, there exists an environment transition as $(\Gamma,\Delta) \to (\Gamma,\Delta')$, which implies the transition $(\Gamma,\Delta \mid \Delta'') \to (\Gamma,\Delta' \mid \Delta'')$, as desired.

Case [STR$_N$]

Apply the rule [STR$_N$] in Figure 4.10 on the given transition $\mathbf{N} \to \mathbf{N}'$, we obtain that

$$\mathbf{N} \equiv \mathbf{N}_0 \to \mathbf{N}_0' \equiv \mathbf{N}'$$

As $\Gamma \vdash \mathbf{N} \triangleright \Delta$, by Lemma 5.5.1, we have $\Gamma \vdash \mathbf{N}_0 \triangleright \Delta_0$ and $\Delta_0 \equiv \Delta$.

Since the transition $\mathbf{N}_0 \to \mathbf{N}_0'$ is well-typed, then we obtain that $\Gamma \vdash \mathbf{N}_0' \triangleright \Delta_0'$ and there is a transition $(\Gamma,\Delta_0) \to (\Gamma,\Delta_0')$.

Again, by Lemma 5.5.1, $\Gamma \vdash \mathbf{N}' \triangleright \Delta'$ and $\Delta_0' \equiv \Delta'$. Thus we have the transition $(\Gamma,\Delta) \to (\Gamma,\Delta')$, as desired.

This concludes all cases for Subject Transition and Reduction Theorems. □

Theorem 4.5.3 (Type Safety)**.** *Suppose* $\Gamma \vdash \mathbf{N}$*. For any* $\mathbf{N}'$ *such that* $\mathbf{N} \longrightarrow^* \mathbf{N}'$*,* $\mathbf{N}'$ *has no type error.*

Proof. Suppose $\Gamma \vdash \mathbf{N}$ has a type error. Then there are reductions such that $\mathbf{N} \longrightarrow^* (\nu s)(s : (\mathtt{p}, \mathtt{q}, l\langle v\rangle) \cdot h \mid\mid P_1 \mid\mid \cdots \mid\mid P_n) \mid\mid \mathbf{N}'$ and $P_j \xrightarrow{s[\mathtt{p},\mathtt{q}]?\langle v\rangle} P_j$ where v does not meet the specified type. Suppose $\Gamma \vdash P_j \triangleright \Delta, s[\mathtt{p}] : \mathcal{T}_j$. By the transition rules in Figure 4.9 $\mathcal{T}_j \xrightarrow{?\langle \mathtt{q}, l\langle U\rangle\rangle} \mathcal{T}_j'$. Since P_j has a type error, $\Gamma \not\vdash v : U$, which contradicts Theorem 4.5.2. □

4.6. Related Works

Other parallel extensions on session types and choreography We discuss two related works by Castagna et al. [CDCP11] and Bultan et al. [BBO12a, BBO12b], which seeks extending session to represent choreographies. The authors in [CDCP11] describe global choreographies in a language of types with fork ($\wedge$), choice ($\vee$) and repetition $(G)^*$. The corresponding local types and projection algorithm are defined and properties like liveness, soundness and completeness of choreographies are proved to be decidable. The problem is that these global types extended in [CDCP11] use series-parallel syntax trees and still lack of the ability to express general joins and merges: the syntax in [CDCP11] could represent simple examples like Ex. 4.2.3 but fail to express examples such as the Alternating Bit Protocol $\mathbf{G}_3$ in and $\mathbf{G}_4$ in Example 4.2.3.

In [BBO12a], the authors define global specifications as a finite state machine and a sound and complete decision algorithm for realising a choreography specification. They also state the sufficient conditions for checking reachability of choreographies and demonstrate the checking in several practical use cases. But the global specification in [BBO12a] lacks support for parallel composition, when modelling examples with fork/join operations. The authors do not give a practical framework, from types to processes to tackle real programs neither. Recently, [BBO12b] extends [BBO12a] to tackle the synchronisability problem. However, they still do not consider typing safety for the choreography.

The main differences between the above two works and our choreographical session types are: (1) [CDCP11, BBO12a] only investigate the relationship between global, local specifications and choreographies, without considering typing programs or processes to ensure safety properties; (2) Bultan et al. [BBO12a, BBO12b] do not mention language for any global description, while the global

specification in [CDCP11] is not rich enough compared to choreographical session types; and (3) [BBO12a, BBO12b] only propose exponential decision results and the algorithmic projectability in [CDCP11] is more limited compared with choreographical session types.

Message sequence graphs (MSGs) In terms of graphical expressiveness, *Message Sequence Graphs* (MSGs) is a very comparable representation, which is an extension of Message sequence charts (MSCs). MSGs shows finite transition systems where each state embeds a single MSC. The literature in [GMP04] investigates variants of MSGs, which provide efficient conditions for verification and implementability. Some of these conditions are similar to the those of well-formedness of global types: for example, the local choice condition with additional data of [GMP04, Def. 2] corresponds to our local choice condition. [CDCP11, § 7.1] gives a detailed comparison between MSGs and global session types.

In general, MSGs, are incomparable with our framework since MSGs' transition system is global and non-deterministic. We aim for a global type language that is compact, precise and suitable for programming. For example, extending the Alternating Bit Protocol $\mathbf{G}_{AB}$ to three parties can easily be implemented in choreographical session types but it will be written in Compositional MSGs (CMSGs), a complex extension of MSGs. The main advantage of type-based approach is that there is no gap between global specifications and programs, and we can check the properties of programs by static type-checking.

4.7. Summary

We present the theory of choreographical multiparty session types. In order to solve the problem of state explosion of large-scale protocols and better support general graphs, the separation of branching and parallel functionalities is necessary. We define the syntax of global choreographical session types, which have two features: explicit fork and choice operations and state variable notations. We describe the well-formedness conditions to guarantee that global types are safe. Similar changes are applied to local types and a projection algorithm is presented. We define the operational semantics and typing rules for choreographical session calculus, and theorems of subject reduction and type safety are proved. Research works related to choreographies, graphs and parallel extension on session types are discussed finally.

5. Timed Choreographical Multiparty Session Types

In this chapter, we show the theory of timed choreographical multiparty session types. First, we present the syntax of global session types and define the labelled transition system. Then we show the syntax of local types and the corresponding transition system. Next we define the semantics of timed choreographical session calculus and transition rules. Then we show proof rules for validating the processes and the typing system for runtime processes. Theorems of subject reduction, type safety and time error-freedom are proved. We also define the progress-enabling property and time-progress property to ensure progress of processes. Finally, related works are compared and reach the summary of this chapter.

5.1. Motivation: An Application of Time and Parallel Combination

When we are trying to describe and verify more applications using session types, it is observed that some user cases are hybrid systems that employ both real-time requirements and parallelism. Among various internet applications, control and sensor networks require precise timing (like "over the Ethernet $\pm$ 65ns") or deterministic and small delays (like "over the Ethernet $\leq 100\mu$s"), and meanwhile, communicate with other components using multiple concurrent threads. Live audio and video applications are another good examples because they require a delay for source-to-destination path and sound/images are sent simultaneously with other data formats. Besides, some programs that do not need internet connection also have similar time and parallel requirements. For example, *Sonic Pi* [son, AOB14] is a music application that use generates music via coding. Producing the music requires that beats are played at periodic and precise timing, and threads for different chords are performed concurrently and synchronised whenever required. These use cases inspire us to combine timed session types

with choreographical session types.

This combination can be divided into two steps: first we merge the syntaxes of two variants of multiparty session types, including global/local session types and processes; then we check if the theorems from either of the two frameworks still hold for the new syntax of timed choreographical session types. On one hand, merging the syntax appears natural; and it is observed that time and parallelism are orthogonal, so proofs for some theorems, such as well-formedness of global types, remain similar. On the other hand, it is challenging to prove theorems that concern time and rely on structural traces, especially for checking feasibility and wait-freedom; by introducing fork/merge operations, timers may be interleaved between parallel branches and this increases the complexity of ensuring safety regarding time. In this chapter, we will describe the solution to addressing such complexity.

Before presenting the syntax, we provide a motivating example that is used in this chapter. It describes a trading process that involves three participants: a trader `T` buys or sells securities based on real time stock prices; a Broker `B` provides a service to trader, such as offering trading software; and Exchange Center `E` is responsible for pushing current stock price to traders and provides a platform for trading securities between individual investors. For the purpose of illustration, the trading process is simplified and described as below, with a BPMN activity diagram in Figure 5.1.

- On one parallel branch, the Exchange Center `E` should *Push* the latest stock price to the trader `T` every 2 seconds.
- On the other parallel branch `T` can decide to *Buy*/*Sell* securities or do nothing (*Ignore*). If *Buy*/*Sell* is chosen, then `T` will send the quantity to the broker `B` and `B` forwards the *Action* to `E`.
- If `T` chooses *Ignore*, then it sends the information to `B` and `B` reports *Nothing* to `E`.
- The process continues until 20 seconds elapse. The Exchange Center `E` informs `T` of the market *Close* and `T` sends `B` message *Quit*.

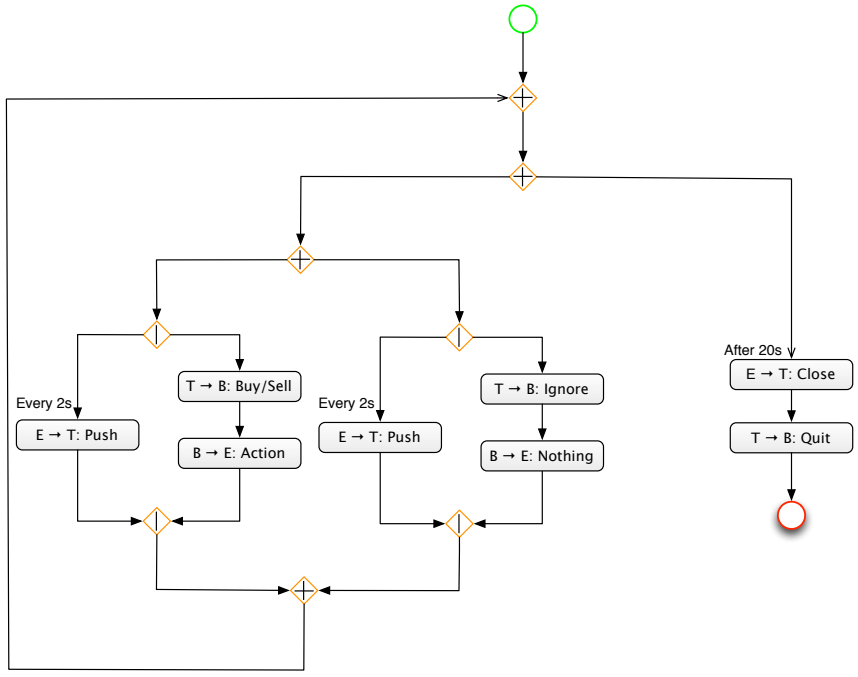


Figure 5.1.: BPMN Activity Diagram for Trading Example

5.2. Timed Choreographical Global Types

The syntax of global types is inherited from choreographical session types. The reason is that apart from time assertions, the syntax of global types for timed session types is a subset of choreographical session types; introducing fork/join enables session types to express more scenarios than merely choice/merge.

The global types are defined on the basis of choreographical multiparty sessions with flexible fork, choice, merge and join operations, with each interaction associated with a time assertion. From global types, we can project them onto local types that describe the behaviours of each single participant and are used for local verification of processes.

Before presenting global types, we define variables in time predicates, which are similar to those of Section 3.2.1. Let X be a set of *clocks* ranging over $x_1, \ldots, x_n$ and taking values in $\mathbb{R}^{\geq 0}$. A *clock assignment* $\nu : X \mapsto \mathbb{R}^{\geq 0}$ records the time of the clocks in X; $\nu + t$ as the assignment mapping to $\nu(x) + t$ for all $x \in X$. We assume ν_0 for the initial assignment mapping all clocks to 0. The *clock constraints* $\Phi(X)$ over X are:

$$\delta \quad ::= \quad \mathsf{true} \mid x > c \mid x = c \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where c is a time constant ranging in $\mathbb{Q}^{\geq 0}$. false, $<$, $\leq$, $\geq$, $\vee$ can be derived in the standard way.

Participants ($\mathtt{p}, \mathtt{q}, \mathtt{p}_1, \ldots \in \mathbb{P}$) communicate via endpoint asynchronous message passing. An interaction, which consists of a send action and a receive action, is annotated with two pairs of a clock constraint and a reset. The clock constraint specifies when that action can be executed and once the action finishes, the clocks that appear in reset predicate should be set to 0.

We assume that the sets of clocks accessible by different participants are disjoint, and that the time assertion of an action performed by a participant only concerns the clocks owned by that participant. What is different from assumptions in timed session types is that we allow each participant to have multiple clocks. Also if the participant appears in two parallel branches, then he should use different clocks for each branch before the fork point is reached.

Definition 5.2.1. The syntax of timed choreographical global types is as below.

$\mathbf{G}$	$::=$	$\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}$	Global type
G	$::=$	$\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}'$	Labelled messages
	$\mid$	$\mathbf{x} = \mathbf{x}' \mid \mathbf{x}''$	Fork
	$\mid$	$\mathbf{x} = \mathbf{x}' + \mathbf{x}''$	Choice
	$\mid$	$\mathbf{x} \mid \mathbf{x}' = \mathbf{x}''$	Join
	$\mid$	$\mathbf{x} + \mathbf{x}' = \mathbf{x}''$	Merge
	$\mid$	$\mathbf{x} = \mathsf{end}$	End
U	$::=$	$\langle \mathbf{G} \rangle \mid \mathsf{bool} \mid \mathsf{nat} \mid \cdots$	Sorts
A	$::=$	$\{\delta_o, \lambda_o, \delta_i, \lambda_i\}$	Time Assertion

Compared with choreographical multiparty session types, time assertions are added in the global types. The *labelled message exchanges* $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}'$ describes that $\mathtt{p}$ are the sending message with label l and the payload type U and $\mathtt{p}'$ is receiving the message. A is a *time assertion* $A = \{\delta_o, \lambda_o, \delta_i, \lambda_i\}$, where δ_o and λ_o are the clock constraint and reset for the output action and δ_i and λ_i are for the input action. This transition specifies that: (1) if δ_o is satisfied, then $\mathtt{p}$ can go from $\mathbf{x}$ to the continuation $\mathbf{x}'$ by sending message l and reset the clock (if λ_o is non-empty); (2) if δ_i is satisfied, $\mathtt{p}'$ goes from $\mathbf{x}$ to $\mathbf{x}'$ by receiving the message and doing reset. The syntax for fork, choice, join, merge and end remains the same as choreographical global types.

We illustrate the timed choreographical session types with the Trading example

$$
\begin{array}{rcl}
\mathbf{G}_{Trade} & = & \mathsf{def} \\
\mathbf{x}_0 + \mathbf{x}_{18} & = & \mathbf{x}_1 \\
\mathbf{x}_1 & = & \mathbf{x}_2 + \mathbf{x}_3 \\
\mathbf{x}_2 & = & \mathbf{x}_4 + \mathbf{x}_5 \\
\mathbf{x}_4 & = & \mathbf{x}_6 \mid \mathbf{x}_7 \\
\mathbf{x}_6 & = & \mathtt{E} \to \mathtt{T} : \mathit{Push}\langle \mathsf{float} \rangle \{ x_{\mathtt{E}_1} = 2, x_{\mathtt{E}_1}, 2 \leq x_{\mathtt{T}_1} \leq 2 + D, x_{\mathtt{T}_1} \}; \mathbf{x}_8 \\
\mathbf{x}_7 & = & \mathtt{T} \to \mathtt{B} : \mathit{Buy}/\mathit{Sell}\langle \mathsf{int} \rangle \{ x_{\mathtt{T}_2} \leq 1, x_{\mathtt{T}_2}, x_{\mathtt{B}_1} \leq 1 + D, _ \}; \mathbf{x}_9 \\
\mathbf{x}_9 & = & \mathtt{B} \to \mathtt{E} : \mathit{Action}\langle \mathsf{int} \rangle \{ x_{\mathtt{B}_1} \leq 2, x_{\mathtt{B}_1}, x_{\mathtt{E}_2} \leq 2 + D, x_{\mathtt{E}_2} \}; \mathbf{x}_{10} \\
\mathbf{x}_8 \mid \mathbf{x}_{10} & = & \mathbf{x}_{11} \\
\mathbf{x}_5 & = & \mathbf{x}_{12} \mid \mathbf{x}_{13} \\
\mathbf{x}_{12} & = & \mathtt{E} \to \mathtt{T} : \mathit{Push}\langle \mathsf{float} \rangle \{ x_{\mathtt{E}_1} = 2, x_{\mathtt{E}_1}, 2 \leq x_{\mathtt{T}_1} \leq 2 + D, x_{\mathtt{T}_1} \}; \mathbf{x}_{14} \\
\mathbf{x}_{13} & = & \mathtt{T} \to \mathtt{B} : \mathit{Ignore}\langle \mathsf{bool} \rangle \{ x_{\mathtt{T}_2} \leq 1, x_{\mathtt{T}_2}, x_{\mathtt{B}_1} \leq 1 + D, _ \}; \mathbf{x}_{15} \\
\mathbf{x}_{15} & = & \mathtt{B} \to \mathtt{E} : \mathit{Nothing}\langle \mathsf{bool} \rangle \{ x_{\mathtt{B}_1} \leq 2, x_{\mathtt{B}_1}, x_{\mathtt{E}_2} \leq 2 + D, x_{\mathtt{E}_2} \}; \mathbf{x}_{16} \\
\mathbf{x}_{14} \mid \mathbf{x}_{16} & = & \mathbf{x}_{17} \\
\mathbf{x}_{11} + \mathbf{x}_{17} & = & \mathbf{x}_{18} \\
\mathbf{x}_3 & = & \mathtt{E} \to \mathtt{T} : \mathit{Close}\langle \mathsf{bool} \rangle \{ x_{\mathtt{E}_3} = 20, _, 20 \leq x_{\mathtt{T}_3} \leq 20 + D, x_{\mathtt{T}_3} \}; \mathbf{x}_{19} \\
\mathbf{x}_{19} & = & \mathtt{T} \to \mathtt{B} : \mathit{Quit}\langle \mathsf{bool} \rangle \{ x_{\mathtt{T}_3} \leq 5, _, x_{\mathtt{B}_2} \leq 25 + 2D, _ \}; \mathbf{x}_{20} \\
\mathbf{x}_{20} & = & \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

Figure 5.2.: Global Types for the Trading Example

in Section 5.1 in Figure 5.2. D denotes the maximum delay between sender-to-receiver path and records as 100 μs. From the global types, we observe that: (1) the clocks used in two fork branches, such as $\mathbf{x}_6 \to \mathbf{x}_8$ and $\mathbf{x}_7 \to \mathbf{x}_{10}$, are different; (2) clocks used in two choice branches like $\mathbf{x}_4 \to \mathbf{x}_{11}$ and $\mathbf{x}_5 \to \mathbf{x}_{17}$ could overlap; (3) clocks used in two choice branches like $\mathbf{x}_2 \to \mathbf{x}_{18}$ and $\mathbf{x}_3 \to \mathbf{x}_{20}$ could be different.

The labelled transition system for global types is defined over states of the form (ν, G) with labels as

$$\ell ::= \mathtt{pq}!l\langle U \rangle \mid \mathtt{pq}?l\langle U \rangle \mid t$$

where $\mathtt{pq}!l\langle U \rangle$ is a send action that $\mathtt{p}$ sends $l\langle U \rangle$ to $\mathtt{q}$; $\mathtt{pq}?l\langle U \rangle$ is the dual receive action, and $t \in \mathbb{R}^{\geq 0}$ is a time action modelling time passing. We extend the syntax of G with $\mathtt{p} \rightsquigarrow \mathtt{q} : l\langle U \rangle \{A\}; \mathbf{x}'$ to describe the state in which message $l\langle U \rangle$ has been sent by $\mathtt{p}$ but not yet received by $\mathtt{q}$. The separation of send and receive actions is used to model the asynchronous behaviour in distributed systems. The labelled transition system for timed choreographical multiparty session types is defined in Figure 5.3.

$$\frac{A = \{\delta_o, \lambda_o, \delta_i, \lambda_i\} \quad \nu \models \delta_o \quad \nu' = [\lambda_o \mapsto 0]\nu}{(\nu, \mathbf{x} = \mathtt{p} \to \mathtt{q} : l\langle U\rangle\{A\}; \mathbf{x}') \quad \xrightarrow{\mathtt{pq}!l\langle U\rangle} \quad (\nu', \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle U\rangle\{A\}; \mathbf{x}')} \lfloor \textsc{Send} \rfloor$$

$$\frac{A = \{\delta_o, \lambda_o, \delta_i, \lambda_i\} \quad \nu \models \delta_i \quad \nu' = [\lambda_i \mapsto 0]\nu}{(\nu, \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle U\rangle\{A\}; \mathbf{x}') \quad \xrightarrow{\mathtt{pq}?l\langle U\rangle} \quad (\nu', \mathbf{x}')} \lfloor \textsc{Recv} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2) \quad \to \quad (\nu, \mathbf{x}_1)} \lfloor \textsc{Choice1} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2) \quad \to \quad (\nu, \mathbf{x}_2)} \lfloor \textsc{Choice2} \rfloor$$

$$\frac{}{(\nu, \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}) \quad \to \quad (\nu, \mathbf{x})} \lfloor \textsc{Merge} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2) \quad \to \quad (\nu, \mathbf{x}_1 \mid \mathbf{x}_2)} \lfloor \textsc{Fork} \rfloor$$

$$\frac{\mathbf{x}_1 \quad \xrightarrow{l} \quad \mathbf{x}_1' \qquad l = \mathtt{pq}!l\langle U\rangle \mid \mathtt{pq}?l\langle U\rangle}{(\nu, \mathbf{x}_1 \mid \mathbf{x}_2) \quad \xrightarrow{l} \quad (\nu', \mathbf{x}_1' \mid \mathbf{x}_2)} \lfloor \textsc{Fork-Com} \rfloor$$

$$\frac{}{(\nu, \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}) \quad \to \quad (\nu, \mathbf{x})} \lfloor \textsc{Join} \rfloor$$

$$\frac{\nu + t \models \mathtt{ready}(G)}{(\nu, G) \quad \xrightarrow{t} \quad (\nu + t, G)} \lfloor \textsc{Time} \rfloor$$

Figure 5.3.: Labelled Transition System for Timed Choreographical Global Types

Definition 5.2.2 (Ready clock constraints)**.**

$$\texttt{ready}(\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}, D) \quad (\text{with } A = \{\delta_o, \lambda_o, \delta_i, \lambda_i\}) = \begin{cases} \{\delta_o\} \cup \texttt{ready}(G_i, D \cup \{\mathtt{p}, \mathtt{q}\}) & \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \mathtt{p} \notin D & \\ \texttt{ready}(G_i, D \cup \{\mathtt{p}, \mathtt{q}\}) & \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise} & \end{cases}$$

$$\texttt{ready}(\mathtt{p} \rightsquigarrow \mathtt{q} : l\langle U\rangle\{A\}; \mathbf{x}', D) \quad (\text{with } A = \{\delta_o, \lambda_o, \delta_i, \lambda_i\}) = \begin{cases} \{\delta_i\} \cup \texttt{ready}(G, D \cup \{\mathtt{q}\}) & \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \mathtt{q} \notin D & \\ \texttt{ready}(G, D \cup \{\mathtt{q}\}) \quad \text{otherwise} & \end{cases}$$

$$\texttt{ready}(\mathbf{x} = \mathbf{x}' + \mathbf{x}'', D) = \texttt{ready}(\mathbf{x}', D) \cup \texttt{ready}(\mathbf{x}'', D)$$

$$\texttt{ready}(\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'', D) = \texttt{ready}(\mathbf{x}', D) \cup \texttt{ready}(\mathbf{x}'', D)$$

$$\texttt{ready}(\mathbf{x}, D) = \texttt{ready}(\mathbf{x}', D) = \texttt{ready}(\mathbf{x}'', D) \qquad \text{if } \mathbf{x} + \mathbf{x}' = \mathbf{x}''$$

$$\texttt{ready}(\mathbf{x}, D) = \texttt{ready}(\mathbf{x}', D') = \texttt{ready}(\mathbf{x}'', D \cup D') \qquad \text{if } \mathbf{x} \mid \mathbf{x}' = \mathbf{x}''$$

$\texttt{ready}(G, D)$ uses a set D of participants, initially empty, to keep track of the causal dependencies between actions. The ready actions are those that have no causal relationship with actions that occur earlier, syntactically. $\texttt{ready}(G, D)$ returns the set of constraints in G associated to ready actions. We write $\texttt{ready}(G)$ for $\texttt{ready}(G, \emptyset)$.

