AN EQUATIONAL LOGIC AND A COORDINATION LANGUAGE FOR DISTRIBUTED OBJECTS

BY

MAHMOOD REZA ZIAEI

B.S., Shahid Beheshti University, 1993
M.S., Sharif University of Technology, 1996

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois
Abstract

Building reliable distributed object-based systems is challenging. The work presented in this dissertation investigates two approaches to address some of these challenges: logical and linguistic.

The first part of the dissertation presents an equational theory of may testing equivalence for asynchronous calculi with locality and no name matching. Name matching, which is similar to pointer comparison in imperative languages, is considered a harmful feature that prevents useful program transformations. A trace-based characterization of may testing is provided for a version of asynchronous \( \pi \)-calculus with locality and no name matching (called L\( \pi \)). Trace-based characterizations simplify reasoning about equivalence of pairs of processes. Using the characterization, a complete axiomatization for the finitary fragment of the calculi is presented.

The first part of the dissertation extends the may testing results to the Actor model of distributed object-based computation. The Actor model has been an inspiring model of distributed object-based computation for two decades. This part of the dissertation gives an overview of the Actor model and presents A\( \pi \), a formalization of the Actor model as a typed asynchronous \( \pi \)-calculus. The type system imposes a certain discipline on the use of names to capture actor properties such as uniqueness and persistence. The notion of may testing in A\( \pi \) and a trace-based characterization of A\( \pi \) are then investigated. This characterization is compared with that of asynchronous \( \pi \)-calculus, and the differences that arise due to actor properties are highlighted.

Even though the Actor model provides a strong foundation for distributed object-based computation, its limited coordination capabilities make specification of coordination logics very difficult. To make up for this limitation, the second part of this dissertation presents SynchNet, a compositional meta-level language for coordination of distributed objects that is based on Petri Nets. Its design is based on the principle of separation of concerns, namely separation of the coordination from computational aspects. SynchNet can be used in combination with any object-based language.
capable of expressing sequential behavior of objects.
To my parents Nasrin and Manouchehr.
Acknowledgment

Working towards a Ph.D. has been the most ambitious and challenging project of my life. Going through all these years in the graduate school would not have been possible without the help of many people, to whom I wish to express my gratitude.

First and foremost, I am grateful to my advisor, Professor Gul Agha, for his advice and kind support in all stages of the Ph.D program. I thank him because of the way he influenced my research philosophy. He taught me how to balance idealism versus pragmatism in research. I would also like to thank him for his support through the times of personal difficulties.

My gratitude goes to the members of my defense committee, professors Mahesh Viswanathan, Jose Meseguer, Michael Loui, and Paul Schupp for their interest in my work and for their useful feedback. Particularly, I would like to thank professor Viswanathan for rekindling the joy of working on formal methods at a time that I was discouraged to pursue this discipline. Our collaboration lead to most of my results in Modal Logics. I would also like to thank professors Jose Meseguer and Michael Loui, not only for their constructive remarks on my dissertation, but also for their unparalleled disciplined and meticulous research attitude that has been an inspiration to me.

I am also greatly thankful to Professor Mehdi Harandi whose support in the first year of graduate school helped me go through the difficult process of adjusting to a new and challenging environment. I would always remember his and his wife, Farideh, for their personal support throughout all these years.

I would like to thank my colleagues and friends in the department of computer science who in one way or other kept me motivated all along and helped me realize that research is more effective as a group activity. I would like to thank Cesare Tinelli who was not only a great friend but also a great teacher. It was from him that I learned the fundamental meaning of logic and formal methods, which later became my main occupation. I will always remember all the pleasant nights
we spent in Chicago restaurants with him and his wife Ana. I would also like to thank my colleague Prasanna Thati. Most of the first half of this dissertation is the result of collaboration with him. I am also grateful to all my peers in the department of computer science, especially Gloria Rendon, Duane Searsmith, and Joshua Kaplan.

The Open System Laboratory was my research home for over six years. During this time I worked with many people who also made the times at OSL memorable. In particular, I would like to thanks Nadeem Jamali, James Waldby, Carlos Varela, Charles Zhang, Sudarshan Srinivasan, Nalini Venkatasubrahmanian, Mark Astley, Nirman Kumar, Sandeep Umatachandi, Predrag Tosic, Wooyoung Kim, Myeong-Wuk Jang, Youngmin Kwon, Po-Hao Chang, Amr Ahmed, Kirill Mechitov, Sameer Sundresh, Koushik Sen, Abhay Vardhan, Mehwish Nagda, Marcelo Bezerra D’Amorim, J ohsua Chia, Tom Brown, and Soham Mazumdar. I would also like to thank Greg Haik, Jean-Pierre Briot, and Caroly Talcott, whose feedback was extremely valuable.

Finally I would like to thank my best friends who were great support throughout all theses years, especially I am grateful to Bijan Ghahreman, Ali Namazifard, Farzad Mashayekhi, Bardia Sadri, Azadeh Farzan, Nasrin Sarrati, Amir Kashef, Salem Desiavi, Reza Lahroodi, Prasad Naldurg, Marco Caccamo, Tsvi Achler, Michael Mulryan, Zahra Golshani, Hadi LeilNahari, Hamidreza Chitsaz, and Dr. Mirshamseddin Adibsoltani. I would also like to especially thank Amy Kam for her love and support during the last and most difficult years of the graduate school.