Definition 5.2.3 ($\nu + t \models \texttt{ready}(G)$)**.** We write $\nu + t \models \texttt{ready}(G)$ if for all $\{\delta_i\}_{i \in I} \in \texttt{ready}(G)$ there exists $t' \geq 0$ and $j \in I$ such that $\nu + t + t' \models \delta_j$

Definition 5.2.4 (Global runs and traces)**.** A timed run r for G_0 is an infinite sequence $(\nu_0, G_0)\,(t_1, \ell_1)\,(\nu_1, G_1)\,(t_2, \ell_2)\ldots$ of timed states (ν_i, G_i) and timed labels (ℓ_i, t_i), with $dom(\nu_i) = \{x_{\mathtt{p}_1}, x_{\mathtt{p}_2}, ..., x_{\mathtt{p}_j} \mid \mathtt{p} \in \mathcal{P}(G_0)\}$, $\ell_i \in \mathcal{L}$, and $t_i \in \mathbb{R}^{\geq 0}$, such that:

(1) $\nu_0(x) = 0\ \forall x \in dom(\nu)$,

(2) $(\nu_i, G_i)(\ell_i, t_i)\ (\nu_{i+1}, G_{i+1}) \in r$ if $(\nu_i, G_i) \xrightarrow{\ell_i} (\nu_{i+1}, G_{i+1})$ by the transitions,

(3) $t_0 = 0$, $t_i = t_{i-1} + t$ if $\ell_i = t$ for some $t \in \mathbb{R}^{\geq 0}$, and $t_i = t_{i-1}$ otherwise.

The trace of a run $TR(r)$ is the sequence of labels $(\tilde{\ell}, \tilde{t})$ of r. The traces of a timed choreographical global type $TR(G)$ is the set of all the traces $TR(r)$ such that r is a run for G.

Well-Formedness

The well-formedness is concerned with structure of global types and the structure of timed choreographical session types is based on that of choreographical session type, thus we could remove the time information from global types and then utilise the well-formedness conditions in Section 4.2.2 to check these conditions in timed choreographical global types. We define *remove*($\mathbf{G}$) as timed choreographical session types without time assertions.

$\texttt{remove}(\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}')$	::=	$\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \texttt{remove}(\mathbf{x}')$	
			Labelled messages
$\texttt{remove}(\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'')$	::=	$\mathbf{x} = \texttt{remove}(\mathbf{x}') \mid \texttt{remove}(\mathbf{x}'')$	Fork
$\texttt{remove}(\mathbf{x} = \mathbf{x}' + \mathbf{x}'')$	::=	$\mathbf{x} = \texttt{remove}(\mathbf{x}') + \texttt{remove}(\mathbf{x}'')$	Choice
$\texttt{remove}(\mathbf{x} \mid \mathbf{x}' = \mathbf{x}'')$	::=	$\mathbf{x} \mid \mathbf{x}' = \texttt{remove}(\mathbf{x}'')$	Join
$\texttt{remove}(\mathbf{x} + \mathbf{x}' = \mathbf{x}'')$	::=	$\mathbf{x} + \mathbf{x}' = \texttt{remove}(\mathbf{x}'')$	Merge
$\texttt{remove}(\mathbf{x} = \mathsf{end})$	::=	end	End

Definition 5.2.5 (Well-Formedness for Timed Choreographical Session Types)**.** A timed choreographical session types $\mathbf{G}$ is well-formed if $\texttt{remove}(\mathbf{G})$ satisfies the well-formedness conditions in Section 4.2.2.

5.3. Timed Choreographical Local Types

Local types, projected from global types, describe the actions of endpoint session that each participant must follow.

Definition 5.3.1 (Syntax of Local Types)**.** The syntax of local types is defined as

below.

$$
\begin{array}{rlll}
\mathcal{T} ::= & \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \text{local type} & B ::= \{\delta, \lambda\} \\
T ::= & \mathbf{x} = !\langle \mathtt{p}', l\langle U\rangle\{B_o\}\rangle.\mathbf{x}' & \text{send} & \\
\mid & \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\{B_i\}\rangle.\mathbf{x}' & \text{receive} & \\
\mid & \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' & \text{fork} & \\
\mid & \mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' & \text{join} & \\
\mid & \mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' & \text{internal choice} & \\
\mid & \mathbf{x} = \mathbf{x}' \ \& \ \mathbf{x}'' & \text{external choice} & \\
\mid & \mathbf{x} + \mathbf{x}' = \mathbf{x}'' & \text{merge} & \\
\mid & \mathbf{x} = \mathbf{x}' & \text{indirection} & \\
\mid & \mathbf{x} = \mathsf{end} & \text{end} &
\end{array}
$$

Definition 5.3.2 (Selected Projection Rules for Timed Choreographical Session Types)**.** We define the projection of a well-formed global type $\mathbf{G}$ to the local type of participant $\mathtt{p}$, denoted as $\mathbf{G} \upharpoonright \mathtt{p}$. Projection Rules for send/receive interactions are shown as below. The remaining rules are same as the projection rules for choreographical session types .

$$
\begin{array}{rcl}
\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x} \upharpoonright \mathtt{p} & = & \mathsf{def}\ \widetilde{G} \upharpoonright_{\widetilde{G}} \mathtt{p}\ \mathsf{in}\ \mathbf{x} \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = !\langle \mathtt{p}', l\langle U\rangle\{B_o\}\rangle.\mathbf{x}' \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p}' & = & \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\{B_i\}\rangle.\mathbf{x}' \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle\{A\}; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p}'' & = & \mathbf{x} = \mathbf{x}' \ (\mathtt{p} \notin \{\mathtt{p}, \mathtt{p}'\})
\end{array}
$$

Example 5.3.3 (Trade Example)**.** Then we could write the local types from participants Trader, Broker and Exchange Center separately in Figure 5.4.

The labelled transition system for local types is defined over local states of forms $(\nu, \mathcal{T})$ and labels l. The rules $\lfloor$Merge$\rfloor$, $\lfloor$Fork$\rfloor$, $\lfloor$Fork-Com$\rfloor$ and $\lfloor$Join$\rfloor$ are same as those in global types and Figure 5.5 shows the transition rules that are different from those in global types.

In rule $\lfloor$Time$\rfloor$, $\nu + t \models^* \mathtt{ready}(T)$ is defined as in Section 5.2 but relying on the function $\mathtt{ready}()$ on timed local types from Definition 5.3.4.

$$
\begin{array}{rrcl}
\mathcal{T}_T = & \mathsf{def}\ \mathbf{x}_0 + \mathbf{x}_{18} & = & \mathbf{x}_1 \\
& \mathbf{x}_1 & = & \mathbf{x}_2 + \mathbf{x}_3 \\
& \mathbf{x}_2 & = & \mathbf{x}_4 + \mathbf{x}_5 \\
& \mathbf{x}_4 & = & \mathbf{x}_6 \mid \mathbf{x}_7 \\
& \mathbf{x}_6 & = & ?\langle \mathtt{E}, \mathit{Push}\langle \mathsf{float} \rangle \{ 2 \leq x_{\mathtt{T}_1} \leq 2 + D, x_{\mathtt{T}_1} \} \rangle . \mathbf{x}_8 \\
& \mathbf{x}_7 & = & !\langle \mathtt{B}, \mathit{Buy}/\mathit{Sell}\langle \mathsf{int} \rangle \{ x_{\mathtt{T}_2} \leq 1, x_{\mathtt{T}_2} \} \rangle . \mathbf{x}_{10} \\
& \mathbf{x}_8 \mid \mathbf{x}_{10} & = & \mathbf{x}_{11} \\
& \mathbf{x}_5 & = & \mathbf{x}_{12} \mid \mathbf{x}_{13} \\
& \mathbf{x}_{12} & = & ?\langle \mathtt{E}, \mathit{Push}\langle \mathsf{float} \rangle \{ 2 \leq x_{\mathtt{T}_1} \leq 2 + D, x_{\mathtt{T}_1} \} \rangle . \mathbf{x}_{14} \\
& \mathbf{x}_{13} & = & !\langle \mathtt{B}, \mathit{Ignore}\langle \mathsf{int} \rangle \{ x_{\mathtt{T}_2} \leq 1, x_{\mathtt{T}_2} \} \rangle . \mathbf{x}_{16} \\
& \mathbf{x}_{14} \mid \mathbf{x}_{16} & = & \mathbf{x}_{17} \\
& \mathbf{x}_{11} + \mathbf{x}_{17} & = & \mathbf{x}_{18} \\
& \mathbf{x}_3 & = & ?\langle \mathtt{E}, \mathit{Close}\langle \mathsf{bool} \rangle \{ 20 \leq x_{\mathtt{T}_3} \leq 20 + D, x_{\mathtt{T}_3} \} \rangle . \mathbf{x}_{19} \\
& \mathbf{x}_{19} & = & !\langle \mathtt{B}, \mathit{Quit}\langle \mathsf{bool} \rangle \{ x_{\mathtt{T}_3} \leq 5,\ _ \} \rangle . \mathbf{x}_{20} \\
& \mathbf{x}_{20} & = & \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

$$
\begin{array}{rrcl}
\mathcal{T}_B = & \mathsf{def}\ \mathbf{x}_0 + \mathbf{x}_{18} & = & \mathbf{x}_1 \\
& \mathbf{x}_1 & = & \mathbf{x}_2 + \mathbf{x}_3 \\
& \mathbf{x}_2 & = & \mathbf{x}_4 + \mathbf{x}_5 \\
& \mathbf{x}_4 & = & ?\langle \mathtt{T}, \mathit{Buy}/\mathit{Sell}\langle \mathsf{float} \rangle \{ x_{\mathtt{B}_1} \leq 1 + D,\ _ \} \rangle . \mathbf{x}_9 \\
& \mathbf{x}_9 & = & !\langle \mathtt{E}, \mathit{Action}\langle \mathsf{float} \rangle \{ x_{\mathtt{B}_1} \leq 2, x_{\mathtt{B}_1} \} \rangle . \mathbf{x}_{11} \\
& \mathbf{x}_5 & = & ?\langle \mathtt{T}, \mathit{Ignore}\langle \mathsf{bool} \rangle \{ x_{\mathtt{B}_1} \leq 1 + D,\ _ \} \rangle . \mathbf{x}_{15} \\
& \mathbf{x}_{15} & = & !\langle \mathtt{E}, \mathit{Nothing}\langle \mathsf{bool} \rangle \{ x_{\mathtt{B}_1} \leq 2, x_{\mathtt{B}_1} \} \rangle . \mathbf{x}_{17} \\
& \mathbf{x}_{11} + \mathbf{x}_{17} & = & \mathbf{x}_{18} \\
& \mathbf{x}_3 & = & ?\langle \mathtt{T}, \mathit{Quit}\langle \mathsf{bool} \rangle \{ x_{\mathtt{B}_2} \leq 25 + 2D,\ _ \} \rangle . \mathbf{x}_{20} \\
& \mathbf{x}_{20} & = & \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

$$
\begin{array}{rrcl}
\mathcal{T}_E = & \mathsf{def}\ \mathbf{x}_0 + \mathbf{x}_{18} & = & \mathbf{x}_1 \\
& \mathbf{x}_1 & = & \mathbf{x}_2 + \mathbf{x}_3 \\
& \mathbf{x}_2 & = & \mathbf{x}_4 + \mathbf{x}_5 \\
& \mathbf{x}_4 & = & \mathbf{x}_6 \mid \mathbf{x}_7 \\
& \mathbf{x}_6 & = & !\langle \mathtt{T}, \mathit{Push}\langle \mathsf{float} \rangle \{ x_{\mathtt{E}_1} = 2, x_{\mathtt{E}_1} \} \rangle . \mathbf{x}_8 \\
& \mathbf{x}_7 & = & ?\langle \mathtt{B}, \mathit{Action}\langle \mathsf{int} \rangle \{ x_{\mathtt{E}_2} \leq 2 + D, x_{\mathtt{E}_2} \} \rangle . \mathbf{x}_{10} \\
& \mathbf{x}_8 \mid \mathbf{x}_{10} & = & \mathbf{x}_{11} \\
& \mathbf{x}_5 & = & \mathbf{x}_{12} \mid \mathbf{x}_{13} \\
& \mathbf{x}_{12} & = & !\langle \mathtt{T}, \mathit{Push}\langle \mathsf{float} \rangle \{ x_{\mathtt{E}_1} = 2, x_{\mathtt{E}_1} \} \rangle . \mathbf{x}_{14} \\
& \mathbf{x}_{13} & = & ?\langle \mathtt{B}, \mathit{Nothing}\langle \mathsf{int} \rangle \{ x_{\mathtt{E}_2} \leq 2 + D, x_{\mathtt{E}_2} \} \rangle . \mathbf{x}_{16} \\
& \mathbf{x}_{14} \mid \mathbf{x}_{16} & = & \mathbf{x}_{17} \\
& \mathbf{x}_{11} + \mathbf{x}_{17} & = & \mathbf{x}_{18} \\
& \mathbf{x}_3 & = & !\langle \mathtt{T}, \mathit{Close}\langle \mathsf{bool} \rangle \{ x_{\mathtt{E}_3} = 20,\ _ \} \rangle . \mathbf{x}_{20} \\
& \mathbf{x}_{20} & = & \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

Figure 5.4.: Local Types for the Trading Example

$$\frac{B=\{\delta,\lambda\} \quad \nu \models \delta \quad \nu' = [\lambda \mapsto 0]\nu}{(\nu, \mathbf{x} = !\langle \mathtt{q}, l\langle U\rangle\{B\}\rangle.\mathbf{x}') \xrightarrow{\mathtt{pq}!l\langle S\rangle} (\nu', \mathbf{x}')} \lfloor \textsc{Send} \rfloor$$

$$\frac{B=\{\delta,\lambda\} \quad \nu \models \delta \quad \nu' = [\lambda \mapsto 0]\nu}{(\nu, \mathbf{x} = ?\langle \mathtt{q}, l\langle U\rangle\{B\}\rangle.\mathbf{x}') \xrightarrow{\mathtt{pq}?l\langle U\rangle} (\nu', \mathbf{x}')} \lfloor \textsc{Recv} \rfloor$$

$$\frac{\nu + t \models \mathtt{ready}(T)}{(\nu, T) \xrightarrow{t} (\nu + t, T)} \lfloor \textsc{Time} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2) \to (\nu, \mathbf{x}_1)} \lfloor \textsc{Internal-Choice1} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2) \to (\nu, \mathbf{x}_2)} \lfloor \textsc{Internal-Choice2} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 \,\&\, \mathbf{x}_2) \to (\nu, \mathbf{x}_1)} \lfloor \textsc{External-Choice1} \rfloor$$

$$\frac{}{(\nu, \mathbf{x} = \mathbf{x}_1 \,\&\, \mathbf{x}_2) \to (\nu, \mathbf{x}_2)} \lfloor \textsc{External-Choice2} \rfloor$$

Figure 5.5.: Selected Transition Rules for Timed Choreographical Local Types

Definition 5.3.4 (Ready local clock constraints)**.**

$$\mathtt{ready}(\mathbf{x} = !\langle \mathtt{p}', l\langle U\rangle\{\delta,\lambda\}\rangle.\mathbf{x}') = \mathtt{ready}(\mathbf{x} = ?\langle \mathtt{p}, \{l.U\}_{\delta,\lambda}\rangle.\mathbf{x}') = \{\delta\}$$
$$\mathtt{ready}(\mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'') = \mathtt{ready}(\mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}'') = \mathtt{ready}(\mathbf{x}') \cup \mathtt{ready}(\mathbf{x}'')$$
$$\mathtt{ready}(\mathbf{x}) = \mathtt{ready}(\mathbf{x}') = \mathtt{ready}(\mathbf{x}'') \qquad \text{if } \mathbf{x} + \mathbf{x}' = \mathbf{x}''$$
$$\mathtt{ready}(\mathbf{x}) = \mathtt{ready}(\mathbf{x}') = \mathtt{ready}(\mathbf{x}'') \qquad \text{if } \mathbf{x} \mid \mathbf{x}' = \mathbf{x}''$$
$$\mathtt{ready}(\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'') = \mathtt{ready}(\mathbf{x}') \cup \mathtt{ready}(\mathbf{x}'')$$
$$\mathtt{ready}(\mathtt{end}) = \emptyset$$

A configuration of local timed types, ranged over by s, s', ..., is a tuple $(T_1, \ldots, T_n, \tilde{w})$ where $\tilde{w} = \{w_{ij}\}_{i \neq j \in \{1,\ldots,n\}}$ and w_{ij} is a unidirectional (possibly empty ε) unbounded queue with elements of the form $l\langle U\rangle$. We let ν be the overriding union (i.e., $\oplus_{(i \in \{1,\ldots,n\}} \nu_i$) of the clock assignments ν_i of participants $\{1, \ldots, n\}$.

The visible transitions for a configuration $s = (T_1, \ldots, T_n, \tilde{w})$ is $(\nu, s) \xrightarrow{\ell} (\nu', s')$ where ν' and $s' = (T_1', \ldots, T_n', \tilde{w}')$ are defined as follows:

1. $(\nu_{\mathtt{p}}, T_{\mathtt{p}}) \xrightarrow{\mathtt{pq}!l\langle S\rangle} (\nu'_{\mathtt{p}}, T'_{\mathtt{p}})$ and (i) $T_{\mathtt{r}} = T'_{\mathtt{r}}$, $\nu_{\mathtt{r}} = \nu'_{\mathtt{r}}$ for all $\mathtt{r} \neq \mathtt{p}$ and (ii) $w'_{\mathtt{pq}} = w_{\mathtt{pq}} \cdot l\langle S\rangle$ and $w'_{\mathtt{rk}} = w_{\mathtt{rk}}$ for all $\mathtt{rk} \neq \mathtt{pq}$, or
2. $(\nu_{\mathtt{q}}, T_{\mathtt{q}}) \xrightarrow{\mathtt{pq}?l\langle S\rangle} (\nu'_{\mathtt{q}}, T'_{\mathtt{q}})$ and (i) $T_{\mathtt{r}} = T'_{\mathtt{r}}$, $\nu_{\mathtt{r}} = \nu'_{\mathtt{r}}$ for all $\mathtt{r} \neq \mathtt{q}$ and

(ii) $l\langle S\rangle \cdot w'_{\mathtt{pq}} = w_{\mathtt{pq}}$ and $w'_{\mathtt{rk}} = w_{\mathtt{rk}}$ for all $\mathtt{rk} \neq \mathtt{pq}$, or

3. $\forall \mathtt{p} \in \mathcal{P}.(\nu_{\mathtt{p}}, T_{\mathtt{p}}) \xrightarrow{t} (\nu_{\mathtt{p}} + t, T_{\mathtt{p}})$ and $w'_{\mathtt{pq}} = w_{\mathtt{pq}}$ for all $\mathtt{p} \neq \mathtt{q} \in \mathcal{P}$.

The runs of a configuration s_0 are defined similarly to the runs of a *TG* (Definition 5.2.4) but with states (ν_i, s_i) and with the transitions in Definition 5.3. The traces of a configuration $TR(s)$ is the set of all the traces $TR(r)$ such that r is a run for s.

Before presenting the theorem of projection preservation for soundness and completeness, we show the following lemma, which is similar to Lemma 3.3.6.

Lemma 5.3.5. Let G be a projectable, see Definition 3.3.3 timed choreographical global types, and ν be the overriding union of the assignments $\nu_{\mathtt{p}}$ of all $\mathtt{p} \in \mathcal{P}(G)$, then for all $G \upharpoonright \mathtt{p}$:

$$\nu + t \models^* \mathtt{ready}(G) \quad \text{iff} \quad \nu_{\mathtt{p}} + t \models^* \mathtt{ready}(G \upharpoonright \mathtt{p})$$

Theorem 5.3.6 (Soundness and completeness of projection)**.** *Let G be a timed global type, $\{T_1, \ldots, T_n\} = \{\mathbf{G} \upharpoonright \mathtt{p}\}_{\mathtt{p} \in \mathcal{P}(G)}$ be the set of its projections, and $(T_1, \ldots, T_n, \tilde{\varepsilon})$ the corresponding initial configuration. Then $TR(G) = TR((T_1, \ldots, T_n, \tilde{\varepsilon}))$.*

Proof. The proof of Theorem 5.3.6 proceeds as the proof of Theorem 3.1 [DY13] and proof of Theorem 3.3.7. And it is noted that $\mathtt{ready}(G)$ uses Definition 5.2.2.

The proof of Theorem 5.3.6 is similar to that of Theorem 3.3.7; they both involve three steps: subtyping of local types, extending projection and proof of soundness and completeness. The difference is that for this proof, (1) in the extension of projection, we add projection configuration for fork/join; (2) in proof of soundness and completeness, we add cases of fork/join transitions for global types.

First, as in [DY13], we define a behaviour-preserving subtyping relation $T < T'$ between timed choreographical local types to add branches in branching types but not selection types. This subtyping is extended to configurations as follows: $(\nu, \tilde{T}, \tilde{w}) < (\nu', \tilde{T}', \tilde{w}')$ if $\tilde{w} = \tilde{w}'$, $\nu = \nu'$ and $\forall \mathtt{p} \in \mathcal{P}$, $T_{\mathtt{p}} < T'_{\mathtt{p}}$ and we obtain the following property: if $(\nu, \tilde{T}, \tilde{w}) < (\nu', \tilde{T}', \tilde{w}')$ then $(\nu, \tilde{T}, \tilde{w}) \approx (\nu', \tilde{T}', \tilde{w}')$.

Second, we (1) extend the definition of projection to global intermediary steps and (2) define projected configuration of a global type G, written $[[G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq} \in \mathcal{P}(G)}$, which takes fork/join into account. The contents of the buffers

$[[G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}$ is defined as:

$$
\begin{aligned}
[[\mathbf{x} = \mathtt{p}' \rightsquigarrow \mathtt{q}' : l\langle U\rangle\{A\}; \mathbf{x}']]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= [[\mathbf{x}']]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}[w_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'} \cdot l\langle S\rangle] \\
[[\mathbf{x} = \mathtt{p}' \rightarrow \mathtt{q}' : l\langle U\rangle\{A\}; \mathbf{x}']]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= [[\mathbf{x}']]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} \\
[[\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= [[\mathbf{x}_1 \mid \mathbf{x}_2]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} \text{ if } \mathtt{p},\mathtt{q} \in \mathcal{P}(G) \\
[[\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= [[\mathbf{x}_i]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} \\
[[\mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= [[\mathbf{x}]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} \\
[[\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= [[\mathbf{x}]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} \\
[[\mathbf{x} = \mathsf{end}]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} &= \{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}
\end{aligned}
$$

Third, we prove the theorem by combining local type subtyping and projection extensions into the following step equivalence lemma: if $[[G]] < (\nu, \tilde{T}, \tilde{w})$ then $G \xrightarrow{\ell} G' \Leftrightarrow (\nu, \tilde{T}, \tilde{w}) \xrightarrow{\ell} (\nu', \tilde{T}', \tilde{w}')$ and $[[G']] < (\nu', \tilde{T}', \tilde{w}')$.

The step equivalence is proved by induction on the structure of the labelled transition of G. The proof of soundness is by induction rules of LTS of $(\nu, G) \xrightarrow{\ell} (\nu', G')$. The cases for labelled messages $\lfloor$Send$\rfloor$, $\lfloor$Recv$\rfloor$, choices and $\lfloor$Time$\rfloor$ are identical to those in Section 3.3. Here we prove the cases of fork, join and merge. For the fork case, where $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2 \rightarrow \mathbf{x}_1 \mid \mathbf{x}_2$, the projected configuration is $[[\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} = [[\mathbf{x}_i]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}$ with local type T as $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2 \upharpoonright \mathtt{p} = \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$. By fork transition rule for local type, we have $(\nu, T, \tilde{w}) \rightarrow (\nu, T', \tilde{w})$, where T' is the projected local type of $\mathbf{x}_1 \mid \mathbf{x}_2$. The cases for join and merge is similar to that of fork because they are only transitions of syntax expressions and do not involve message exchanges, queue changes and time changes.

Completeness proceeds similarly to [DY13] and the step $\lfloor$Time$\rfloor$ which also follows by Lemma 5.3.5. □

5.4. Timed Choreographical Processes

In this section, we present processes for timed choreographical session types and the operational semantics to allow processes to reduce. The types systems are defined for verifying the processes. We also extend to runtime processes, which are used in proofs in Section 5.5.

5.4.1. Syntax of Process Calculus

The processes are modelled using a timed extension of the asynchronous choreographical session calculus in Section 4.4.

Definition 5.4.1 (Syntax of Process and Network)**.** The syntax of timed choreographical session calculus is defined below. For the purpose of simple extension, we add $\texttt{delay}(t).\mathbf{x}(\tilde{x})$ in the syntax definition and remaining expressions are as same as choreographical session processes.

$v ::= a \mid s[\mathtt{p}] \mid \texttt{true} \mid \texttt{false} \mid \ldots$	values
$e ::= v \mid x \mid e \wedge e \mid \ldots$	expression
$h ::= \emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)$	messages
$P ::= \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}$	definition
$P ::=$	process transition
$\mid \mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e})$	init
$\mid \mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e})$	request
$\mid \mathbf{x}(\tilde{x}) = x!\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e})$	send
$\mid \mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e})$	receive
$\mid \mathbf{x}(\tilde{x}) = \texttt{delay}(t).\mathbf{x}'(\tilde{x})$	delay
$\mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z})$	parallel
$\mid \mathbf{x}(\tilde{x}) = \texttt{if}\ e\ \texttt{then}\ \mathbf{x}'(\tilde{e}')\ \texttt{else}\ \mathbf{x}''(\tilde{e}'')$	conditional
$\mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x})$	external choice
$\mid \mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x})$	join
$\mid \mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x})$	merge
$\mid \mathbf{x}(\tilde{x}) = (\nu a)\ \mathbf{x}'(a\tilde{x})$	new name
$\mid \mathbf{x}(\tilde{x}) = \mathbf{0}$	null

$$
\begin{array}{lll}
\mathbf{X} ::= & & \text{state} \\
\quad | & \mathbf{x}(\tilde{v}) & \text{thread} \\
\quad | & \mathbf{X} \mid \mathbf{X} & \text{parallel} \\
\quad | & (\nu a)\mathbf{X} & \text{restriction} \\
\quad | & \mathbf{0} & \text{null} \\
\mathbf{N} ::= & & \text{network} \\
\quad | & P & \text{def} \\
\quad | & \mathbf{N} \,||\, \mathbf{N} & \text{parallel} \\
\quad | & (\nu a)\mathbf{N} & \text{new name} \\
\quad | & \mathbf{0} & \text{null} \\
\quad | & (\nu s)\mathbf{N} & \text{new session} \\
\quad | & s : h & \text{queue} \\
\quad | & a\langle s \rangle[\mathtt{p}] & \text{invitation}
\end{array}
$$

The operational semantics describe the processes transition and they are used in subject reductions. Compared with operational semantics rules for choreographical session processes, we simply add [T-DELAY] to present the passage of time. The other rules remain the same, shown in Figure 5.6.

5.4.2. Typing Timed Choreographical Processes

As to typing rules, we use the same judgement to express the relation between processes and the corresponding local types. The main difference is that we add clock assignment in the typing system so that we can track the time passing in the system. Then the session environment is defined as: $\Delta ::= \emptyset \mid \Delta, c : (\nu, \mathcal{T})$ where a channel is accompanied by the pair of clock assignment and local types. The typing systems for all rules are shown in Figure 5.7 and Figure 5.8.

Rule [T-INIT] shows the session initialisation, and Rule [T-REQ] is similar except we record the session and projected session type $\mathcal{T} = \mathbf{G} \upharpoonright \mathtt{p}$. It is noted that there may be delay between [T-INIT] and [T-REQ], but the actions of [T-INIT] and [T-REQ] cost no time, that is, clock assignment remains unchanged.

Rule [T-SEND] records the send type for $z_i(T_i = \mathbf{x} =! \langle \mathtt{p}, l\langle U \rangle \rangle . \mathbf{x}')$ and $\mathbf{x} = \mathbf{x}'$ for all other sessions, and requires that the clock assignment ν satisfies the send constraint δ_o, and resets the clock. The rule [T-RECV] is symmetric and similar changes are applied to rule [T-DELEG] and rule [T-CATCH] for session delegation.

Rule [T-DELAY] shows the passage of time t in clock assignment ν and recorded

$$\frac{x[\tilde{v}/\tilde{x}] = a \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{a\langle \mathbf{G} \rangle} \mathbf{x}'(\tilde{v}')} \quad \text{[T-INIT]}$$

$$\frac{x[\tilde{v}/\tilde{x}] = a \qquad \tilde{e}[\tilde{v}/\tilde{x}][s/y] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{a\langle s \rangle[\mathtt{p}]} \mathbf{x}'(\tilde{v}')} \quad \text{[T-ACC]}$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad e[\tilde{v}/\tilde{x}] \downarrow v \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x\,!\,\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v \rangle} \mathbf{x}'(\tilde{v}')} \quad \text{[T-SEND]}$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad \tilde{e}[\tilde{v}/\tilde{x}][v/y] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v' \rangle} \mathbf{x}'(\tilde{v}')} \quad \text{[T-RCV]}$$

$$\frac{a \notin \tilde{v}}{\mathbf{x}(\tilde{x}) = (\nu a)\mathbf{x}'(a\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} (\nu a)\mathbf{x}'(a\tilde{v})} \quad \text{[T-NEW]}$$

$$\frac{e[\tilde{v}/\tilde{x}] \downarrow \mathsf{true} \qquad \tilde{e}'[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'') \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} \mathbf{x}'(\tilde{v}')} \quad \text{[T-IFT]}$$

$$\frac{}{\mathbf{x}(\tilde{x}) = \mathtt{delay}(t).\mathbf{x}'(\tilde{x}) \xrightarrow{t} \mathbf{x}'(\tilde{x})} \quad \text{[T-DELAY]}$$

$$\frac{\widetilde{P}, \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}'(\tilde{v}) \xrightarrow{\alpha} \mathbf{X}}{\widetilde{P}, \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\alpha} \mathbf{X}} \quad \text{[T-EXT]}$$

$$\frac{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}{\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \xrightarrow{\alpha} \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}'} \quad \text{[T-DEF]} \qquad \frac{P \xrightarrow{\tau} P'}{P \to P'} \quad \text{[T-TAU]}$$

$$\frac{P \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v \rangle} P'}{P \mid\mid s : h \to P' \mid\mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)} \quad \text{[T-PUT]}$$

$$\frac{P \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v \rangle} P'}{P \mid\mid s : (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \cdot h \to P' \mid\mid s : h} \quad \text{[T-GET]}$$

$$\frac{P \xrightarrow{a\langle \mathbf{G} \rangle} P' \quad \mathtt{p}_0, \ldots, \mathtt{p}_k \in \mathbf{G} \quad s \notin \mathsf{fn}(P')}{P \to (\nu s)(P' \mid\mid s : \varepsilon \mid\mid a\langle s \rangle[\mathtt{p}_0] \mid\mid \ldots \mid\mid a\langle s \rangle[\mathtt{p}_k])} \quad \text{[T-INIT}_N\text{]}$$

$$\frac{e[\tilde{v}/\tilde{x}] \downarrow \mathsf{false} \qquad \tilde{e}''[\tilde{v}/\tilde{x}] \downarrow \tilde{v}''}{\mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{x})\ \mathtt{else}\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} \mathbf{x}''(\tilde{v}'')} \quad \text{[T-IFF]}$$

Figure 5.6.: Operational Semantics of Timed Choreographical Session Calculus

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \tilde{y}:\tilde{U}\vdash x:\langle\mathbf{G}\rangle \quad \forall i,\mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=\mathbf{x}'}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=x\langle\mathbf{G}\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':(\nu,\tilde{U}'\,\widetilde{\mathcal{T}'})} \quad \text{[T-INIT]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \tilde{y}:\tilde{U}\vdash x:\langle\mathbf{G}\rangle \quad \forall i,\mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=\mathbf{x}' \quad \mathcal{T}=\mathbf{G}\upharpoonright\mathtt{p}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y)\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':(\nu,\tilde{U}'\,\widetilde{\mathcal{T}'}\mathcal{T})} \quad \text{[T-REQ]}$$

$$\frac{\begin{array}{c}\tilde{y}:\tilde{U}\vdash e:U \quad \tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \nu\models\delta_o\\ \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=!\langle\mathtt{p}',l\langle U\rangle\{B_o\}\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'\end{array}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=z_i\,!\,\langle\mathtt{p},l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':([\lambda_o\mapsto 0]\nu,\tilde{U}'\,\widetilde{\mathcal{T}'})} \quad \text{[T-SEND]}$$

$$\frac{\begin{array}{c}\tilde{y}:\tilde{U},y:U\vdash\tilde{e}:\tilde{U}' \quad \nu\models\delta_i\\ \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=?\langle\mathtt{p},l\langle U\rangle\{B_i\}\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'\end{array}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=z_i\,?\,\langle\mathtt{p},l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':([\lambda_i\mapsto 0]\nu,\tilde{U}'\,\widetilde{\mathcal{T}'})} \quad \text{[T-RECV]}$$

$$\frac{\begin{array}{c}\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \nu\models\delta_o\\ \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=!\langle\mathtt{p}',l\langle\mathcal{T}\rangle\{B_o\}\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'\end{array}}{\vdash\mathbf{x}(\tilde{y}\tilde{z}y)=z_i\,!\,\langle\mathtt{p},l\langle y\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}\mathcal{T}) \;[\!]\; \mathbf{x}':([\lambda_o\mapsto 0]\nu,\tilde{U}'\,\widetilde{\mathcal{T}'})} \quad \text{[T-DELEG]}$$

$$\frac{\begin{array}{c}\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \nu\models\delta_i\\ \mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=?\langle\mathtt{p},l\langle\mathcal{T}\rangle\{B_i\}\rangle.\mathbf{x}' \quad \forall j\neq i,\mathcal{T}_j=\mathcal{T}_j'\uplus\mathbf{x}=\mathbf{x}'\end{array}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=z_i\,?\,\langle\mathtt{p},l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z}y)\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':([\lambda_i\mapsto 0]\nu,\tilde{U}'\,\widetilde{\mathcal{T}'}\mathcal{T})} \quad \text{[T-CATCH]}$$

$$\frac{\mathcal{T}_i=\mathcal{T}_i'\uplus\mathbf{x}=\mathbf{x}}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=\mathtt{delay}(t).\mathbf{x}'(\tilde{y}\tilde{z})\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':((\nu+t),\tilde{U}\,\widetilde{\mathcal{T}'})} \quad \text{[T-DELAY]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{y}_1:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{y}_2:\tilde{U}_2 \quad \forall i,\mathcal{T}_i=(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})\uplus\mathbf{x}=\mathbf{x}_1\mid\mathbf{x}_2}{\vdash\mathbf{x}(\tilde{y}\tilde{z})=\mathbf{x}_1(\tilde{y}_1\tilde{z})\mid\mathbf{x}_2(\tilde{y}_2\tilde{z})\triangleright\mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}_1:(\nu,\tilde{U}_1\,\widetilde{\mathcal{T}_1}),\mathbf{x}_2:(\nu,\tilde{U}_2\,\widetilde{\mathcal{T}_2})} \quad \text{[T-PAR]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{y}_1:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{y}_2:\tilde{U}_2 \quad \forall i,(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})=\mathcal{T}_i'\uplus\mathbf{x}_1\mid\mathbf{x}_2=\mathbf{x}}{\vdash\mathbf{x}_1(\tilde{y}_1\tilde{z})\mid\mathbf{x}_2(\tilde{y}_2\tilde{z})=\mathbf{x}(\tilde{y}\tilde{z})\triangleright\;\mathbf{x}_1:(\nu,\tilde{U}_1\,\widetilde{\mathcal{T}_1}),\mathbf{x}_2:(\nu,\tilde{U}_2\,\widetilde{\mathcal{T}_2}) \;[\!]\; \mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}'})} \quad \text{[T-JOIN]}$$

Figure 5.7.: Typing System for Timed Choreographical Processes (Part 1)

$$\frac{\tilde{y}:\tilde{U}\vdash e:U \quad \tilde{y}:\tilde{U}\vdash \tilde{e}_1:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash \tilde{e}_2:\tilde{U}_2 \quad \forall i,\mathcal{T}_i=(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})\uplus \mathbf{x}=\mathbf{x}_1\oplus\mathbf{x}_2}{\begin{array}{c}\vdash \mathbf{x}(\tilde{y}\tilde{z}) = \texttt{if } e \texttt{ then } \mathbf{x}_1(\tilde{e}_1\tilde{z}) \texttt{ else } \mathbf{x}_2(\tilde{e}_2\tilde{z}) \\ \triangleright \mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}_1:(\nu,\tilde{U}_1\,\widetilde{\mathcal{T}_1}),\mathbf{x}_2:(\nu,\tilde{U}_2\,\widetilde{\mathcal{T}_2})\end{array}} \quad \text{[T-IF]}$$

$$\frac{\forall i,\mathcal{T}_i=(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})\uplus \mathbf{x}=\mathbf{x}_1 \,\&\, \mathbf{x}_2}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=\mathbf{x}_1(\tilde{y}\tilde{z})\,\&\,\mathbf{x}_2(\tilde{y}\tilde{z})\triangleright \mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}_1:(\nu,\tilde{U}\,\widetilde{\mathcal{T}_1}),\mathbf{x}_2:(\nu,\tilde{U}\,\widetilde{\mathcal{T}_2})} \quad \text{[T-CHOICE]}$$

$$\frac{\forall i,(\mathcal{T}_{1i}\cup\mathcal{T}_{2i})=\mathcal{T}_i'\uplus \mathbf{x}_1+\mathbf{x}_2=\mathbf{x}}{\vdash \mathbf{x}_1(\tilde{y}\tilde{z})+\mathbf{x}_2(\tilde{y}\tilde{z})=\mathbf{x}(\tilde{y}\tilde{z})\triangleright \;\mathbf{x}_1:(\nu,\tilde{U}\,\widetilde{\mathcal{T}_1}),\mathbf{x}_2:(\nu,\tilde{U}\,\widetilde{\mathcal{T}_2}) \;[\!]\; \mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}'})} \quad \text{[T-MERGE]}$$

$$\frac{\vdash P_i\triangleright\Sigma_i \;[\!]\; \Sigma_i' \quad comp(\{\Sigma_i \;[\!]\; \Sigma_i'\}_i) \quad \mathbf{x}_0:\tilde{U}=(\cup\Sigma_i)\setminus(\cup\Sigma_i') \quad \Gamma\vdash\tilde{v}:\tilde{U}}{\Gamma\vdash \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}_0(\tilde{v})} \quad \text{[T-DEF]}$$

$$\frac{\forall i,\widetilde{\mathcal{T}_i'}=\widetilde{\mathcal{T}_i}\uplus \mathbf{x}=\mathbf{x}' \qquad \widetilde{\mathcal{T}_i}=\widetilde{\mathcal{T}_i'}\uplus \mathbf{x}=\mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=(\nu a)\mathbf{x}'(a\tilde{y}\tilde{z})\triangleright \mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}':(\nu,\langle\mathbf{G}\rangle\tilde{U}\,\widetilde{\mathcal{T}'})} \quad \text{[T-RES]}$$

$$\frac{\forall i,\mathcal{T}_i=\mathsf{def}\ \mathbf{x}=\mathsf{end}\ \mathsf{in}\ \mathbf{x}}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=\mathbf{0}\triangleright \mathbf{x}:(\nu,\tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]} \quad \text{[T-NIL]}$$

$$\frac{\Gamma\vdash \mathbf{N}_1 \quad \Gamma\vdash \mathbf{N}_2}{\Gamma\vdash \mathbf{N}_1 \,||\, \mathbf{N}_2}\text{[T-NPAR]} \qquad \frac{\Gamma,a:\langle\mathbf{G}\rangle\vdash \mathbf{N}}{\Gamma\vdash(\nu a)\mathbf{N}}\text{[T-NRES]} \qquad \frac{}{\Gamma\vdash \mathbf{0}} \quad \text{[T-NNIL]}$$

Figure 5.8.: Typing System for Timed Choreographical Processes (Part 2) and Network

$$
\begin{array}{rcl}
\mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z}) & \vdash & \mathbf{x}(\tilde{v}) \equiv \mathbf{x}'(\tilde{y}[\tilde{v}/\tilde{x}]) \mid \mathbf{x}''(\tilde{z}[\tilde{v}/\tilde{x}]) \\
\mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x}) & \vdash & \mathbf{x}(\tilde{y}[\tilde{v}/\tilde{x}]) \mid \mathbf{x}'(\tilde{z}[\tilde{v}/\tilde{x}]) \equiv \mathbf{x}''(\tilde{v}) \\
\mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) & \vdash & \mathbf{x}(\tilde{v}) \equiv \mathbf{x}''(\tilde{v}) \\
\mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) & \vdash & \mathbf{x}'(\tilde{v}) \equiv \mathbf{x}''(\tilde{v}) \\
\mathbf{x}(\tilde{x}) = \mathbf{0} & \vdash & \mathbf{x}(\tilde{v}) \equiv \mathbf{0} \\
\texttt{delay}(t+t').\mathbf{x}(\tilde{x}) & \equiv & \texttt{delay}(t).\texttt{delay}(t').\mathbf{x}(\tilde{x}) \\
\texttt{delay}(0).\mathbf{x}(\tilde{x}) & \equiv & \mathbf{x}(\tilde{x}) \\
\texttt{delay}(t).(\nu a)\mathbf{x}(\tilde{x}) & \equiv & (\nu a)\texttt{delay}(t).\mathbf{x}(\tilde{x}) \\
\texttt{delay}(t).(\mathbf{x}(\tilde{x}) \mid \mathbf{x}'(\tilde{x}')) & \equiv & \texttt{delay}(t).\mathbf{x}(\tilde{x}) \mid \texttt{delay}(t).\mathbf{x}'(\tilde{x}')
\end{array}
$$

Figure 5.9.: Selected Structural Congruence Rules

types remain unchanged. Rule [T-PAR] describes the parallel composition of two concurrent branches, denoted as $(\mathcal{T}_1 \cup \mathcal{T}_2) \uplus \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$, and both two concurrent branches start with the same clock assignment ν. Rule [T-JOIN] is a symmetric rule and at the join point all clocks should synchronise as ν. We apply similar changes to external/internal choice rules [T-IF] and [T-CHOICE] and merge rule [T-MERGE]. Rule [T-RES] records the binding and time remains the same. Rule [T-NIL] shows the termination of a process and the following time information is ignored.

Other rules for state and networks are similar to those in choreographical session types in Section 4.4.2.

The structural congruence rules for parallel/join and delay transitions are shown in Figure 5.9. The structural congruence rules for state and network are as same as in Section 4.4.2.

5.4.3. Typing Run-Time Processes

In this subsection we extend the typing system of programs to type run-time processes, which is necessary for subject reduction.

The definition for message types $\mathtt{T}$ is same as in Section 4.4.3. We define the choreographical session types which have both message and completed types.

$$
\begin{array}{rrcll}
\text{Choreographical} & \mathsf{T} & ::= & T & \textit{session} \\
 & & \mid & \mathtt{T} & \textit{message} \\
 & & \mid & \mathtt{T}; T & \textit{continuation}
\end{array}
$$

Then ; is defined by:

$$\Delta;\{s[\mathtt{q}:r]:\mathtt{T}\} = \begin{cases} \Delta', s[\mathtt{q}:r]:\mathtt{T}';\mathtt{T} & \text{if } \Delta = \Delta', s[\mathtt{q}:r]:\mathtt{T}', \\ \Delta, s[\mathtt{q}:r]:\mathtt{T} & \text{otherwise.} \end{cases}$$

The difference from choreographical session types is that we define the specifications as follows, which clock assignment is associated with message types.

$$\Sigma \quad ::= \quad \mathtt{T} \mid (\nu, \mathcal{T})$$

The equivalence relation $\equiv$ for generalised specifications is as follows:

$!\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle;!\langle \mathtt{p}_2, l_2\langle U_2\rangle\rangle;\mathtt{T} \equiv !\langle \mathtt{p}_2, l_2\langle U_2\rangle\rangle;!\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle;\mathtt{T} \quad \text{if } \mathtt{p}_1 \neq \mathtt{p}_2'$
$\mathtt{T} \equiv \mathtt{T}'$ implies $\mathtt{T};\mathcal{T} \equiv \mathtt{T}';\mathcal{T}$
$\mathcal{T} \equiv \mathcal{T}'$ implies $(\nu, \mathcal{T}) \equiv (\nu, \mathcal{T}')$

The choreographical proof rules rely on the following definitions:

- $\Delta \equiv \Delta'$ if $c : \Sigma \in \Delta$ and $\Sigma \neq (\nu, \mathtt{end})$ imply $c : \Sigma' \in \Delta'$ with $\Sigma \equiv \Sigma'$ and vice versa.
- $\Delta \mid \Delta' = \Delta \setminus dom(\Delta') \cup \Delta' \setminus dom(\Delta) \cup \{y : \Sigma \mid \Sigma' \mid y : \Sigma \in \Delta \wedge y : \Sigma' \in \Delta'\}$
- $\Sigma \mid \Sigma' = \begin{cases} \Sigma;\Sigma' & \text{if } \Sigma \text{ is a message type} \\ \Sigma';\Sigma & \text{if } \Sigma' \text{ is a message type} \\ \text{undefined} & \text{otherwise} \end{cases}$
- $(\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1) \mid (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_2) = \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ (\mathbf{x}_1 \mid \mathbf{x}_2)$
  $!\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle;\cdots!\langle \mathtt{p}_n, l_1\langle U_n\rangle\rangle \mid (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}) = \mathsf{def}\ \widetilde{T}'\ \mathsf{in}\ \mathbf{x}$
  with $\widetilde{T}' = \mathbf{x} = !\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle.\mathbf{x}_1, \cdots \mathbf{x}_n = !\langle \mathtt{p}_2, l_2\langle U_2\rangle\rangle.\mathbf{x}_{n+1}, \widetilde{T}[\mathbf{x}_{n+1}/\mathbf{x}]$

By adding clock assignment to environment specification, the typing rule [T-STATE] for run-time state processes is defined as

$$\frac{\vdash P_i \triangleright \Sigma_i \ [\!]\ \Sigma_i' \quad comp(\{\Sigma_i;\Sigma_i'\}_i) \quad \mathbf{x} : (\nu, \tilde{U}\tilde{\mathcal{T}}) \in \Sigma_i \quad \Gamma \vdash \tilde{v} : \tilde{U}}{\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : (\nu, \tilde{\mathcal{T}})}\ \text{[T-STATE]}$$

The rules for [PAR], [RES], [NIL] and [RDEF] are as same as those for choreographical run-time processes.

The typing system for run-time state processes and typing system for queues are same as in Section 4.4.2.

$$\frac{(\nu,\mathcal{T}) \xrightarrow{\mathtt{pq}!l\langle U\rangle\{\delta_o,\lambda_o\}} (\nu',\mathcal{T}') \qquad \Gamma \vdash v : U}{(\Gamma,\Delta,s[\mathtt{p}] : (\nu,\mathcal{T})) \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v\rangle} (\Gamma,\Delta,s[\mathtt{p}] : (\nu',\mathcal{T}'))} \quad [\textsc{Send}]$$

$$\frac{(\nu,\mathcal{T}) \xrightarrow{\mathtt{qp}?l\langle U\rangle\{\delta_i,\lambda_i\}} (\nu',\mathcal{T}') \qquad \nu \models \delta_i \quad \nu' = [\lambda_i \mapsto 0]\nu \qquad \Gamma \vdash v : U}{(\Gamma,\Delta,s[\mathtt{p}] : (\nu,\mathcal{T})) \xrightarrow{s[\mathtt{q},\mathtt{p}]?l\langle v\rangle} (\Gamma,\Delta,s[\mathtt{p}] : (\nu',\mathcal{T}'))} \quad [\textsc{Recv}]$$

$$\frac{(\nu,\mathcal{T}) \xrightarrow{\mathtt{pq}!l\langle \mathcal{T}_s\rangle\{\delta_o,\lambda_o\}} (\nu',\mathcal{T}') \qquad \nu \models \delta_o \quad \nu' = [\lambda_o \mapsto 0]\nu}{(\Gamma,\Delta,s[\mathtt{p}] : (\nu,\mathcal{T}),s'[\mathtt{p}'] : (\nu,\mathcal{T}_s)) \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle s'[\mathtt{p}']\rangle} (\Gamma,\Delta,s[\mathtt{p}] : (\nu',\mathcal{T}'))} \quad [\textsc{Deleg}]$$

$$\frac{(\nu,\mathcal{T}) \xrightarrow{\mathtt{pq}?l\langle \mathcal{T}_r\rangle\{\delta_i,\lambda_i\}} (\nu',\mathcal{T}') \qquad \nu \models \delta_i \quad \nu' = [\lambda_i \mapsto 0]\nu}{(\Gamma,\Delta,s[\mathtt{p}] : (\nu,\mathcal{T})) \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle s'[\mathtt{p}']\rangle} (\Gamma,\Delta,s[\mathtt{p}] : \mathcal{T}',s'[\mathtt{p}'] : (\nu',\mathcal{T}_r))} \quad [\textsc{Catch}]$$

$$\frac{\nu + t \models \mathtt{ready}(\mathcal{T})}{(s[\mathtt{p}] : (\nu,\mathcal{T})) \xrightarrow{t} (s[\mathtt{p}] : (\nu + t,\mathcal{T}))} \quad [\textsc{Time}]$$

$$\frac{(\nu,\mathcal{T}_1) \xrightarrow{\mathtt{pq}!l\langle U\rangle\{\delta_o,\lambda_o\}} (\nu',\mathcal{T}_1') \qquad (\nu',\mathcal{T}_2) \xrightarrow{\mathtt{pq}?l\langle U\rangle\{\delta_i,\lambda_i\}} (\nu'',\mathcal{T}_2')}{(s[\mathtt{p}] : (\nu,\mathcal{T}_1),s[\mathtt{q}] : (\nu,\mathcal{T}_2),\Delta) \rightarrow (s[\mathtt{p}] : (\nu'',\mathcal{T}_1'),s[\mathtt{q}] : (\nu'',\mathcal{T}_2'),\Delta)} \quad [\textsc{Com}]$$

Figure 5.10.: Reduction Rules for Session Environment

Since we add clock assignment to environment definition, then the reduction rules for session environment also adjust the same changes and are defined in Figure 5.10

5.5. Safety of Timed Choreographical Multiparty Session Types

We define properties that ensure the soundness and safety of timed choreographical session types: subjection reduction and time error-freedom.

Lemma 5.5.1 (Subject Congruence)**.** *Suppose* $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}'$*. Then* $\Gamma, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta$*. Similarly for P and* $\mathbf{N}$*.*

Proof.

1. (Process P) With structural congruence in Section 5.4, the proofs are are straightforward by induction.
2. (Def agents P) The same as Section 4.5

3. (Network **N**) The same as [CDCYP15].

□

Theorem 5.5.2 (Subject Transition and Reduction)**.**

1. $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$ *imply* $\Gamma', \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
2. $\Gamma \vdash P \triangleright \Delta$ *and* $P \xrightarrow{\alpha} P'$ *imply* $\Gamma' \vdash P' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
3. $\Gamma \vdash_{\mathcal{S}} \mathbf{N} \triangleright \Delta$ *and* $\mathbf{N} \longrightarrow \mathbf{N}'$ *imply* $\Gamma \vdash_{\mathcal{S}} \mathbf{N}' \triangleright \Delta'$ *with* $\Delta \longrightarrow^* \Delta'$.

Proof. **(Process *P*)**

Case [T-SEND] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = x!\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v \rangle} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, $e[\tilde{v}/\tilde{y}] \downarrow v$ and $\tilde{e}[\tilde{v}/\tilde{y}] \downarrow \tilde{w}$ by [T-SEND]. Then by [T-STATE],

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : (\nu, \widetilde{\mathcal{T}})$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = x!\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;\; [\!] \;\; \mathbf{x}' : ([\lambda_i \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}}')$$

with

$$\tilde{y} : \tilde{U} \vdash e : U, \quad \tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}',$$

and

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = !\langle \mathtt{p}', l\langle U \rangle \{B_o\} \rangle.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U} \vdash e[\tilde{v}/\tilde{y}] : U, e_i[\tilde{v}/\tilde{y}] : U_i$$

On the other hand, by $\lfloor$T-SEND$\rfloor$ in 5.3, we have

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = !\langle \mathtt{p}', l\langle U \rangle \{B_o\} \rangle.\mathbf{x}' \xrightarrow{!\langle \mathtt{p}, l\langle U \rangle \rangle} \mathcal{T}_i'$$

This implies, by [SEND] in Figure 5.10, with $\Gamma \vdash v : U$ and $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, we have:

$$(\Gamma, \Delta) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v\rangle} (\Gamma, \Delta')$$

where $\Delta = \tilde{c} : (\nu, \tilde{\mathcal{T}})$ and $\Delta' = \Delta \setminus s[\mathtt{q}] \cup \{s[\mathtt{q}] : \mathcal{T}_i'\}$ as desired.

Case [T-RECV] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = x?\langle \mathtt{p}, l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{s[\mathtt{q},\mathtt{p}]?l\langle v\rangle} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$ and $\tilde{e}[\tilde{v}/\tilde{x}][v/y] \downarrow \tilde{w}$ by [RECV]. Then by [T-STATE],

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : (\nu, \widetilde{\mathcal{T}})$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = x?\langle \mathtt{p}, l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;\; [] \;\; \mathbf{x}' : ([\lambda_i \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}}')$$

with

$$\tilde{y} : \tilde{U}, y : U \vdash \tilde{e} : \tilde{U}'$$

and

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = ?\langle \mathtt{p}', l\langle U\rangle\{B_i\}\rangle.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U}, v : U \vdash e[\tilde{v}/\tilde{y}] : U, e_i[\tilde{v}/\tilde{y}] : U_i$$

On the other hand, by $\lfloor$T-RECV$\rfloor$ in Section 5.3, we have

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = ?\langle \mathtt{p}', l\langle U\rangle\{B_i\}\rangle.\mathbf{x}' \quad \xrightarrow{?\langle \mathtt{p}, l\langle U\rangle\rangle} \quad \mathcal{T}_i'$$

This implies, by [RECV] in Figure 5.10, with $\Gamma \vdash v : U$ and $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, we have:

$$(\Gamma, \Delta) \xrightarrow{s[\mathtt{q},\mathtt{p}]?l\langle v\rangle} (\Gamma, \Delta')$$

where $\Delta = \tilde{c} : (\nu, \tilde{\mathcal{T}})$ and $\Delta' = \Delta \setminus s[\mathtt{q}] \cup \{s[\mathtt{q}] : \mathcal{T}_i'\}$ as desired.

Case [T-INIT]

Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = x\langle\mathbf{G}\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{w}\tilde{c}) \quad \xrightarrow{a\langle\mathbf{G}\rangle} \quad \mathbf{x}'(\tilde{v}\tilde{c})$$

with $a = x[\tilde{w}/\tilde{y}]$, $\tilde{e}[\tilde{w}/\tilde{y}] \downarrow \tilde{v}$, By [T-INIT],

$$\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x\langle\mathbf{G}\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \rhd \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;\; [\!] \;\; \mathbf{x}' : (\nu, \tilde{U}'\,\widetilde{\mathcal{T}}')$$

with $\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}'$, $\tilde{y} : \tilde{U} \vdash x : \langle\mathbf{G}\rangle$ and $\forall i, \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}'$. Then by [T-STATE], we have:

$$\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{w}\tilde{c}) \rhd \tilde{c} : (\nu, \tilde{\mathcal{T}})$$

and, by [INIT] in Figure 4.17, we have

$$(\Gamma, \Delta) \quad \xrightarrow{a\langle\mathbf{G}\rangle} \quad (\Gamma, \Delta)$$

By [T-STATE], we have

$$\Gamma, \widetilde{P} \vdash \mathbf{x}'(\tilde{v}\tilde{c}) \rhd \tilde{c} : (\nu, \tilde{\mathcal{T}})$$

as desired, noting $\mathcal{T}_i' \equiv \mathcal{T}_i$.

Case [T-ACCEPT]

Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \vdash \mathbf{x}(\tilde{w}\tilde{c}) \quad \xrightarrow{a[\mathtt{p}]\langle s\rangle} \quad \mathbf{x}'(\tilde{v}\tilde{c}s)$$

with $a = x[\tilde{w}/\tilde{y}]$, $\tilde{e}[\tilde{w}/\tilde{y}] \downarrow \tilde{v}$, By [T-REQ],

$$\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \rhd \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;\; [\!] \;\; \mathbf{x}' : (\nu, \tilde{U}'\,\widetilde{\mathcal{T}}'\mathcal{T})$$

with $\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}'$, $\tilde{y} : \tilde{U} \vdash x : \langle\mathbf{G}\rangle$, $\forall i, \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}'$ and $\mathcal{T} = \mathbf{G} \upharpoonright \mathtt{p}$. Then by [T-STATE], we have:

$$\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{w}\tilde{c}) \rhd \tilde{c} : (\nu, \tilde{\mathcal{T}})$$

and, by [ACC] in Figure 4.17, we have

$$(\Gamma, \Delta) \quad \xrightarrow{a[\mathtt{p}]\langle s\rangle} \quad (\Gamma, \Delta, s[\mathtt{p}] : \mathcal{T})$$

By [T-STATE], we have

$$\Gamma, \widetilde{P} \vdash \mathbf{x}'(\tilde{v}\tilde{c}s) \rhd \tilde{c} : (\nu, \tilde{\mathcal{T}}, s[\mathtt{p}] : \mathcal{T})$$

as desired, noting $\mathcal{T}_i' \equiv \mathcal{T}_i$.

Case [T-IFT] Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = \texttt{if } e \texttt{ then } \mathbf{x}_1(\tilde{e}_1\tilde{z}) \texttt{ else } \mathbf{x}_2(\tilde{e}_2\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{\tau} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $e[\tilde{v}/\tilde{y}] \downarrow \mathsf{true}$ and $\tilde{e}_1[\tilde{v}/\tilde{y}] \downarrow \tilde{w}$ by [IFT]. Then by [T-STATE] in,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : (\nu, \widetilde{\mathcal{T}})$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = \texttt{if } e \texttt{ then } \mathbf{x}_1(\tilde{e}_1\tilde{z}) \texttt{ else } \mathbf{x}_2(\tilde{e}_2\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\,\widetilde{\mathcal{T}} \;\; [\!] \;\; \mathbf{x}_1 : (\nu, \tilde{U}_1\,\widetilde{\mathcal{T}_1}), \mathbf{x}_2 : (\nu, \tilde{U}_2\,\widetilde{\mathcal{T}_2})$$

with

$$\tilde{y} : \tilde{U} \vdash e : U \quad \tilde{y} : \tilde{U} \vdash \tilde{e}_1 : \tilde{U}_1 \quad \tilde{y} : \tilde{U} \vdash \tilde{e}_2 : \tilde{U}_2$$

and

$$\forall i, \mathcal{T}_i = (\mathcal{T}_{1i} \cup \mathcal{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U} \vdash e_1[\tilde{v}/\tilde{y}] : U_1$$

On the other hand, by $\lfloor$COND$\rfloor$ in Figure 4.18, we have

$$\mathcal{T}_i = (\mathcal{T}_{1i} \cup \mathcal{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2 \quad \xrightarrow{\tau} \quad \mathcal{T}_i'$$

This implies, by [TAU] in Figure 4.17, we have:

$$(\Gamma, \Delta) \quad \xrightarrow{\tau} \quad (\Gamma, \Delta)$$

where $\Delta = \tilde{c} : (\nu, \tilde{\mathcal{T}})$.

Case [T-NEW] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = (\nu a)\mathbf{x}'(a\tilde{y}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{\tau} (\nu a)\mathbf{x}'(a\tilde{v}\tilde{c})$$

with $a \notin \tilde{v}$ by [T-NEW]. Then by [T-STATE] ,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : (\nu, \widetilde{\mathcal{T}})$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = (\nu a)\mathbf{x}'(a\tilde{y}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;\; [\!] \;\; \mathbf{x}' : (\nu, \langle \mathbf{G} \rangle \tilde{U}\,\widetilde{\mathcal{T}}')$$

with

$$\forall i, \widetilde{\mathcal{T}}_i' = \widetilde{\mathcal{T}}_i \uplus \mathbf{x} = \mathbf{x}'$$

and $z_i = x$. On the other hand, by $\lfloor$EQ$\rfloor$ in Figure 4.18, we have

$$\widetilde{\mathcal{T}}_i' = \widetilde{\mathcal{T}}_i \uplus \mathbf{x} = \mathbf{x}' \quad \xrightarrow{\tau} \quad \mathcal{T}_i'$$

This implies, by [TAU] in Figure 4.17, we have:

$$(\Gamma, \Delta) \quad \xrightarrow{\tau} \quad (\Gamma, \Delta)$$

where $\Delta = \tilde{c} : (\nu, \tilde{T})$.

Case [T-EXT] Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{x}_1(\tilde{y}\tilde{z}) \;\&\; \mathbf{x}_2(\tilde{y}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{\alpha} \mathbf{x}_1(\tilde{v}\tilde{c})$$

by [T-EXT]. Then by [T-STATE],

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : (\nu, \widetilde{\mathcal{T}})$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{x}_1(\tilde{y}\tilde{z}) \;\&\; \mathbf{x}_2(\tilde{y}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : \tilde{U}\,\widetilde{\mathcal{T}} \;\; [\!] \;\; \mathbf{x}_1 : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}_1), \mathbf{x}_2 : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}_2)$$

with

$$\forall i, \mathcal{T}_i = (\mathcal{T}_{1i} \cup \mathcal{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \;\&\; \mathbf{x}_2$$

with $z_i = x$. On the other hand, by $\lfloor$CHOICE$\rfloor$ in Figure 4.18, we have

$$\mathcal{T}_i = (\mathcal{T}_{1i} \cup \mathcal{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \;\&\; \mathbf{x}_2 \quad \xrightarrow{\alpha} \quad \mathcal{T}_{1i}'$$

This implies, by [TAU], we have:

$$(\Gamma,\Delta) \quad \xrightarrow{\alpha} \quad (\Gamma,\Delta')$$

where $\Delta = \tilde{c} : (\nu, \tilde{\mathcal{T}})$ and $\Delta' = \Delta \setminus \{\mathbf{x}_2 : (\nu, \tilde{U}\, \widetilde{\mathcal{T}_2})\}$.

Case [T-DELAY] Suppose:

$$\mathbf{x}(\tilde{x}) = \mathtt{delay}(t).\mathbf{x}'(\tilde{x}) \xrightarrow{t} \mathbf{x}'(\tilde{x})$$

by [DELAY]. Then by [STATE] ,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \triangleright \tilde{c} : \widetilde{\mathcal{T}}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = \mathtt{delay}(t).\mathbf{x}'(\tilde{y}\tilde{z})$$

has type

$$\vdash P \triangleright \mathbf{x} : (\nu, \tilde{U}\, \widetilde{\mathcal{T}}) \;\; [\!] \;\; \mathbf{x}' : ((\nu + t), \tilde{U}\, \widetilde{\mathcal{T}'})$$

with

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}$$

On the other hand, we have

$$\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x} \xrightarrow{t} \mathcal{T}_i'$$

This implies, with $\Gamma \vdash \tilde{y} : U$, we have

$$(\Gamma,\Delta) \xrightarrow{t} (\Gamma,\Delta')$$

The proof of the remaining rules are same as those of Theorem 4.5.2. This concludes all cases for Subject Transition and Reduction.

□

We say P has ***a type error*** if expressions in P contain either a type error for a value or constant in the standard sense or ***a reception error***.

Theorem 5.5.3 (Type Safety)**.** *Suppose* $\Gamma \vdash \mathbf{N}$. *For any* $\mathbf{N}'$ *such that* $(\nu, \mathbf{N}) \longrightarrow^* (\nu', \mathbf{N}')$, $\mathbf{N}'$ *has no type error.*

Proof. Suppose $\Gamma \vdash \mathbf{N}$ has a type error. Then there are reductions such that $(\nu, \mathbf{N}) \longrightarrow^* (\nu', (\nu s)(s : (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \cdot h \,||\, P_1 \,||\, \cdots \,||\, P_n) \,||\, \mathbf{N}')$ and $P_j \xrightarrow{s[\mathtt{p},\mathtt{q}]?\langle v \rangle} P_j$ where v does not meet the specified type. Suppose $\Gamma \vdash P_j \triangleright \Delta, s[\mathtt{p}] : (\nu', \mathcal{T}_j)$. By labelled transition of local types in Section 5.3, $(\nu', \mathcal{T}_j) \xrightarrow{?\langle \mathtt{q}, l\langle U \rangle \rangle \{\delta, \lambda\}} (\nu'', \mathcal{T}_j')$. Since P_j has a type error, $\Gamma \not\vdash v : U$, which contradicts Theorem 5.5.2. □

Theorem 5.5.4 (Time-error freedom)**.** *If* $\Gamma \vdash P \triangleright \Delta$*, and* $P \to^* P'$ *then* $P' \not\equiv \mathtt{error}$

Proof. We prove time-error freedom through similar steps:

- **Step 1 - extend processes** An error state denoted as `error` is a state when a process violates the clock constraints associated with the action prescribed by the corresponding local timed types. To model the reductions to error states, we extend the syntax of processes with clock constraints, reset predicates and clock assignment.

$$
\begin{array}{lll}
v ::= & a \mid s[\mathtt{p}] \mid \mathtt{true} \mid \mathtt{false} \mid \ldots & \text{values} \\
e ::= & v \mid x \mid e \wedge e \mid \ldots & \text{expression} \\
h ::= & \emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle) & \text{messages} \\
c ::= & s[\mathtt{p}] \mid y & \text{session names/variables} \\
P ::= & \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} & \text{definition} \\
P ::= & & \text{process transition} \\
 & \mid \mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}) & \text{init} \\
 & \mid \mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}) & \text{request} \\
 & \mid \mathbf{x}(\tilde{x}) = x!\langle \mathtt{p}, l\langle e \rangle \{\delta_o, \lambda_o\} \rangle.\mathbf{x}'(\tilde{e}) & \text{send} \\
 & \mid \mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y) \{\delta_i, \lambda_i\} \rangle.\mathbf{x}'(\tilde{e}) & \text{receive} \\
 & \mid \mathbf{x}(\tilde{x}) = \mathtt{delay}(t).\mathbf{x}'(\tilde{x}) & \text{delay} \\
 & \mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z}) & \text{parallel} \\
 & \mid \mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'') & \text{conditional} \\
 & \mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) & \text{external choice} \\
 & \mid \mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x}) & \text{join} \\
 & \mid \mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) & \text{merge} \\
 & \mid \mathbf{x}(\tilde{x}) = (\nu a)\ \mathbf{x}'(a\tilde{x}) & \text{new name} \\
 & \mid \mathbf{x}(\tilde{x}) = \mathbf{0} & \text{null} \\
 & \mid \mathtt{error} & \text{Error} \\
\mathbf{X} ::= & & \text{state} \\
 & \mid \mathbf{x}(\tilde{v}) & \text{thread} \\
 & \mid \mathbf{X} \mid \mathbf{X} & \text{parallel} \\
 & \mid (\nu a)\mathbf{X} & \text{restriction} \\
 & \mid \mathbf{0} & \text{null} \\
\mathbf{N} ::= & & \text{network} \\
 & \mid P & \text{def} \\
 & \mid \mathbf{N} \mid\mid \mathbf{N} & \text{parallel} \\
 & \mid (\nu a)\mathbf{N} & \text{new name} \\
 & \mid \mathbf{0} & \text{null} \\
 & \mid (\nu s)\mathbf{N} & \text{new session} \\
 & \mid (c, \nu) & \text{clock assignment} \\
 & \mid s : h & \text{queue} \\
 & \mid a\langle s \rangle[\mathtt{p}] & \text{invitation}
\end{array}
$$

$$\frac{\begin{array}{c} x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad e[\tilde{v}/\tilde{x}] \downarrow v \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}' \\ \nu \models \delta_o \quad \nu' = [\lambda_o \mapsto 0]\nu \end{array}}{\begin{array}{c} \mathbf{x}(\tilde{x}) = x!\langle \mathtt{p}, l\langle e\rangle\{\delta_o, \lambda_o\}\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \mid (s[\mathtt{q}], \nu) \\ \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v\rangle} \mathbf{x}'(\tilde{v}') \mid (s[\mathtt{q}], \nu') \end{array}} \quad \text{[T-SEND]}$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad \tilde{e}[\tilde{v}/\tilde{x}][v/y] \downarrow \tilde{v}' \qquad \nu \models \delta_i \quad \nu' = [\lambda_i \mapsto 0]\nu}{\begin{array}{c} \mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y)\{\delta_i, \lambda_i\}\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \mid (s[\mathtt{q}], \nu) \\ \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v'\rangle} \mathbf{x}'(\tilde{v}') \mid (s[\mathtt{q}], \nu') \end{array}} \quad \text{[T-RCV]}$$

$$\frac{\forall i \in \{1, ..., n\} \qquad s \notin fn(P_i)}{\begin{array}{c} \overline{\mathtt{a}}[n](y) \mid \prod_{i\in\{2,...,n\}} a[i](y).P_i \\ \rightarrow (\nu s)(\prod_{i\in\{1,...,n\}}(P_i[s[i]/y] \mid (s[i], \nu_0)) \mid s : \emptyset) \end{array}} \quad \text{[T-LINK]}$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad \neg\nu \models \delta_o}{\begin{array}{c} \mathbf{x}(\tilde{x}) = x!\langle \mathtt{p}, l\langle e\rangle\{\delta_o, \lambda_o\}\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \mid (s[\mathtt{q}], \nu) \\ \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v\rangle} \mathtt{error} \mid (s[\mathtt{q}], \nu) \end{array}} \quad \text{[T-ESEND]}$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad \neg\nu \models \delta_i}{\begin{array}{c} \mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y)\{\delta_i, \lambda_i\}\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \mid (s[\mathtt{q}], \nu) \\ \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v'\rangle} \mathtt{error} \mid (s[\mathtt{q}], \nu) \end{array}} \quad \text{[T-ERCV]}$$

Figure 5.11.: Extended Operational Semantics with Error State

- **Step 2 - extend reductions** We extend the reduction rules for [T-SEND] and [T-RCV] shown in Figure 5.11, with the rest reductions remain the same. In addition, we also define the rules [T-LINK] to extend session initiation by adding an initial clock assignment ν_0 for each participant of the new session.
- **Step 3 - extend validation rules** Below we show the validation rules for processes with errors and clocks. Here we only show the rules that are different from original validation rules in Figure 5.12.

Assume that $\Gamma \vdash P \triangleright \Delta$, and $P \longrightarrow^* P \longrightarrow P'$. We proceed with the assumption that by induction $P \not\equiv \mathtt{error}$, and continue with case study on the last reduction $P \longrightarrow P'$.

In the case study, we investigate all the reduction rules for processes with errors and clocks. The inductive rules, e.g [T-PUT] and other reduction rules for P and

$$\frac{\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \quad \tilde{y} : \tilde{U} \vdash x : \langle \mathbf{G} \rangle \quad \forall i, \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \mid (y, \nu_0) \triangleright \mathbf{x} : \tilde{U}\,\widetilde{\mathcal{T}} \;[\!]\; \mathbf{x}' : (\nu_0, \tilde{U}'\,\widetilde{\mathcal{T}'})} \quad \text{[T-INIT]}$$

$$\frac{\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \quad \tilde{y} : \tilde{U} \vdash x : \langle \mathbf{G} \rangle \quad \forall i, \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}' \quad \mathcal{T} = \mathbf{G} \upharpoonright \mathtt{p}}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \mid (y, \nu_0) \triangleright \mathbf{x} : \tilde{U}\,\widetilde{\mathcal{T}} \;[\!]\; \mathbf{x}' : (\nu_0, \tilde{U}'\,\widetilde{\mathcal{T}'\mathcal{T}})} \quad \text{[T-REQ]}$$

$$\frac{\begin{array}{c}\tilde{y} : \tilde{U} \vdash e : U \quad \tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \qquad \nu \models \delta_o \\ \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =! \langle \mathtt{p}, l\langle U \rangle \rangle B_o.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'\end{array}}{\begin{array}{c}\vdash \mathbf{x}(\tilde{y}\tilde{z}) = z_i\,!\,\langle \mathtt{p}, l\langle e \rangle, \{\delta_o, \lambda_o\} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \mid (z_i, \nu) \\ \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}' : ([\lambda_o \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}'})\end{array}} \quad \text{[T-SEND]}$$

$$\frac{\begin{array}{c}\tilde{y} : \tilde{U}, y : U \vdash \tilde{e} : \tilde{U}' \qquad \nu \models \delta_i \\ \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =? \langle \mathtt{p}, \{l.U\}_{B_i} \rangle.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'\end{array}}{\begin{array}{c}\vdash \mathbf{x}(\tilde{y}\tilde{z}) = z_i\,?\,\langle \mathtt{p}, l(y), \{\delta_i, \lambda_i\} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \mid (z_i, \nu) \\ \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}' : ([\lambda_i \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}'})\end{array}} \quad \text{[T-RECV]}$$

$$\frac{\begin{array}{c}\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \quad \nu \models \delta_o \\ \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =! \langle \mathtt{p}, l\langle \mathcal{T} \rangle \rangle B_o.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'\end{array}}{\begin{array}{c}\vdash \mathbf{x}(\tilde{y}\tilde{z}y) = z_i\,!\,\langle \mathtt{p}, l\langle y \rangle, \{\delta_o, \lambda_o\} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \mid (z_i, \nu) \\ \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}\mathcal{T}}) \;[\!]\; \mathbf{x}' : ([\lambda_o \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}'})\end{array}} \quad \text{[T-DELEG]}$$

$$\frac{\begin{array}{c}\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \quad \nu \models \delta_i \\ \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =? \langle \mathtt{p}, \{l.\mathcal{T}\}_{B_i} \rangle.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}'\end{array}}{\begin{array}{c}\vdash \mathbf{x}(\tilde{y}\tilde{z}) = z_i\,?\,\langle \mathtt{p}, l(y), \{\delta_i, \lambda_i\} \rangle.\mathbf{x}'(\tilde{e}\tilde{z}y) \mid (z_i, \nu) \\ \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}' : ([\lambda_i \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}'\mathcal{T}})\end{array}} \quad \text{[T-CATCH]}$$

$$\frac{\mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} = \mathbf{x}}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = \mathtt{delay}(t).\mathbf{x}'(\tilde{y}\tilde{z}) \mid (z_i, \nu) \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]\; \mathbf{x}' : ((\nu + t), \tilde{U}\,\widetilde{\mathcal{T}'})} \quad \text{[T-DELAY]}$$

$$\frac{\forall i, \mathcal{T}_i = \mathsf{def}\ \mathbf{x} = \mathsf{end}\ \mathsf{in}\ \mathbf{x}}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{0} \mid (z_i, \nu) \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;[\!]} \quad \text{[T-NIL]}$$

Figure 5.12.: Extended Typing Rules with Time Assertions

N, are straightforward. If the process is reduced by the rules for **X** like [T-SEND], then $P' \not\equiv \texttt{error}$ because these reductions do not introduce `error` processes, which are run-time processes and cannot appear as a continuation of P. Thus, the only reductions that could introduce error processes are [T-ESEND] and [T-ERECV]. Consider rule [T-ESEND], and the other rule is similar. Assume that [T-ESEND] can be applied on P, then P can be of the form $\mathbf{x}(\tilde{x}) = x\,!\,\langle \mathrm{p}, l\langle e\rangle, \{\delta_o, \lambda_o\}\rangle.\mathbf{x}'(\tilde{e})$. Then by [T-ESEND],

$$\neg\nu \models \delta_o$$

By the assumption P is well-typed, and the derivation of $\mathbf{x}(\tilde{x}) = x\,!\,\langle \mathrm{p}, l\langle e\rangle, \{\delta_o, \lambda_o\}\rangle.\mathbf{x}'(\tilde{e})$ is done by the rule [T-SEND] among all the validation rules

$$\frac{\begin{array}{c}\tilde{y} : \tilde{U} \vdash e : U \quad \tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \qquad \nu \models \delta_o \\ \mathcal{T}_i = \mathcal{T}_i' \uplus \mathbf{x} =! \langle \mathrm{p}, l\langle U\rangle\rangle B_o.\mathbf{x}' \quad \forall j \neq i, \mathcal{T}_j = \mathcal{T}_j' \uplus \mathbf{x} = \mathbf{x}' \end{array}}{\begin{array}{c} \vdash \mathbf{x}(\tilde{y}\tilde{z}) = z_i\,!\,\langle \mathrm{p}, l\langle e\rangle, \{\delta_o, \lambda_o\}\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \mid (z_i, \nu) \\ \triangleright \mathbf{x} : (\nu, \tilde{U}\,\widetilde{\mathcal{T}}) \;\; [\!] \;\; \mathbf{x}' : ([\lambda_o \mapsto 0]\nu, \tilde{U}'\,\widetilde{\mathcal{T}}') \end{array}}\,[\text{T-SEND}]$$

with the condition $\nu \models \delta_o$, which contradicts with the fact that $\neg\nu \models \delta_o$ when the reduction rule can be applied. Thus, the fact is that none of the reductions that introduces the errors can be applied, with the fact that $P \not\equiv \texttt{error}$ leads to the thesis. □

5.6. Sufficient Conditions for Progress Enablement and Time-progress of Processes

In this section, we define conditions for the progress enabling property and time-progress property of processes. The satisfaction of this property guarantees that global types will not get stuck because of unsuitable time assertions. In Section 5.6.1, we start with defining the time graph that presents every state and transition that could occur during reduction. In Section 5.6.2 based on the global time graph, we derive the dependency graph for a specified state and extract the time assertions in this graph. From the trace of time assertions reaching the destination state, we check if that state is progress enablement or not. The method of examining progress-enablement in 5.6.2 is adopted from [BLY15] with adaptation on choreographical global graphs. In Section 5.6.3, we show the time-progress property, which is similar to that in Section 3.6.5; based on the progress enablement property, we show that timed choreographical processes

enjoy progress property if the processes with time satisfy progress property.

5.6.1. Time Graph

We define the $_^*$ function over vector of states:

$$\vec{\mathbf{x}}^* \quad = \quad \begin{cases} \vec{\mathbf{x}_1}^* & \text{if } \vec{\mathbf{x}} = (\vec{\mathbf{x}_1}, \vec{\mathbf{x}_2}) \text{ and } \vec{\mathbf{x}_1}^* = \vec{\mathbf{x}_2}^* \\ \vec{\mathbf{x}} & \text{otherwise} \end{cases}$$

Then we define

$$\mathbf{x} \Subset \vec{\mathbf{x}}' \iff \begin{cases} \mathbf{x} = \mathbf{x}'' \quad \text{and} \quad \vec{\mathbf{x}}' = (\mathbf{x}''), \text{ or} \\ \vec{\mathbf{x}}' = (\vec{\mathbf{x}_1}, \vec{\mathbf{x}_2}) \wedge \mathbf{x} \Subset \vec{\mathbf{x}_i} \wedge i \in \{1,2\} \end{cases}$$

Definition 5.6.1 (Parellel Automation)**.** The composition of $M_i = (\mathbf{X}_i, {\mathbf{x}_0}^i, \mathbf{T}_i)$, $i \in \{1,2\}$, written $M_1 || M_2$, is the automation $((\mathbf{X}_1 \times \mathbf{X}_2)^*, ({\mathbf{x}_0}^1, {\mathbf{x}_0}^2), \mathbf{T})$ s.t.

$$((\mathbf{x}_1, \mathbf{x}_2)^*, l, (\mathbf{x}_1', \mathbf{x}_2')^*) \in \mathbf{T} \Longleftrightarrow \begin{cases} (\mathbf{x}_i, l, \mathbf{x}_i') \in \mathbf{T}_i & \text{if } \mathbf{x}_j = \mathbf{x}_j', \mathbf{x}_i \notin \mathbf{X}_j, \text{ and } i \neq j \{1,2\} \\ (\mathbf{x}_i, l, \mathbf{x}_i') \in \mathbf{T}_i & \text{if } \mathbf{x}_i \Subset \mathbf{X}_j, \text{ and } i \neq j \{1,2\} \end{cases}$$

Before introducing the time graph, we extend the labelled messages syntax to send/receive action as

$$\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle \{A\}; \mathbf{x}' \Rightarrow \begin{cases} \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle \{A\}; \mathbf{x}_r \text{ and} \\ \mathbf{x}_r = \mathtt{p} \rightsquigarrow \mathtt{p}' : l\langle U \rangle \{A\}; \mathbf{x} \end{cases}$$

Then the time graph of global types G is obtained by applying a *Time Transformation Function* denoted as $TT(G, \mathbf{n}, \mathbb{N})$. The definition is as below:

$$
TT(G,\mathbf{n},\mathbb{N}) = \begin{cases}
\langle\{\mathbf{n}\},\mathbf{n},\emptyset\rangle & \text{if } \mathbf{n} = \mathsf{end} \text{ or } \mathbf{n} \in \mathbb{N} \\
\langle \mathbf{X}\cup\{\mathbf{n}\},\mathbf{n},\mathbf{T}\cup\{(\mathbf{n},l,\mathbf{n}')\}\rangle & \text{if } \mathbf{n} = \mathtt{s} \to \mathtt{r} : l_g\langle U\rangle\{A\};\mathbf{n}' \text{ and } l = sr!l_g\{A\} \text{ or} \\
 & \text{if } \mathbf{n} = \mathtt{s} \rightsquigarrow \mathtt{r} : l_g\langle U\rangle\{A\};\mathbf{n}' \text{ and } l = sr?l_g\{A\} \\
 & \text{and } \langle \mathbf{X},\mathbf{n}',\mathbf{T}\rangle = TT(G,\mathbf{n}',\mathbb{N}\cup\{\mathbf{n}\}) \\
\langle\{\mathbf{n}\}\cup\mathbf{X}_1\cup\mathbf{X}_2,\mathbf{n}, & \text{if } \mathbf{n} = \mathbf{n}_1 + \mathbf{n}_2 \text{ and } \langle \mathbf{X}_i,\mathbf{n}_i,\mathbf{T}_i\rangle = \\
\mathbf{T}_1\cup\mathbf{T}_2\cup(\mathbf{n},\varepsilon,\mathbf{n}_1)\cup(\mathbf{n},\varepsilon,\mathbf{n}_2)\rangle & TT(G,\mathbf{n}_i,\mathbb{N}\cup\{n\}), i\in\{1,2\} \\
\langle \mathbf{X}\cup\{\mathbf{n}\},\mathbf{n},\mathbf{T}\cup\{(\mathbf{n},\varepsilon,\mathbf{n}'')\}\rangle & \text{if } \mathbf{n} = \mathbf{n}_1 \mid \mathbf{n}_2 \text{ and } \langle \mathbf{X},\mathbf{n}'',\mathbf{T}\rangle = \\
 & TT(G,\mathbf{n}_1,\mathbb{N}\cup\{n\})||TT(G,\mathbf{n}_2,\mathbb{N}\cup\{n\}) \\
\langle \mathbf{X}\cup\{\mathbf{n}_i\},\mathbf{n}_i,\mathbf{T}\cup(\mathbf{n}_i,\varepsilon,\mathbf{n})\rangle & \text{if } \mathbf{n}_1 \mid \mathbf{n}_2 = \mathbf{n} \text{ or } \mathbf{n}_1 + \mathbf{n}_2 = \mathbf{n} \\
 & \text{and } \langle \mathbf{X},\mathbf{n},\mathbf{T}\rangle = TT(G,\mathbf{n},\mathbb{N}\cup\{\mathbf{n}_i\}), \in \{1,2\}
\end{cases}
$$

5.6.2. Checking Progress Enabling

Understanding the past We check that G has progress by analysing paths, i.e., sequences of events, in $TT(G,\mathbf{n},\mathbb{N})$. Since $TT(G,\mathbf{n},\mathbb{N})$ gives an over-approximation of the causal dependencies between actions, we will construct a graph of the actual dependencies of the underlying actions of a path. We define a path φ as a sequence of $\mathbf{n}_0\mathbf{n}_1\cdots\mathbf{n}_k$, where for all $i\in[0,k-1]$ there exists l such that $(\mathbf{n}_i,l,\mathbf{n}_{i+1})\in\mathbb{N}$. We compute the underlying actions of a path via the function:

$$
\mathsf{nodes}(\mathbf{n}_0\mathbf{n}_1\cdots\mathbf{n}_k) \stackrel{\text{def}}{=} \mathbf{n}_0\lfloor_{\mathsf{id}(\mathbf{n}_0)}\cdot\mathbf{n}_1\lfloor_{\mathsf{id}(\mathbf{n}_1)}\cdots\mathbf{n}_k\lfloor_{\mathsf{id}(\mathbf{n}_k)} \qquad (k\geq 0)
$$

And

$$
\mathbf{n}\lfloor_{\mathsf{id}(\mathbf{n})} \stackrel{\text{def}}{=} \begin{cases}
sr!l_g\{\delta_o,\lambda_o\} & \text{if } \mathbf{n} = \mathtt{s} \to \mathtt{r} : l_g\langle U\rangle\{\delta_o,\lambda_o,\delta_i,\lambda_i\};\mathbf{n}' \\
sr?l_g\{\delta_i,\lambda_i\} & \text{if } \mathbf{n} = \mathtt{s} \rightsquigarrow \mathtt{r} : l_g\langle U\rangle\{\delta_o,\lambda_o,\delta_i,\lambda_i\};\mathbf{n}' \\
\emptyset & \text{otherwise}
\end{cases}
$$

Remarkably, given a path φ and two actions ℓ_i and ℓ_j in $\mathsf{nodes}(\varphi)$, $i<j$ does not imply that there is a causal dependency between ℓ_i and ℓ_j.

We define the graph of dependencies of an action ℓ_k in a sequence of actions $\ell_1\cdots\ell_k$ This graph is an abstraction of all actions on which ℓ_k depends. We

consider two kinds of dependencies: output/input dependencies between send and corresponding receive actions, and local dependencies within a single participant.

Definition 5.6.1 (Graph of Dependencies)**.** Let $\mathsf{dep}(\varepsilon;\ell) \stackrel{\text{def}}{=} \emptyset$ and

$$\mathsf{dep}(\rho\cdot\ell_1;\ell_2) \stackrel{\text{def}}{=} \begin{cases} \{(\ell_1,\ell_2)\} \cup \mathsf{dep}(\rho;\ell_i)_{i=1,2} & \text{if } \ell_1 = \mathtt{pq}!a(g_1,\lambda_1),\ \ell_2 = \mathtt{pq}?a(g_2,\lambda_2) \\ \{(\ell_1,\ell_2)\} \cup \mathsf{dep}(\rho;\ell_1) & \text{if } \mathsf{subj}(\ell_1) = \mathsf{subj}(\ell_2) \text{ and} \\ & \{\ell_1 = \mathtt{pq}?a(g_1,\lambda_1),\ \ell_2 = \mathtt{qp}'!a(g_2,\lambda_2) \\ & \text{or} \\ & \ell_1 = \mathtt{pq}!a(g_1,\lambda_1),\ \ell_2 = \mathtt{q}'\mathtt{p}?a(g_2,\lambda_2)\} \\ \mathsf{dep}(\rho;\ell_2) & \text{otherwise} \end{cases}$$

The graph of dependencies of $\rho = \ell_1 \cdots \ell_k$ ($k>0$), written $\mathsf{DG}(\rho)$, is the graph (D,A) s.t. $A = \mathsf{dep}(\ell_1 \cdots \ell_{k-1};\ell_k) \setminus \{(\ell_i,\ell_k) \mid 1 \le i < k\}$ and $D = \{\ell_r \mid \exists(\ell_i,\ell_j) \in A \wedge r \in \{i,j\}\}$. This is a graph whose nodes form a subset of $\{\ell_1,\ldots,\ell_{k-1}\}$ and whose edges model causal dependencies between actions and the graph is computed backwards starting from ℓ_k.

$$\mathsf{idx}(\rho) \stackrel{\text{def}}{=} \{i \mid \ell_i \in D\}$$

$$W_x^i(\rho) \stackrel{\text{def}}{=} \begin{cases} v_i - v_j & \text{if } 0 \le j = max\{j < i \mid \ell_j \in D \wedge x \in \ell_j\} \\ v_i & \text{otherwise} \end{cases}$$

$$\mathsf{allpast}(\rho) \stackrel{\text{def}}{=} \bigwedge_{i \in \mathsf{idx}(\rho)} \mathsf{absolute}_\rho(\ell_i)$$

$$\mathsf{elapse}(\rho) \stackrel{\text{def}}{=} \bigwedge_{(\ell_i,\ell_j)\in A} v_i \le v_j$$

$$\mathsf{absolute}_\rho(\ell_i) \stackrel{\text{def}}{=} \mathsf{guard}(\ell_i)\ \left\{x \mapsto W_x^i(\rho)\right\}_{x\in\mathcal{X}}$$

Given a graph of dependencies $\mathsf{DG}(\rho)$, we define several functions that allow us to construct predicates that model the past. First, we transform the guard of an action ℓ_i such that its solutions are the possible *absolute* times, that is total time passing from the initial configuration of the system, in which one may execute ℓ_i with taking into account the last reset of each clock in ρ. Next, we accumulate the information in $\mathsf{DG}(\rho)$, by (1) in $\mathsf{idx}(\rho)$, recording the indices of all the actions on which ℓ_k depends; (2) in $\mathsf{allpast}(\rho)$, calculating the conjunction of all constraints in absolute time ; and (3) in $\mathsf{elapse}(\rho)$, recording the fact that time never decreases between two causally dependent actions.

Predicting the future We now give the definition of progress enabling that allows to check whether the past implies the existence of a satisfiable future. We use the functions defined above to check whether a given event in $TT(G,\mathbf{n},\mathbb{N})$ can meet its time constraints.

Definition 5.6.2 (Progress enabling (PE))**.** A node $\mathbf{n}$ is *progress enabling* (PE) for $\mathtt{p} \in \mathsf{id}(\mathbf{n})$ if for *all* paths φ such that $\mathbf{n}_0 \overset{\varphi}{\hookrightarrow} \mathtt{n}$, letting: $\rho = \mathsf{nodes}(\varphi \cdot \mathbf{n})$ and $k = |\rho|$, $\ell_k = \mathbf{n}\lfloor_{\mathtt{p}}$, $\vec{v} = \{v_i \mid i \in \mathsf{idx}(\rho)\}$; the following holds

$$\forall \vec{v}\ \exists v_k :\ \mathsf{allpast}(\rho) \wedge \mathsf{elapse}(\rho) \implies \mathsf{absolute}_{\rho}(\ell_k) \wedge \textstyle\bigwedge_{v_i \in \vec{v}} v_i \leq v_k$$

A pair $(\mathbf{n},\varphi)$ is *recursively progress enabling* (RPE) for $P \subseteq \mathcal{P}$ if $\varphi = \varepsilon$ and $P = \emptyset$; or if $\mathbf{n}$ is PE for $\mathbf{n}\lfloor_{\mathsf{id}(\mathbf{n})}$ and $(\mathbf{n}',\varphi')$ is RPE for $P \setminus \mathsf{id}(\mathbf{n})$ with $\varphi = e \cdot \varphi'$ and $\mathbf{n} \overset{e}{\hookrightarrow} \mathbf{n}'$.

For a pair $(\mathtt{n},\varphi)$, if $\mathsf{guard}(\ell_k) = \bigvee_{j \in J} \delta_j$, the pair $(\mathtt{n},\varphi)$ is *wait-free* for $P \subseteq \mathcal{P}$ if for every $j \in J$, the following is satisfied

$$\forall \vec{v}\ \exists v_k :\ \mathsf{allpast}(\rho) \wedge \mathsf{elapse}(\rho) \implies \mathsf{absolute}_{\rho}(\delta_j) \wedge \textstyle\bigwedge_{v_i \in \vec{v}} v_i \leq v_k$$

where $\mathsf{absolute}_{\rho}(\delta_j) = \delta_j \left\{x \mapsto W_x^i(\rho)\right\}_{x \in \mathcal{X}}$

Given a node $\mathbf{n}$ in $TT(G,\mathbf{n},\mathbb{N})$ and a participant $\mathtt{p}$, the above definition ensures that for all possible past clock valuations, there exists a *future* time where participant $\mathtt{p}$ has the possibility to execute action $\mathbf{n}\lfloor_{\mathsf{id}(\mathbf{n})}$.

5.6.3. Time-Progress for Timed Choreographical Processes

In order to verify if a validated program is guaranteed to proceed until the completion, we still need to define the theorem of time-progress. Again, we define an *erasure* function for timed choreographical processes by removing the delays in P and leaving unchanged the untimed parts. Under the assumption that $\mathtt{erase}(P)$ can progress, we investigate under which conditions P is also ensured to progress.

Definition 5.6.2 (Time erasure)**.** The erasure $\mathtt{erase}(P)$ of time from a process P

is defined as below:

$$
\begin{array}{ll}
\texttt{erase}(\mathbf{x}) = x\langle \mathbf{G}\rangle.\texttt{erase}(\mathbf{x}') & \text{if } \mathbf{x}(\tilde{x}) = x\langle \mathbf{G}\rangle.\mathbf{x}'(\tilde{e}) \\
\texttt{erase}(\mathbf{x}) = x[\mathrm{p}](y).\texttt{erase}(\mathbf{x}') & \text{if } \mathbf{x}(\tilde{x}) = x[\mathrm{p}](y).\mathbf{x}'(\tilde{e}) \\
\texttt{erase}(\mathbf{x}) = x\,!\,\langle \mathrm{p}, l\langle e\rangle\rangle.\texttt{erase}(\mathbf{x}') & \text{if } \mathbf{x}(\tilde{x}) = x\,!\,\langle \mathrm{p}, l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}) \\
\texttt{erase}(\mathbf{x}) = x\,?\,\langle \mathrm{p}, l(y)\rangle.\texttt{erase}(\mathbf{x}') & \text{if } \mathbf{x}(\tilde{x}) = x\,?\,\langle \mathrm{p}, l(y)\rangle.\mathbf{x}'(\tilde{e}) \\
\texttt{erase}(\mathbf{x}) = \texttt{erase}(\mathbf{x}') & \text{if } \mathbf{x}(\tilde{x}) = \texttt{delay}(t).\mathbf{x}'(\tilde{x}) \\
\texttt{erase}(\mathbf{x}) = \texttt{erase}(\mathbf{x}_1) \mid \texttt{erase}(\mathbf{x}_2) & \text{if } \mathbf{x}(\tilde{x}) = \mathbf{x}_1(\tilde{y}) \mid \mathbf{x}_2(\tilde{z}) \\
\texttt{erase}(\mathbf{x}) = \texttt{if } e \texttt{ then erase}(\mathbf{x}_1) \texttt{ else erase}(\mathbf{x}_2) & \\
\qquad\qquad \text{if } \mathbf{x}(\tilde{x}) = \texttt{if } e \texttt{ then } \mathbf{x}_1(\tilde{e}') \texttt{ else } \mathbf{x}_2(\tilde{e}'') & \\
\texttt{erase}(\mathbf{x}) = \texttt{erase}(\mathbf{x}_1) \,\&\, \texttt{erase}(\mathbf{x}_2) & \text{if } \mathbf{x}(\tilde{x}) = \mathbf{x}_1(\tilde{x}) \,\&\, \mathbf{x}_2(\tilde{x}) \\
\texttt{erase}(\mathbf{x}_1) \mid \texttt{erase}(\mathbf{x}_2) = \texttt{erase}(\mathbf{x}) & \text{if } \mathbf{x}_1(\tilde{y}) \mid \mathbf{x}_2(\tilde{z}) = \mathbf{x}(\tilde{x}) \\
\texttt{erase}(\mathbf{x}_1) + \texttt{erase}(\mathbf{x}_2) = \texttt{erase}(\mathbf{x}) & \text{if } \mathbf{x}_1(\tilde{x}) + \mathbf{x}_2(\tilde{x}) = \mathbf{x}(\tilde{x}) \\
\texttt{erase}(\mathbf{x}) = (\nu a)\ \texttt{erase}(\mathbf{x}') & \text{if } \mathbf{x}(\tilde{x}) = (\nu a)\ \mathbf{x}'(a\tilde{x}) \\
\texttt{erase}(\mathbf{x}) = 0 & \text{if } \mathbf{x}(\tilde{x}) = \mathbf{0} \\
\texttt{erase}(s:h) = s:h &
\end{array}
$$

Definition 5.6.3 (Deferrability)**.** A process P is *deferrable* iff one of the following holds:

$P = \texttt{delay}(t).\mathbf{x}$ and $t > 0$
$P = \texttt{delay}(t).\mathbf{x}$, $t = 0$ and $\mathbf{x}$ is deferrable
$P = \mathbf{x}_1 \mid \mathbf{x}_2$ and both $\mathbf{x}_1$ and $\mathbf{x}_2$ are deferrable
$P \in \{\mathbf{0}, \mathbf{t}\}$
$P = s : h$
$P = (\nu a)P'$ and P' is deferrable

Intuitively, a process is *non-deferrable* if no action ready to execute in any of its threads is a time action. We say that P is *non deferrable* if P is not deferrable.

Lemma 5.6.4. *If* $\texttt{erase}(P) \longrightarrow$ *then* $\texttt{erase}(P)$ *is of one of the forms that are as same as the eight cases in Lemma 3.6.13 in Section 3.6.5 with the ninth case (external choice)*

9. $\mathbf{x}'(\tilde{x}) \,\&\, \mathbf{x}''(\tilde{x})$

Lemma 5.6.5. *Besides the eight cases in Lemma 3.6.14, we need to add external choice.*

9. *if* $\texttt{erase}(P) = \texttt{erase}(\mathbf{x}_1) \,\&\, \texttt{erase}(\mathbf{x}_2)$ *then* $\exists \mathbf{x}_1', \mathbf{x}_2', t \geq 0$ *such that* $P \equiv \texttt{delay}(t).\mathbf{x}_1' \,\&\, \mathbf{x}_2'$ *and* $\mathbf{x}_i = \texttt{erase}(\mathbf{x}_i')$ *for* $i \in \{1,2\}$.

Proof. The proof is mechanical by Lemma 5.6.4. □

Lemma 5.6.6. *If P is non-deferrable and* $\texttt{erase}(P) \longrightarrow P'$ *then* $P \longrightarrow P''$ *with* $\texttt{erase}(P'') = P'$.

Proof. Mechanical, by induction on the derivation of $\texttt{erase}(P)$, proceeding by case analysis using Lemma 3.6.13, Lemma 3.6.14 (for first eight cases), Lemma 5.6.4 and Lemma 5.6.5 (for external choice case). □

Proposition 5.6.7 (Conformance)**.** *If* $P \longrightarrow P'$ *then* $\texttt{erase}(P) \longrightarrow^* \texttt{erase}(P')$.

Similarly, we define that: (1) P is a *deadlock process* if $P \longrightarrow^* P'$ where $P' \not\longrightarrow$ and $P' \not\equiv 0$, and (2) P is session delay if for each delay process of the form $\texttt{delay}(t).P'$ occurring in P, there are no session request and session accept in P'. We say that Γ is progress enabling if for all $u \in dom(\Gamma)$, $\Gamma(u)$ is progress enabling.

Lemma 5.6.8 (Time progress - parallelism and interleaved sessions)**.** *Let* Γ *be a* progress enabling *mapping,* $\Gamma \vdash P_0 \triangleright \emptyset$*, and* $P_0 \longrightarrow^+ P$ *where P is session delay and*

$$P \equiv \texttt{delay}(t).P \mid R \qquad \textit{for some } t > 0 \textit{ and non-deferrable } R$$

If $\texttt{erase}(P)$ *is not a deadlock process and* $\texttt{erase}(P) \longrightarrow$ *then* $R \longrightarrow$.

Proof. We assume that for some L

$$R = R' \mid \prod_{l \in L} s_l : h_l$$

We continue by induction on the complexity of R', proceeding by case analysis on the possible form of process R'.

In all other cases, one can immediately show that R' reduces, and for receive and session receive are the complex cases.

- R' cannot be of the form $x\langle \mathbf{G} \rangle.R''$ or $x[\mathrm{p}](y).R''$ since (1) $\texttt{erase}(P)$ is deadlock free, which means that all the other corresponding session requests or accepts to start session y are in P, and (2) P is session delay, which means that none of the corresponding session accepts/delay can be in P_i.

- R' cannot be a delay process as P it is non deferrable (Definition 5.6.3) by hypothesis,

- if R' is of the form $x!\langle \mathtt{p}, l\langle e\rangle\rangle.R''$ then one can immediately show that R can reduce by $\lfloor$T-SEND$\rfloor$.
- in the inductive cases (e.g., for hiding and parallel processes) the result follows by induction.

We will show the case for R' being an input process.

Case $R' \equiv s[\mathtt{p}] : x?\langle \mathtt{p}, l(y)\rangle.R''$. In this case:

$$R \equiv s[\mathtt{p}] : x?\langle \mathtt{p}, l(y)\rangle.R'' \mid \prod_{l\in L} s_l : h_l$$

We continue with contradiction by assuming Γ progress enabling, $\mathtt{erase}(P)$ deadlock free, $\mathtt{erase}(P) \longrightarrow$ and $R \not\longrightarrow$. If $R \not\longrightarrow$ we have the following cases depending on where the corresponding output of $s[\mathtt{p}] : x?\langle \mathtt{p}, l(y)\rangle.R''$ is:

1. the corresponding output appears (immediately or after some other input) in some continuation R'',
2. the corresponding output appears in P,
3. the corresponding output does not appear.

If (1) or (3) then we contradict that $\mathtt{erase}(P)$ is deadlock free. If (2) then we contradict the fact P_0 was validated against a progress enabling Γ: the property of progress enabling ensures that in all correct time implementations the output does not precede the input, hence the output for a non deferrable input does not occur after $t_i > 0$ time which contradicts fact that the output appears in P_i. □

Theorem 5.6.9 (Timed progress in interleaved sessions)**.** *Let* Γ *be a* progress enabling *mapping,* $\Gamma \vdash P_0 \triangleright \emptyset$*, and* $P_0 \longrightarrow^+ P$*. If* P *is session delay,* $\mathtt{erase}(P)$ *is not a deadlock process and if* $\mathtt{erase}(P) \longrightarrow$ *then* $P \longrightarrow$*.*

Proof. We prove Theorem 5.6.9 by induction on the proof of P, proceeding by case analysis on the possible forms of P according to Lemma 3.6.13 and Lemma 5.6.4. Cases from Lemma 3.6.13 (1) to (8) are identical to the proof of Theorem 3.6.18. Here we present the case for external choice.

Case Lemma 5.6.4 (9): the erasure does not affect the conditions of conditional statements as such conditions do not involve delays, thus decision of external choice immediately follows both P and $\mathtt{erase}(P)$.

□

Note that if P implements a single session then $\texttt{erase}(P)$ is deadlock free [DY12]; this yields the following corollary. We characterise processes implementing single sessions, or *simple*, as follows: a process P is simple if $P_0 \longrightarrow^* P$ for some P_0 such that $a : \langle G \rangle \vdash P_0 \triangleright \emptyset$, and $P_0 = \overline{a}[\mathtt{n}](y).P_1 \mid \prod_{i \in \{2,..,n\}} a[\mathtt{i}](y).P_i$ where $P_1, \ldots, P_n$ do not contain any name hiding, request/accept, and session receive/delegate.

Corollary 5.6.10 (Time progress in single sessions)**.** *Let G be* feasible *and* wait-free*, and P be a* simple process *with $a : \langle G \rangle \vdash P \triangleright \emptyset$. If $\texttt{erase}(P) \longrightarrow$, then there exists P' such that $\texttt{erase}(P) \longrightarrow P'$ and $P \longrightarrow^+ P''$ such that $\texttt{erase}(P'') = P'$.*

Proof. Since P is simple, by the Proposition of progress for a *simple* process in [DY12], P has no delegation input or output. Thus P is a *session delay* process. By Theorem 5.6.9, the corollary is proved. □

5.7. Related Works

In this section, we compare several related works on timed session types and choreographies.

Timed Multiparty Session Types: In [BYY14], we present the theory of timed multiparty session types that models real-time distributed systems with branching and recursions. The theory of time error-freedom and properties of feasibility and wait-freedom are defined to guarantee that modelled protocol enjoys progress property in terms of time. Compared with timed session types, the theory presented in this chapter is advantageous in two aspects: (1) by distinguishing parallel and choice operations, timed choreographical session types are more flexible and able to model complex real-time concurrent systems; (2) we define the property of progress-enabling as a general solution of feasibility and wait-freedom checking for timed choreographical session types, and we could verify this property with standard model checking tools.

While timed session types are mapped into a subset of communicating timed automata in [BYY14], the authors in [BLY15] solve a reverse problem, that is, constructing timed global types from a collection of CTAs. The authors classify CTAs with four properties and define characterisation for checking satisfiability of these properties. By referring multiparty compatibility property in CFSMs [DY13], an algorithm is presented to build timed global types in [BYY14] that are

trace equivalent to the original CTA systems. Though progress-enabling property is proposed in this work, we check this property from timed choreographical global types, while in [BLY15], this property is verified against local synchronous transition systems. In addition, as authors base their work on the theory of timed session types, where fork/join operations are absent from global syntax, the constructed global types may be not optimised.

Choreographies: In [DY12], the authors extend the theory of multiparty session types with parallel operations for flexible expressions and precise thread management. Also, the projected local types are translated into the format of CFSMs denoted as multiparty session automata and properties like buffer boundedness, liveness and progress are defined. A recent work [LTY15] studies the reverse direction to construct global session types from local CFSMs. The authors present the characterisation of a subset of CFSMS that can be constructed into global graphs and define an algorithm to convert the CFSMs into choreographical global session types in [DY12]. Compared with these two works, the theory of timed choreographical session types absorbs the strengths, which are flexible fork/join, choice/merge and constructions of global graphs for tracking traces. Also, the extended theory has the potential to translate timed choreographical session types into a large set of CTAs and vice versa.

5.8. Summary

Motivated by Chapter 3 and 4, we aim to propose a theory of generalised timed multiparty session types to connect these two research branches. The syntax of global types and local types is presented, as well as the labelled transition system and projection rules. We define the semantics of timed choreographical session calculus and transition rules. Then the list of proof rules for validating the processes and the typing system for runtime processes are presented. We prove the theorems of subject reduction, type safety and time error-freedom. Also, we define the progress-enabling property and time-progress property for checking progress of timed choreographical processes.

6. Conclusion and Future Work

This thesis presents the theory of timed choreographical multiparty session types. Instead of formalising the theory directly, we take three steps. First, we propose the theory of timed session types, where time assertions are added to labelled message interactions and other syntaxes are as same as multiparty session types. Second, in choreographical session types, we introduce flexible fork/join and choice/merge operations in global and local types, and this introduction changes the semantics of primitive multiparty session types, except labelled message interactions. Finally, since time assertions only guard message interactions, we infer that time and parallelism are independent, and we can easily merge the semantics of timed session types and choreographical session types together. With these three steps, we realise the objective of this thesis.

In the last chapter, we start with summarising the key points in this thesis. Then we present several future work ideas of enriching the theory and implementing timed choreographical session types.

6.1. Conclusion

The π-calculus, a continuation of process calculus CCS, is a process algebra that uses message-passing to exchange channel names and is applied to a high-level description of concurrent mobile communicating systems, in which network configurations may change over time. As it costs time to pass messages from one endpoint to another, the asynchronous π-calculus is used to model such inherent asynchrony in distributed computation. Later it is observed that in concurrent systems, interactions are often accompanied by a significant amount of message exchanges with various types. Then *session types* were introduced as a formal framework in the polyadic π-calculus with type systems, so as to allow the typing of channel names by structured sequences of types and represent the trace of usage of the channels. Since some conversations require message changes between more than two participants, multiparty session types are proposed to describe

and verify behaviours in such systems by way of allowing one session to be owned by multiple participants. In recent years, several extensions appear to gain more expressivity and more computational properties, such as design-by-contract [BHTY10] and association with Communicating Finite State Machines [DY12].

In distributed systems, time-related specifications, such as deadline and timeout, could appear in protocol descriptions, by imposing restrictions on interactions to ensure the quality of the provided service. These specifications cannot be simply described by design-by-contract because time variables are special in the way that they are continuously increasing with time. This case is worse when the system recurs infinitely as time variables could grow extremely large without resetting their values. Thus, we propose the theory of timed multiparty session types to solve this problem.

Motivated by a distributed timed protocol, we add time assertions for send/receive interactions to the semantics of global types. A time assertion consists of time constraints for specifying if actions meet their requirements, and a reset predicate for setting the clock of participants to 0. The semantics for choice, End and recursion are the same as standard multiparty session types. With the definition of $ready()$ function to record the next time constraints of participants in the system, we describe the labelled transition system for timed global types to track the trace of actions and time passing. Similarly, we provide the semantics and transitions systems of timed local types, for analysing behaviours from local views, and we show that soundness and completeness of local types are preserved through projection.

Compared with standard session calculus, we add the delay primitive $delay().P$ to describe time elapse to achieve minimum extension. We also show the reduction rules for timed processes and structural congruence for $delay().P$ action. The proof rules are changed because we add a clock assignment, which records time values for all participants, to the session environment when validating the processes.

Furthermore, we investigate properties of timed session types: (1) we show and prove the theorems of type preservation under equivalence and subject reduction, which are inherited from multiparty session types; then (2) we present the theorem of time error-freedom, which indicates that if proof rules are conducted for all timed session types, then processes will not get stuck because of time error.

One challenging issue is that deadlock could be introduced by an inappropriate trace of time restrictions. Thus, we define two properties – feasibility and wait-

freedom – to prevent such inappropriate trace causing deadlock. We also formulate the theorem of session delay to ensure that processes still enjoy the progress property if timing part is removed from processes.

Parallel computation is a natural feature in distributed programming to improve performance and efficiency. If we want to model in the session types, one problem we face, as we increase the scale of programs, is the state explosion due to the lack of a parallel operation. As one of the techniques to reduce analysing states, subtyping cannot solve this problem. Therefore, we need to propose a generalised version of session types with parallel operations.

One way of evolving standard session types is to distinguish branching and parallel semantics. Thus, the syntax of global types contains a choice/merge operand and a fork/join operand for these two functionalities respectively. However, adding parallel operations can introduce new problems for well-formedness checking of global types, such as unmatchable numbers of fork operands and join operands. Besides the conventional properties inherited from standard multiparty session types, like sanity conditions and local choice, we also add thread correctness and linearity conditions to well-formedness checking for choreographical session types. In this way, the theory of well-formedness ensures the safety at the level of global types.

We apply similar changes to local types and give the projection algorithm for that syntax.

The session calculus is also extended with external/internal choices and fork/join transitions. The operational semantics for choreographical session calculus is provided to allow us to reduce the processes. Based on the operational semantics, we define the typing rules for state processes. Then the typing system is extended with run-time processes, in which message queues are included in the typing rules.

We find that typed multiparty session processes enjoy type safety property. With the lemma of subject congruence, we proved the subject reduction theorem; the network that is composed of typed processes is guaranteed to have no type error.

Now we have proposed two independent semantics for multiparty session types, and each one addresses independent problems. Another question, however, arises when we want to use session types to describe protocols that include *both* concurrent computing *and* time-related requirements. It is hard to model and verify such systems in two individual frameworks of session types; the algorithm for

feasibility and wait-freedom checking in timed session types cannot be applied to fork/join operations directly. In order to deal with the problem, we propose the theory of timed choreographical session types.

The global types are based on generalised session types, and each send/receive interaction is associated with a time assertion. Then we define the labelled transition system and global timed runs for timed choreographical session types. Similarly, time assertions are attached to local interactions and we prove the soundness and completeness preserved through projection. Compared with generalised processes, timed choreographical session calculus is simply an extension with a delay primitive. The time constraints and resets are included in operational semantics and typing systems.

We prove some theorems in a similar way as we prove those in timed session types and generalised session types, such as time error-freedom, because of orthogonality between time and parallel design. The algorithm for time progress-enabling check is different; we need to construct the roadmap with extinct parallel and choice representations first, and then find the dependency path for checking, based on this graph. Thus, well-typed and time progress-enabling processes are guaranteed to enjoy progress property.

Evaluation:

Comparing to other works, this is the first attempt to add time in multiparty session types and merge different variants together. In this way, expressiveness of multiparty session types are improved.

The proposed theory in this thesis leads to theoretical foundation for implementations. Based on timed multiparty session types, a verification framework with timed runtime monitors [NY17] are proposed and implemented in Scribble and Python language. Based on choreogrpahical session types, [LTY17] presents ChorGram, a tool of implementing the generalised multiparty compatibility and analysing message-passing programs. More implementable ideas are discussed later in Section 6.2.3.

6.2. Future Work

We discuss ideas for future work in three aspects. First, we present the possible subtyping and bisimulation theory for timed choreographical session types. Then

we discuss the potential of visualising the global types in Harel's Statecharts and BPMN choreographies. Second, we describe the idea of incorporating dependent types into multiparty session types so as to express data flow control. Last, we discuss possible implementation issues, like the Scribble expressions for time and choreographical extensions and monitor implementation.

6.2.1. Further Improvement on Timed Choreographical Session Types

Subtyping The formalism of [GH05] enables session types to describe protocols for richer behaviours. If the local types are 'upgraded', such from int to real number, without changing the global-level validation, run-time communication errors will occur because of type mismatch between send and receive actions. Subtyping solves this problem by allowing that a subset of types will not reduce to errors. In some sense, subtyping gives session types great flexibility to validate a large set of programs.

The work of applying subtyping to timed choreographical session types at least includes the following steps: 1) we define the subtyping rules for session types and typing rules for subtyped session types; 2) then we need to define and prove soundness, substitution, structural congruence, type safety and completeness [CDCY14]; 3) we convert typing rules to algorithmic subtyping rules and provide inference rules for the type checking algorithm to ensure that typable processes can proceed safely.

Bisimulation [Par81] is a relation between two transition systems that are observed to have same behaviours: for state transitions, we could find the same sequence of actions. This feature can be useful for validating large-scale systems [KY14]; if we could find a section of the large program is bisimilar with another previously validated well-typed session types, then we could treat this section as a safe super-state, and save efforts that need to be spent on verifying the scalable programs.

The preparation work for bisimulation of timed choreographical session types has been done: we define the labelled transition system for global types and local types, in which each label is associated with a local time assertion. Then we will define typed relation [Kou13] and typed barbs. And based on the definitions, we will provide the reduction-closed congruence and asynchronous

$$
\begin{array}{rcl}
\mathbf{G} & = & \mathsf{def} \\
\mathbf{x}_0 & = & \mathbf{x}_1 \mid \mathbf{x}_2 \\
\mathbf{x}_1 & = & \mathtt{A} \to \mathtt{B} : l_1\langle \mathsf{string} \rangle ; \mathbf{x}_3 \\
\mathbf{x}_2 & = & \mathbf{x}_4 + \mathbf{x}_5 \\
\mathbf{x}_4 & = & \mathtt{A} \to \mathtt{C} : l_2\langle \mathsf{nat} \rangle \{x_A \leq 3, x_C \leq 5\}; \mathbf{x}_6 \\
\mathbf{x}_5 & = & \mathtt{A} \to \mathtt{C} : l_3\langle \mathsf{nat} \rangle \{x_A > 3, x_C > 5\}; \mathbf{x}_7 \\
\mathbf{x}_6 + \mathbf{x}_7 & = & \mathbf{x}_8 \\
\mathbf{x}_3 \mid \mathbf{x}_8 & = & \mathbf{x}_9 \\
\mathbf{x}_9 & = & \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

Figure 6.1.: Global Types for the Example

timed choreographical session bisimulation.

Other visual formalisms that are used for displaying actions and state transitions can be potentially translated into session types in order to verify behaviour safety. Here we particular discuss the possibility of expressing Harel's Statecharts and BPMN choreographies in timed generalised session types.

- Harel's Statecharts, as an extension of state diagrams, are widely used in the design of event-driven systems, such as communicating protocols and distributed real-time systems. In this statechart, one event can trigger two components in the system to transition simultaneously. This feature is consistent with interactions in session types. Here is an example of how we could represent timed choreographical session types in Harel's statecharts.

  Inspired by the work of constructing global graph from CFSMs [LTY15], we could abstract communicating behaviours in Harel's Statecharts as timed choreographical session types and further validate if the system is deadlock-free and safe.

- BPMN, similar to flowcharts and UML activity diagrams, is a graphical language that describes end-to-end business processes and by hiding complex technical details, this formalism can be easily understood in all areas of business. The BPMN choreography models, specifies and analyses how participants exchange messages through their interactions. This feature addresses the same problem as session types do; thus we could translate the choreographies into timed choreographical session types to check if the target processes are safe or not.

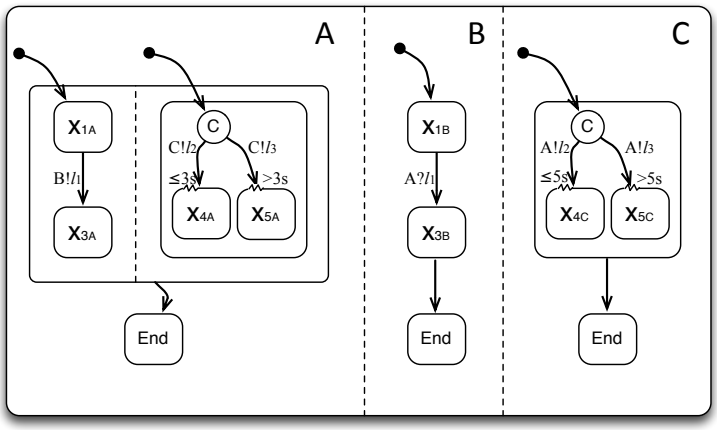


Figure 6.2.: Harel's Statecharts for Example 6.1

We will first identify the session types expression for each individual components in BPMN choreography. With these basic building blocks, we could decompose a choreography into separate choreographical tasks, connected with proper gateways. Then based on these pieces and logical relations between them, we could synthesise a global session type for further validation and code generation. Figure 6.3 shows an example that could be expressed in timed choreographical session types in Example 6.1.

6.2.2. Possible Extensions to Multiparty Session Types

The previous subsection describes the future improvements on timed choreographical session types. Here we enlarge the scope by discussing the extensions to multiparty session types in order to further enhance expressiveness.

When designing a communicating protocol, we sometimes find that only writing the communication interactions between participants is not enough; we may also want to annotate how data is changed during the whole communicating system, and the relations between the messages being sent and received. Then analysing the data flow of the protocol could solve the problem. The current version of session types, however, focuses on interactions between participants, so it can not provide sufficient information about data flow analysis. Since this analysis could be important in some systems, we need to extend session types with explicit data flow annotation.

In session types of design-by-contract, we have the assertions to guard the

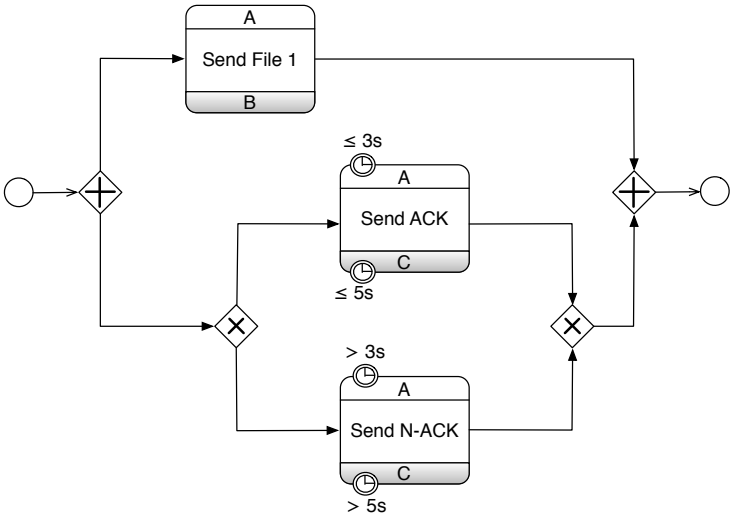


Figure 6.3.: BPMN choreography for Example 6.1

interactions. By contrast, in timed session types, we not only have time constraints, as a special type of assertions, but also have reset operations, assigning the time variable to zero. Based on this observation, this assignment operation could be integrated into session types. This integration could reflect one aspect of how data has been modified.

Another way of refining data flow is to find the relations between interaction messages. Dependent types could provide the solution to this problem.

Incrementor $= \mathtt{C} \to \mathtt{I} : \mathit{Send_num}\langle v : \mathsf{int}\rangle; \mathtt{I} \to \mathtt{C} : \mathit{Result}\langle v_1 : v+1 \to \mathsf{int}\rangle; \mathtt{end}$

This dependency helps us to reflect message relations; thus we gain better data flow control.

$$\begin{aligned} \textbf{Comparator} \quad = \quad & \mathtt{P} \to \mathtt{C} : \mathit{Send_num}_1\langle v_1 : \mathsf{int}\rangle\{\mathit{temp} \leftarrow v_1\}; \\ & \mathtt{P} \to \mathtt{C} : \mathit{Send_num}_2\langle v_2 : \mathsf{int}\rangle; \\ & \mathtt{C} \to \mathtt{P} : \mathit{Result}\langle v_3 : \{\mathit{temp}, v_2\} \to \mathsf{bool}\rangle; \mathtt{end} \end{aligned}$$

Sometimes dependent types are not enough to represent complex data transitions. For example, we send a table of historical data of weather temperatures to a cloud server and ask what is the possible degree Celsius of tomorrow's weather. The process of making this prediction is complex and we are unable to describe how the server makes the forecast. The general global view of this application is that we send a table and a number to the cloud server and the server replies a number back to us. Recall that our goal is to find the dependent relation between data in the communicating system. One simple but expressive way is to use

function/method expression.

$$\begin{aligned}\textbf{Forecast} \quad = \quad & \mathtt{C} \to \mathtt{S} : \mathit{History}\langle \mathit{hist} : \mathit{str} \rangle; \\ & \mathtt{C} \to \mathtt{S} : \mathit{Request}\langle \mathit{date} : \mathit{int} \rangle; \\ & \{\mathit{prediction} = f_{\mathit{regression}}(\mathit{hist}, \mathit{date})\}; \\ & \mathtt{S} \to \mathtt{C} : \mathit{Reply}\langle \mathit{prediction} : \mathit{float} \rangle; \mathtt{end}\end{aligned}$$

For the previous three extensions, we cluster the first two together and compare them with the last one based on the transparency of the message: data modification and data dependency are exposed in the expression of session types; the last extension hides details and it is easy to design internal transition functions and offer more security and satisfy easy-understanding considerations. We could adopt different specifications for different applications: for simple systems with public, transparent data, we could use the first two options; for systems with complex data processing and private data fields, we could apply the last option for data flow analysis.

6.2.3. Implementation

We will implement these timed choreographical session types in Scribble [Scr]. Scribble is a language for describing and validating protocols among communicating systems. Here is the procedure of how Scribble works: The protocol, an agreement which prescribes how participants in the communicating system should behave, is firstly written in the Scribble language; then a local description, according to a projection algorithm, is automatically produced from the global protocol. Based on the theory of session types, Scribble checks the properties of the protocol, such as deadlock-freedom and soundness. If the protocol is verified, then Scribble can generate the skeleton codes for each endpoint projection, with user specified languages. Also, monitors are associated with endpoint projection to monitor the activities of local programs in order to ensure that the endpoint implements the expected behaviours.

According to the procedure, we could also decompose the future work of implementing timed choreographical session types in Scribble into the following steps: design global/local expressions, build the verification modules, enlarge the library of code generation, and build the monitors for endpoints. Due to the independence between time and structure of global types, we could extend Scribble with timed session types first and then with choreographical session types separately.

```
global protocol SimpleConversation(role A, role B)
        [xA@A: xA<1 || xA>5][xb@B: reset(xB)]
        msg(string)from A to B;
        [xb@B:xB<3][]
        ack(boolean)from B to A;
```

Figure 6.4.: Scribble Example with Time Extension

Time Extension in Scribble has been proposed in [NBY14] and completed in [NY17]. Scribble is extended with time annotation illustrated in Figure6.4. Algorithms for checking feasibility and wait-freedom are implemented. The example programs and timed runtime monitors are presented in Python. In addition, the monitors are responsible for verifying message types, synchronisation between clocks and triggering exceptions when violation occurs.

Further improvements could be to evaluate the robustness of communications and dynamically adjust parameters at runtime. Robustness [BMS13] in timed automata is defined as the ability of a system to resist slight perturbations or time errors. These variations are usually caused by delays in communication channels, imprecise clocks, or longer execution time than expected. Due to the unpredictable variations during runtime execution, we will need to measure some parameters, such as Round-Trip Time (RTT), and make adjustments according to the analysis of tolerance to time error. For example, we could measure the RTT as t_{RTT}; then we may require the timed constraints to shrink as $t_P < m - t_{RTT}$, where m is predefined upper bound for that constraint. By adjusting the runtime assertion, we could guarantee the performance of the protocol.

Choreography Extension of multiparty session types is non-trivial. In generalised session types, we use auxiliary $\mathbf{x}$ to annotate states for describing state transitions. Although choreographical session types introduce great flexibility, they bring complexity in designing the Scribble language; while we can easily write nested expressions, it is difficult to describe the interleaved expressions, such as the example in Figure 6.5. So the implementation of fork/merge will inevitably make many more changes to the Scribble language than timed session types do.

We propose two possible ways for implementing choreographical session types in Scribble. One is to introduce a state variable x to note interactions and relations between them. Then we could write Scribble in the same way as session types. The disadvantage is that the Scribble language will be changed greatly: recursion

$$
\begin{array}{rcl}
\mathbf{G} & = & \mathsf{def} \\
\mathbf{x}_0 & = & \mathbf{x}_1 \mid \mathbf{x}_2 \\
\mathbf{x}_1 & = & \mathtt{A} \rightarrow \mathtt{B} : \mathit{Msg}_1\langle \mathsf{string} \rangle ; \mathbf{x}_3 \\
\mathbf{x}_2 & = & \mathbf{x}_4 \mid \mathbf{x}_5 \\
\mathbf{x}_4 & = & \mathtt{B} \rightarrow \mathtt{A} : \mathit{Msg}_2\langle \mathsf{nat} \rangle ; \mathbf{x}_6 \\
\mathbf{x}_5 & = & \mathtt{B} \rightarrow \mathtt{C} : \mathit{Msg}_3\langle \mathsf{nat} \rangle ; \mathbf{x}_7 \\
\mathbf{x}_3 \mid \mathbf{x}_6 & = & \mathbf{x}_8 \\
\mathbf{x}_8 & = & \mathtt{A} \rightarrow \mathtt{C} : \mathit{Msg}_4\langle \mathsf{string} \rangle ; \mathbf{x}_9 \\
\mathbf{x}_7 \mid \mathbf{x}_9 & = & \mathbf{x}_{10} \\
\mathbf{x}_{10} & = & \ldots \quad \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

Figure 6.5.: An Example of Choreographical Timed Global Session Types with Interleaved Join/Merge

```
global protocol SimpleFork(role A, role B, role C){
        state  x[11];
        x[0]:  x[1] Par x[2];
        x[1]:  Msg1(string)from A to B then x[3];
        x[2]:  x[4] Par x[5];
        x[4]:  Msg2(nat)from B to A then x[6];
        x[5]:  Msg3(nat)from B to C then x[7];
        x[3] Par x[6] :  x[8];
        x[8]:  Msg4(string)from A to C then x[9];
        x[7] Par x[9] :  x[10];
        x[10]:  ......
}
```

Figure 6.6.: Illustration of Scribble Extension with State Variable x

are removed and a large set of state variables are introduced.

Another way is that we only note the interactions and use the *goto* keyword to jump between interleaved session types. Figure 6.7 illustrates how this keyword could work on the example. The weakness of this method is less explicit than the first one because the person writing Scribble code should know the structural relations between interactions very clearly and may need such auxiliary BPMN-like graph notations.

Moreover, in verification, we need to implement well-formedness checking and time progress-enabling checking. At runtime, the monitors may hold an action checklist to note which actions complete. This checklist is used to determine if the interactions following a merge operation could happen or not.

```
global protocol SimpleFork(role A, role B, role C){
        transition  T13,T46,T57,T89;
        join  J1,J2;
        PAR {
                Msg1(string)from A to B as T13 goto J1;
                Msg2(nat)from B to A as T46 goto J1;
                Msg3(nat)from B to C as T57 goto J2;
                J1:  [pre (T13 & T46)]
                Msg4(string)from A to C as T89 goto J2;
                J2:  [pre (T57 & T89)]
                ......
        }
}
```

Figure 6.7.: Illustration of Scribble Extension with *goto*

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

[AFK87] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in distributed languages. In *POPL*, pages 189–198. ACM, 1987.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1 – 70, 1999.

[AOB14] Samuel Aaron, Dominic Orchard, and Alan F. Blackwell. Temporal semantics for a live coding language. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling ; Design*, Gothenburg, Sweden, 2014.

[BBO12a] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In *POPL '12*, pages 191–202, Philadelphia, PA, USA, 2012. ACM.

[BBO12b] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchonously communicating systems. In *VMCAI'12*, LNCS. Springer, 2012.

[BC08] Eduardo Bonelli and Adriana Compagnoni. Multipoint session types for a distributed calculus. In *TGC07*, volume 4912 of *LNCS*, pages 240–256. Springer, 2008.

[BHTY10] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176, 2010.

[BLY15] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together. In *CONCUR 2015*, volume 42 of *LIPIcs*, pages 283–296, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BMS13] Patricia Bouyer, Nicolas Markey, and Ocan Sankur. Robustness in timed automata. In *RP '13*, volume 8169 of *LNCS*, pages 1–18, Uppsala, Sweden, September 2013. Springer.

[BPM] Business process model and notation. `http://www.bpmn.org/`.

[BY07] Martin Berger and Nobuko Yoshida. Timed, distributed, probabilistic, typed processes. In *Programming Languages and Systems*, volume 4807 of *LNCS*, pages 158–174. Springer Berlin Heidelberg, 2007.

[BYY14] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed Multiparty Session Types. In *CONCUR*, volume 8704 of *LNCS*. Springer, 2014.

[Cai04] Luís Caires. Behavioral and spatial observations in a logic for the π-calculus. In *FOSSACS*, pages 72–89. Springer, 2004.

[CDCP11] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party sessions. In *FMOODS/FORTE*, volume 6722 of *LNCS*, pages 1–28, 2011.

[CDCY14] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *PPDP 2014*, pages 135–146. ACM, 2014.

[CDCYP15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 760, 2015.

[CNP01] David G. Clarke, James Noble, and John M. Potter. *Simple Ownership Types for Object Containment*, chapter ECOOP 2001, pages 53–76. Springer, Berlin, Heidelberg, 2001.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, January 1993.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011.

[DY12] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213, 2012.

[DY13] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186, 2013.

[FFW+04] Marco Fischer, Stefan Förster, Andé Windisch, Dieter Monjau, and Burkhard Balser. A new time extension to π-calculus based on time consuming transition semantics. In *Languages for System Specification*, pages 271–283. 2004.

[FHY14] Luca Fossati, Raymond Hu, and Nobuko Yoshida. *Multiparty Session Nets*, pages 112–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[GH05] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.

[GMP04] Blaise Genest, Anca Muscholl, and Doron Peled. Message sequence charts. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 537–558, 2004.

[GVR+10] Simon J. Gay, Vasco T. Vasconcelos, Antonio Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular Session Types for Distributed Object-Oriented Programming. In *POPL*, 2010. To appear.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[HVK] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP'98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[HYH08] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[Kou13] Dimitrios Kouzapas. *A Study of Bisimulation Theory for Session Types*. PhD thesis, Imperial College London, 2013.

[KY06] Pavel Krcal and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In *CAV*, LNCS, 2006.

[KY14] Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. *Logical Methods in Computer Science*, 10(4), 2014.

[LOP09] Hugo A. López, Carlos Olarte, and Jorge A. Pérez. Towards a unified framework for declarative structured communications. In *PLACES*, volume 17 of *EPTCS*, pages 1–15, 2009.

[LP11] Hugo A López and Jorge A Pérez. The timed, compensable conversation calculus. *Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2011.

[LP12] Hugo A. López and Jorge A. Pérez. Time and exceptional behavior in multiparty structured interactions. In *WS-FM*, volume 7176 of *LNCS*, pages 48–63. 2012.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Cows: A timed service-oriented calculus. In *ICTAC 2007*, volume 4711 of *LNCS*, pages 275–290. Springer Berlin Heidelberg, 2007.

[LTY15] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232. ACM, 2015.

[LTY17] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. *A tool for choreography-based analysis of message-passing software*. River Publishers, 2017.

[LZ02] Jeremy Y. Lee and John Zic. On modeling real-time mobile processes. In *Proceedings of the Twenty-fifth Australasian Conference on Computer Science - Volume 4*, pages 139–147. Australian Computer Society, Inc., 2002.

[LZ05] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. pages 282–298. Springer, 2005.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil91] Robin Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[MY07] Dimitris Mostrous and Nobuko Yoshida. *TLCA 2007*, chapter Two Session Typing Systems for Higher-Order Mobile Processes, pages 321–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[MY09] Dimitris Mostrous and Nobuko Yoshida. Tlca '09. pages 203–218, Berlin, Heidelberg, 2009. Springer-Verlag.

[NBY14] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT 2014*, pages 19–26, 2014.

[NCY15] Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction*, volume 9031 of *LNCS*. Springer, 2015.

[NT04] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[NY17] Rumyana Neykova and Nobuko Yoshida. Timed Runtime Monitoring for Multiparty Conversations. *Formal Aspects of Computing*, 2017. To appear.

[NYH12] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. *Multiparty Session C: Safe Parallel Programming with Message Optimisation*, pages 202–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[OOI] Ocean Observatories Initiative (OOI). `http://oceanobservatories.org/`.

[OV08] Carlos Olarte and Frank D. Valencia. Universal Concurrent Constraint Programing: Symbolic Semantics and Applications to Security. In *23rd Annual ACM Symposium on Applied Computing*, Fortaleza, Brésil, 2008. ACM.

[Par81] David Park. *Theoretical Computer Science: 5th GI-Conference Karlsruhe, March 23–25, 1981*, chapter Concurrency and automata on infinite sequences, pages 167–183. Springer, Berlin, Heidelberg, 1981.

[Puh06] Frank Puhlmann. Why do we actually need the pi-calculus for business process management? In *Business Information Systems, 9th International Conference on Business Information Systems, BIS 2006, May 31 - June 2, 2006, Klagenfurt, Austria*, pages 77–89, 2006.

[RR98] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, London, UK, UK, 1998. Springer-Verlag.

[RS11] Mark D Ryan and Ben Smyth. Applied pi calculus. 2011.

[Scr] Scribble Project homepage. `www.scribble.org`.

[son] Sonic pi. http://sonic-pi.net/.

[SW01a] Davide Sangiorgi and David Walker. *On Barbed Equivalences in π-Calculus*, chapter CONCUR 2001 — Concurrency Theory: 12th International Conference Aalborg, Denmark, August 20–25, 2001 Proceedings, pages 292–304. Springer, Berlin, Heidelberg, 2001.

[SW01b] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *In PARLE'94, volume 817 of LNCS*, pages 398–413. Springer-Verlag, 1994.

[Tri99] Stavros Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 299–314. 1999.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52 – 70, 2012.

[VCS08] Hugo T. Vieira, Luís Caires, and João C. Seco. The conversation calculus: a model of service oriented computation. In *Proc. of ESOP'08, LNCS*. Springer, 2008.

[VGR06] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.

[Vie10] Hugo Filipe Mendes Torres Vieira. *A calculus for modeling and analyzing conversations in service-oriented computing*. PhD thesis, Faculdade de Ciências e Tecnologia, 2010.

[YHNN13] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC 2013*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.

[YV07] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, July 2007.

[YVPH09] Nobuko Yoshida, Vasco Vasconcelos, Hervé Paulino, and Kohei Honda. Session-based compilation framework for multicore programming. In *FMCO*, volume 5751 of *LNCS*, pages 226–246. 2009.

A. Supporting Materials for Chapter 4

A.1. Appendix for Section 4.2.1

Additional Global Type Examples

In Figure A.1, we give several additional global type examples that illustrate some particular features that were not developed in the main sections.

1. A simple one-message (*Msg* of type nat) is exchanged between `Alice` and `Bob`.Figure
2. A protocol with a simple choice between messages *Book* and *Film*.
3. `Alice` and `Bob` concurrently exchange the messages *Book* and *Film*.
4. $\mathbf{G}_4$ features an initial choice between directly contacting `Carol` or to do it through `Bob`. Note that without the last interaction from `Carol` to `Bob` (in $\mathbf{x}_6$), if the chosen path leads to $\mathbf{x}_3$, `Bob` enters a deadlock, waiting forever for a message from `Alice`.
5. This global type represent the extension of the Alternating-Bit protocol to three messages. As we can see the global type only linearly increases in size, while the CFSM it represents double its number of states.

Local Choice Condition

We list here some omitted definitions and examples from the main section.

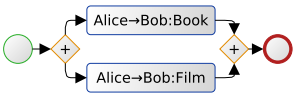
Active Sender The active sender computation is not as much a counting algorithm as a verification method that checks whether a given choice has a unique (i.e. local) responsible participant. In particular, the deduction rules gather the active senders on each branch and restrict it to be a singleton `p`.

1. $\mathbf{G}_1 = \text{def } \mathbf{x}_0 = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}\langle \text{nat} \rangle ; \mathbf{x}_1$
$\mathbf{x}_1 = \text{end in } \mathbf{x}_0$



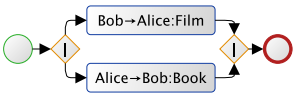
---

2. $\mathbf{G}_2 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2$
$\mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Book}\langle \text{string} \rangle ; \mathbf{x}_3$
$\mathbf{x}_2 = \texttt{Alice} \to \texttt{Bob} : \mathit{Film}\langle \text{string} \rangle ; \mathbf{x}_4$
$\mathbf{x}_3 + \mathbf{x}_4 = \mathbf{x}_5$
$\mathbf{x}_5 = \text{end in } \mathbf{x}_0$



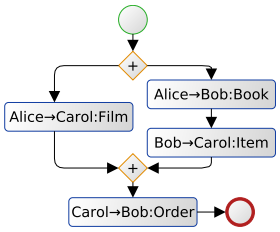
---

3. $\mathbf{G}_3 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$
$\mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Book}\langle \text{string} \rangle ; \mathbf{x}_3$
$\mathbf{x}_2 = \texttt{Bob} \to \texttt{Alice} : \mathit{Film}\langle \text{string} \rangle ; \mathbf{x}_4$
$\mathbf{x}_3 \mid \mathbf{x}_4 = \mathbf{x}_5$
$\mathbf{x}_5 = \text{end in } \mathbf{x}_0$



---

4. $\mathbf{G}_4 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_3$
$\mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Book}\langle \text{string} \rangle ; \mathbf{x}_2$
$\mathbf{x}_2 = \texttt{Bob} \to \texttt{Carol} : \mathit{Item}\langle \text{nat} \rangle ; \mathbf{x}_4$
$\mathbf{x}_3 = \texttt{Alice} \to \texttt{Carol} : \mathit{Film}\langle \text{string} \rangle ; \mathbf{x}_5$
$\mathbf{x}_4 + \mathbf{x}_5 = \mathbf{x}_6$
$\mathbf{x}_6 = \texttt{Carol} \to \texttt{Bob} : \mathit{Order}\langle \text{string} \rangle ; \mathbf{x}_7$
$\mathbf{x}_7 = \text{end in } \mathbf{x}_0$



---

5. $\mathbf{G}_5 = \text{def } \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$
$\mathbf{x}_1 = \mathbf{x}_3 \mid \mathbf{x}_4$
$\mathbf{x}_2 + \mathbf{x}_5 = \mathbf{x}_6$
$\mathbf{x}_3 + \mathbf{x}_7 = \mathbf{x}_8$
$\mathbf{x}_4 + \mathbf{x}_9 = \mathbf{x}_{10}$
$\mathbf{x}_6 = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_1\langle \text{string} \rangle ; \mathbf{x}_{11}$
$\mathbf{x}_{11} = \mathbf{x}_{12} \mid \mathbf{x}_{13}$
$\mathbf{x}_{13} = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_1\langle \text{unit} \rangle ; \mathbf{x}_{14}$
$\mathbf{x}_{12} \mid \mathbf{x}_8 = \mathbf{x}_{15}$
$\mathbf{x}_{15} = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_2\langle \text{string} \rangle ; \mathbf{x}_{16}$
$\mathbf{x}_{16} = \mathbf{x}_{17} \mid \mathbf{x}_{18}$
$\mathbf{x}_{17} = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_2\langle \text{unit} \rangle ; \mathbf{x}_7$
$\mathbf{x}_{18} \mid \mathbf{x}_{10} = \mathbf{x}_{19}$
$\mathbf{x}_{19} = \texttt{Alice} \to \texttt{Bob} : \mathit{Msg}_3\langle \text{string} \rangle ; \mathbf{x}_{20}$
$\mathbf{x}_{20} = \mathbf{x}_{21} \mid \mathbf{x}_{22}$
$\mathbf{x}_{21} = \texttt{Bob} \to \texttt{Alice} : \mathit{Ack}_3\langle \text{unit} \rangle ; \mathbf{x}_9$
$\mathbf{x}_{22} \mid \mathbf{x}_{14} = \mathbf{x}_5 \text{ in } \mathbf{x}_0$

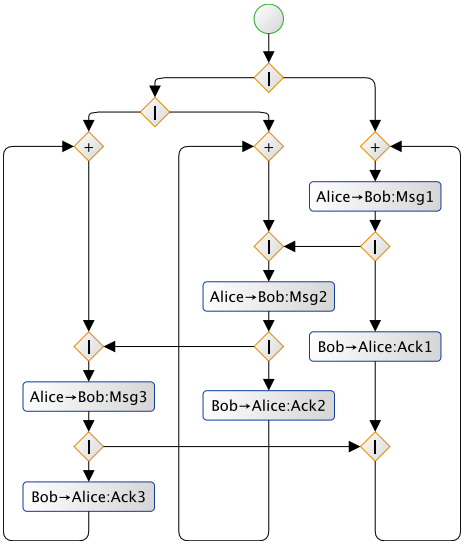


Figure A.1.: Examples of Global Types (2)

$$\frac{\mathbf{x} = \mathsf{end} \in \widetilde{G}}{\widetilde{G} \vdash \mathbf{x} : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x} : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2} \qquad \frac{\mathbf{x}_1 = \mathsf{end} \in \widetilde{G} \quad \mathbf{x}_2 \in \tilde{\mathbf{x}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 \in \tilde{\mathbf{x}}_1 \quad \mathbf{x}_2 \in \tilde{\mathbf{x}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2} \qquad \frac{\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x} \in \widetilde{G}}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{(\mathbf{x}_1 \mid \mathbf{x} = \mathbf{x}' \in \widetilde{G} \vee \mathbf{x} \mid \mathbf{x}_1 = \mathbf{x}' \in \widetilde{G}) \quad \widetilde{G} \vdash \mathbf{x}' : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{(\mathbf{x}_1 + \mathbf{x} = \mathbf{x}' \in \widetilde{G} \vee \mathbf{x} + \mathbf{x}_1 = \mathbf{x}' \in \widetilde{G}) \quad \widetilde{G} \vdash \mathbf{x}' : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 = \mathbf{x} \mid \mathbf{x}' \in \widetilde{G} \quad \widetilde{G} \vdash \mathbf{x} : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2 \quad \widetilde{G} \vdash \mathbf{x}' : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 = \mathbf{x} + \mathbf{x}' \in \widetilde{G} \quad \widetilde{G} \vdash \mathbf{x} : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2 \quad \widetilde{G} \vdash \mathbf{x}' : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 = \mathrm{p} \to \mathrm{p}' : l\langle U \rangle; \mathbf{x}_1' \in \widetilde{G} \quad \mathrm{p} \in \tilde{\mathrm{p}}_1 \quad \widetilde{G} \vdash \mathbf{x}_1' : \mathbf{x}_1'\tilde{\mathbf{x}}_1, \mathrm{p}'\tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\widetilde{G} \vdash \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2 \between \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \between \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 \in \widetilde{G} \quad \widetilde{G} \vdash \mathbf{x}_1 : \emptyset, \mathrm{p} \between \mathbf{x}_2 : \emptyset, \mathrm{p}}{\mathit{ASend}(\widetilde{G})(\mathbf{x}) = \mathrm{p}}$$

Figure A.2.: Active Sender Computation

Local Choice Example (Local choice) is essential for the consistency of the distributed interaction with respect to choice.

Consider the following incorrect global type:

$$\begin{aligned}
\mathbf{G}_{\neg\text{ch.a.}} = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 + \mathbf{x}_3 \\
\mathbf{x}_1 &= \mathtt{Alice} \to \mathtt{Bob} : \mathit{Item}\langle \mathsf{string} \rangle; \mathbf{x}_2 \\
\mathbf{x}_2 &= \mathtt{Bob} \to \mathtt{Dave} : \mathit{Book}\langle \mathsf{string} \rangle; \mathbf{x}_4 \\
\mathbf{x}_3 &= \mathtt{Carol} \to \mathtt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle; \mathbf{x}_5 \\
\mathbf{x}_4 + \mathbf{x}_5 &= \mathbf{x}_6 \\
\mathbf{x}_6 &= \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0
\end{aligned}$$

This global type is incorrect because both `Alice` and `Carol` are supposed to make a local choice between *Item* and *Film* (prevented by $\mathit{ASend}(\widetilde{G}_{\neg\text{ch.a.}})(\mathbf{x}_0)$). Even if `Alice` were to send message *Film*, it forgets to inform `Dave` of the choice, which may result in him waiting forever for a message *Book* that will never come (it is prevented by the condition $\mathit{Rcv}(\widetilde{G}_{\neg\text{ch.a.}})(\mathbf{x}_1) \neq \mathit{Rcv}(\widetilde{G}_{\neg\text{ch.a.}})(\mathbf{x}_3)$). Note that $\mathbf{G}_5$ satisfies the local choice property.

The following incorrect examples illustrates how local choice can also prevents two identical labels from being in the two branches of the same choice (it confuses

Bob):

$$\begin{array}{rcl}
\mathbf{G}_{\neg\text{lin}} = \mathsf{def}\ \mathbf{x}_0 & = & \mathbf{x}_1 + \mathbf{x}_3 \\
\mathbf{x}_1 & = & \mathtt{Alice} \to \mathtt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle ; \mathbf{x}_2 \\
\mathbf{x}_2 & = & \mathtt{Bob} \to \mathtt{Alice} : \mathit{Book}\langle \mathsf{string} \rangle ; \mathbf{x}_4 \\
\mathbf{x}_3 & = & \mathtt{Alice} \to \mathtt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle ; \mathbf{x}_5 \\
\mathbf{x}_4 + \mathbf{x}_5 & = & \mathbf{x}_6 \\
\mathbf{x}_6 & = & \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0
\end{array}$$

A.2. Appendix for Section 4.4

New Name Creation The following P_1 corresponds to $\mu X.x[\mathtt{p}](y).(\nu a)y!\langle \mathtt{p}, l\langle a \rangle \rangle.X$ where $\mu X.P$ represents the standard recursion. This agent sends a fresh name whenever asked.

$$\begin{array}{rcl}
P_1 = \mathsf{def}\ \mathbf{x}_0(x) & = & x[\mathtt{p}](y).\mathbf{x}_1(x,y) \\
\mathbf{x}_1(x,y) & = & (\nu a)\mathbf{x}_2(x,y,a) \\
\mathbf{x}_2(x,y,z) & = & y!\langle \mathtt{p}, l\langle z \rangle \rangle.\mathbf{x}_0(x)\ \mathsf{in}\ \mathbf{x}_0(b)
\end{array}$$

The following P_2 represents $(\nu a)\mu X.x[\mathtt{p}](y).y!\langle \mathtt{p}, l\langle a \rangle \rangle.X$ which sends the same name each time.

$$\begin{array}{rcl}
P_2 = (\nu a)\mathsf{def}\ \mathbf{x}_0(x,z) & = & z[\mathtt{p}](y).\mathbf{x}_1(x,z,y) \\
\mathbf{x}_1(x,z,y) & = & y!\langle \mathtt{p}, l\langle x \rangle \rangle.\mathbf{x}_0(x,z)\ \mathsf{in}\ \mathbf{x}_0(a,b)
\end{array}$$

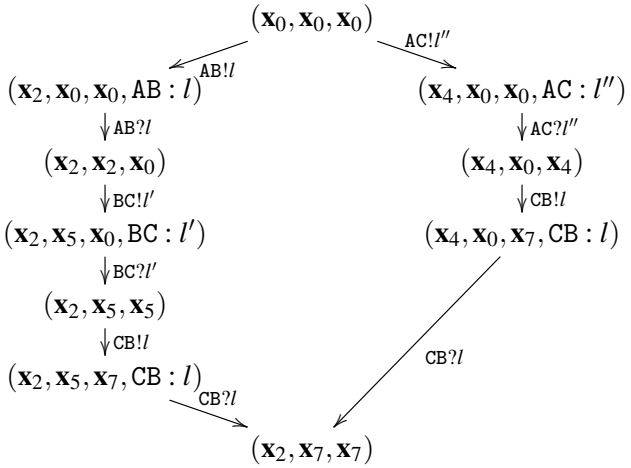
A.3. Examples for global types and well-formed inference

This appendix lists examples of global types, thread collection rules, local types, and automata.

Example 1. This example corresponds to $\mathbf{G}_4$ in example A.1.

$$
\begin{array}{lrcl}
\mathbf{G}_4 = \mathsf{def} & \mathbf{x}_0 & = & \mathbf{x}_1 + \mathbf{x}_3 \\
& \mathbf{x}_1 & = & \mathtt{A} \to \mathtt{B} : l\langle U \rangle ; \mathbf{x}_2 \\
& \mathbf{x}_2 & = & \mathtt{B} \to \mathtt{C} : l'\langle U' \rangle ; \mathbf{x}_5 \\
& \mathbf{x}_3 & = & \mathtt{A} \to \mathtt{C} : l''\langle U'' \rangle ; \mathbf{x}_4 \\
& \mathbf{x}_4 + \mathbf{x}_5 & = & \mathbf{x}_6 \\
& \mathbf{x}_6 & = & \mathtt{C} \to \mathtt{B} : l\langle U \rangle ; \mathbf{x}_7 \\
& \mathbf{x}_7 & = & \mathsf{end}
\end{array}
$$

Automaton (we omit empty channels).



For A, we have $\mathbf{x}_0 \equiv \mathbf{x}_1 \equiv \mathbf{x}_3$ and $\mathbf{x}_2 \equiv \mathbf{x}_4 \equiv \mathbf{x}_5 \equiv \mathbf{x}_6 \equiv \mathbf{x}_7$.
For B, we have $\mathbf{x}_0 \equiv \mathbf{x}_1 \equiv \mathbf{x}_3 \equiv \mathbf{x}_4 \equiv \mathbf{x}_5 \equiv \mathbf{x}_6$.
For C, we have $\mathbf{x}_0 \equiv \mathbf{x}_1 \equiv \mathbf{x}_2 \equiv \mathbf{x}_3$ and $\mathbf{x}_4 \equiv \mathbf{x}_5 \equiv \mathbf{x}_6$.

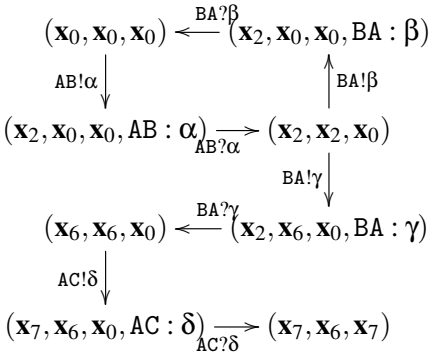
Local types.

$$
\begin{array}{lrcl}
\mathcal{T}_A = \mathsf{def} & \mathbf{x}_0 & = & \mathbf{x}_1 \oplus \mathbf{x}_3 \\
& \mathbf{x}_1 & = & !\langle \mathtt{B}, l\langle U\rangle\rangle.\mathbf{x}_5 \\
& \mathbf{x}_3 & = & !\langle \mathtt{C}, l''\langle U\rangle\rangle.\mathbf{x}_4 \\
& \mathbf{x}_4 + \mathbf{x}_5 & = & \mathbf{x}_7 \\
& \mathbf{x}_7 & = & \mathsf{end} \\
\mathcal{T}_B = \mathsf{def} & \mathbf{x}_0 & = & \mathbf{x}_1 \,\&\, \mathbf{x}_4 \\
& \mathbf{x}_1 & = & ?\langle \mathtt{A}, l\langle U\rangle\rangle.\mathbf{x}_2 \\
& \mathbf{x}_2 & = & !\langle \mathtt{C}, l'\langle U'\rangle\rangle.\mathbf{x}_5 \\
& \mathbf{x}_4 + \mathbf{x}_5 & = & \mathbf{x}_6 \\
& \mathbf{x}_6 & = & ?\langle \mathtt{C}, l\langle U\rangle\rangle.\mathbf{x}_7 \\
& \mathbf{x}_7 & = & \mathsf{end} \\
\mathcal{T}_C = \mathsf{def} & \mathbf{x}_0 & = & \mathbf{x}_2 \,\&\, \mathbf{x}_3 \\
& \mathbf{x}_2 & = & ?\langle \mathtt{B}, l'\langle U'\rangle\rangle.\mathbf{x}_5 \\
& \mathbf{x}_3 & = & ?\langle \mathtt{B}, l''\langle U''\rangle\rangle.\mathbf{x}_4 \\
& \mathbf{x}_4 + \mathbf{x}_5 & = & \mathbf{x}_6 \\
& \mathbf{x}_6 & = & !\langle \mathtt{B}, l\langle U\rangle\rangle.\mathbf{x}_7 \\
& \mathbf{x}_7 & = & \mathsf{end}
\end{array}
$$

Example 2. Global type.

$$
\begin{array}{lrcll}
\mathbf{G}_7 = \mathsf{def} & \mathbf{x}_0 + \mathbf{x}_4 & = & \mathbf{x}_1 & \\
& \mathbf{x}_1 & = & \mathtt{A} \to \mathtt{B} : \alpha\langle U\rangle; \mathbf{x}_2 & \\
& \mathbf{x}_2 & = & \mathbf{x}_3 + \mathbf{x}_5 & \\
& \mathbf{x}_3 & = & \mathtt{B} \to \mathtt{A} : \beta\langle U\rangle; \mathbf{x}_4 & \\
& \mathbf{x}_5 & = & \mathtt{B} \to \mathtt{A} : \gamma\langle U\rangle; \mathbf{x}_6 & \\
& \mathbf{x}_6 & = & \mathtt{A} \to \mathtt{C} : \delta\langle U\rangle; \mathbf{x}_7 & \\
& \mathbf{x}_7 & = & \mathsf{end} & \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

Automaton.



Example 3. Global type syntax based on a syntax trees, even with recursion, is not enough to represent concurrent, yet not independent, interactions, we present here an example that illustrates that situation. The alternating bit protocol is an example that cannot be written using a syntax tree. An informal argument is that, in syntax trees, join points are always constrained by the well-parenthesised characteristics of fork points.

$$
\begin{array}{rrcll}
\mathbf{G}_9 = \mathsf{def} & \mathbf{x}_0 & = & \mathbf{x}_1 \mid \mathbf{x}_2 & \\
& \mathbf{x}_1 & = & \mathtt{A} \to \mathtt{B} : \alpha\langle U \rangle ; \mathbf{x}_3 & \\
& \mathbf{x}_2 & = & \mathtt{A} \to \mathtt{B} : \beta\langle U \rangle ; \mathbf{x}_4 & \\
& \mathbf{x}_3 & = & \mathbf{x}_5 \mid \mathbf{x}_6 & \\
& \mathbf{x}_4 \mid \mathbf{x}_5 & = & \mathbf{x}_7 & \\
& \mathbf{x}_6 & = & \mathtt{A} \to \mathtt{B} : \gamma\langle U \rangle ; \mathbf{x}_8 & \\
& \mathbf{x}_7 & = & \mathtt{A} \to \mathtt{B} : \delta\langle U \rangle ; \mathbf{x}_9 & \\
& \mathbf{x}_8 \mid \mathbf{x}_9 & = & \mathbf{x}_{10} & \\
& \mathbf{x}_{10} & = & \mathsf{end} & \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

The possible traces are αγβδ, βαγδ,αβγδ, βαδγ,αβδγ.

It's easy to write with a standard syntax a representation for the relation between messages αβδ, but γ should always follow α without being linked with β or δ. Only a graph-like syntax can represent that set of traces.

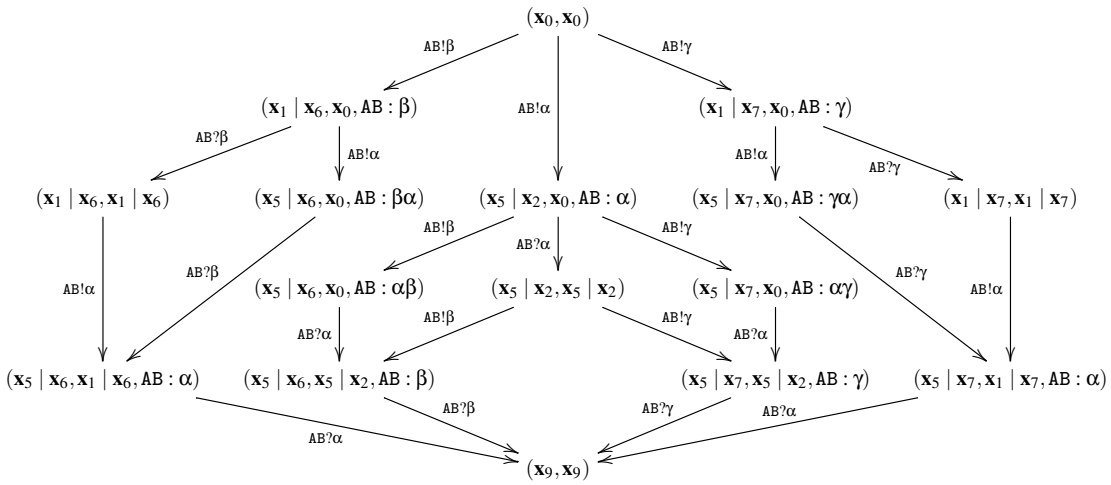


Figure A.3.: Example 4: global state machine

Example 4.

$$
\begin{aligned}
\mathbf{G}_8 = \mathsf{def} \qquad \mathbf{x}_0 &= \mathbf{x}_1 \mid \mathbf{x}_2 \\
\mathbf{x}_1 &= \mathtt{A} \to \mathtt{B} : \alpha\langle U \rangle ; \mathbf{x}_5 \\
\mathbf{x}_2 &= \mathbf{x}_3 + \mathbf{x}_4 \\
\mathbf{x}_3 &= \mathtt{A} \to \mathtt{B} : \beta\langle U \rangle ; \mathbf{x}_6 \\
\mathbf{x}_4 &= \mathtt{A} \to \mathtt{B} : \gamma\langle U \rangle ; \mathbf{x}_7 \\
\mathbf{x}_6 + \mathbf{x}_7 &= \mathbf{x}_8 \\
\mathbf{x}_5 \mid \mathbf{x}_8 &= \mathbf{x}_9 \\
\mathbf{x}_9 &= \mathsf{end} \qquad\qquad \mathsf{in}\ \mathbf{x}_0
\end{aligned}
$$

We give the global state automaton in Figure A.3.

For $\mathtt{A}$, we have $\mathbf{x}_0 \equiv \mathbf{x}_1 \mid \mathbf{x}_2 \equiv \mathbf{x}_1 \mid \mathbf{x}_3 \equiv \mathbf{x}_1 \mid \mathbf{x}_4$ and $\mathbf{x}_5 \mid \mathbf{x}_2 \equiv \mathbf{x}_5 \mid \mathbf{x}_3 \equiv \mathbf{x}_5 \mid \mathbf{x}_4$ and $\mathbf{x}_1 \mid \mathbf{x}_6 \equiv \mathbf{x}_1 \mid \mathbf{x}_7 \equiv \mathbf{x}_1 \mid \mathbf{x}_8$ and $\mathbf{x}_5 \mid \mathbf{x}_6 \equiv \mathbf{x}_5 \mid \mathbf{x}_7 \equiv \mathbf{x}_5 \mid \mathbf{x}_8 \equiv \mathbf{x}_9$.

For $\mathtt{B}$, we have $\mathbf{x}_0 \equiv \mathbf{x}_1 \mid \mathbf{x}_2 \equiv \mathbf{x}_1 \mid \mathbf{x}_3 \equiv \mathbf{x}_1 \mid \mathbf{x}_4$ and $\mathbf{x}_5 \mid \mathbf{x}_6 \equiv \mathbf{x}_5 \mid \mathbf{x}_7 \equiv \mathbf{x}_5 \mid \mathbf{x}_8 \equiv \mathbf{x}_9$.