Finally I would like to thank my family, Monouchehr, Nasrin, Minoo, Mehrdad, Forough, Ata, Ehsan, Leili, and Maryam for their unconditional love and support.
# Table of Contents

Chapter 1  Introduction .......................................................... 1
  1.1 Logics for Asynchronously Communicating Objects ...................... 3
  1.2 Meta-level Coordination of Objects Using Petri Nets .................. 5

Chapter 2  Asynchronous Process Calculi and May Testing ..................... 7
  2.1 Introduction ........................................................................ 7
  2.2 Asynchronous CCS ............................................................ 7
  2.3 May Testing for Asynchronous CCS ....................................... 9
  2.4 Asynchronous $\pi$-calculus .............................................. 10
  2.5 May Testing for Asynchronous $\pi$-calculus ............................ 13

Chapter 3  A Theory of May Testing for $L_\pi^=$ and $L_\pi$ ................... 15
  3.1 Introduction ........................................................................ 15
  3.2 Trace Based Characterization for $L_\pi^=$ ............................... 16
  3.3 Trace Based Characterization for $L_\pi$ .................................. 19
  3.4 An Axiomatization of Finitary $L_\pi^=$ and $L_\pi$ ...................... 26
  3.5 Related Work ...................................................................... 31

Chapter 4  A Modal Logic for ACCS .................................................. 33
  4.1 Introduction ........................................................................ 33
  4.2 Two Modal Logics for Asynchronous CCS ............................... 35
    4.2.1 Syntax and Semantics of $L_\text{ACCS}^+$ and $L_\text{ACCS}^-$ ............. 36
    4.2.2 A Logic of Safety: $L_\text{ACCS}^-$ ........................................... 37
    4.2.3 Logical Characterization of May-Invariant Properties by $L_\text{ACCS}^-$ .. 40
    4.2.4 Expressive Power .......................................................... 43
  4.3 Summary and Discussion ...................................................... 44

Chapter 5  Actors: From $L_\pi$ to Distributed Objects ......................... 46
  5.1 Introduction ........................................................................ 46
  5.2 The Actor Model ................................................................... 47
  5.3 The Calculus $A_\pi$ ............................................................ 48
  5.4 Type System ........................................................................ 49
  5.5 Reduction Semantics ........................................................... 53
  5.6 May Testing ......................................................................... 57
  5.7 Labeled Transition System .................................................... 58
  5.8 Alternate Characterization of May Testing ............................... 66
  5.9 Variants of $A_\pi$ ............................................................... 75
Chapter 1

Introduction

Building reliable distributed object-based systems is challenging. Among many software engineering practices to guarantee sufficient reliability that many applications demand, two approaches stand out. First is the formal approach to verification of correctness conditions. Formal methods, when applicable, provide the complete guarantee, at least with respect to formal specification, of correctness that testing methods cannot provide. Unfortunately, these methods are complicated in terms of both formulation and the computational complexity required to solve them automatically. It is of no surprise that a considerable research activity in the computer science community is devoted towards formalizing models, logics, and decision procedure for verification of larger and larger classes of systems.

The second approach, which is taken up by the programming language community, is to invent novel linguistic ideas and programming features to guarantee reliability at development time. Successful instances of this approach in the history of programming language design include concepts such as data abstraction, encapsulation, modular programming, scope, inheritance, structured synchronization mechanisms (such as rendez-vous, critical sections) among others. A more recent trend follows the principle of separation of concerns and advocates specification of different aspects of a system separately, and mostly in distinct languages each design to semantically reflect a particular aspect of the system. The so called aspect-oriented programming provides the advantage of verifying correctness of different aspects of the system independent of other aspects.

The work that resulted in this dissertation offers two solutions, one in the spirit of formal verification and one in the spirit of language mechanism design. The dissertation contributes to both these solutions:
In the first part of the dissertation, we start with a theory of may testing for asynchronous calculi with locality and no name matching. Locality is a non-interference property that is common in systems based on the object-paradigm. Concurrent languages such as Join and Pict disallow name matching, which is akin to pointer comparison in imperative languages. Disallowing name matching provides for an abstract semantics that would allow useful program transformations. May testing is widely acknowledged to be an effective mechanism for reasoning about safety properties. We provide a trace-based characterization of may testing for a version of asynchronous π-calculus with locality and no name matching (called Lπ) and some of its variations. This characterization greatly simplifies establishing equivalences between processes. Using the characterization, a complete axiomatization for the finitary fragment of the calculi is presented. Even though Lπ provides some basis for modeling distributed objects, it still lacks two important characteristics: encapsulation and object identity.

The first part of the dissertation extends the may testing results to the Actor model of distributed object-based computation. The Actor model has been an inspiring model of distributed object-based computation for two decades [1]. This part of the dissertation gives an overview of the Actor model and presents Aπ, a formalization of the Actor model as a typed asynchronous π-calculus. The type system imposes a certain discipline on the use of names to capture actor properties such as uniqueness and persistence. The notion of may testing in Aπ and a trace based characterization of Aπ are then investigated. This characterization is compared with that of asynchronous π-calculus, and the differences that arise due to actor properties are highlighted.

Even though the Actor model provides a formal foundation for distributed object based computation, its limited coordination capabilities make specification of coordination logics very difficult. To address this limitation, the second part of the thesis presents SynchNet, a compositional meta-level language for coordination of distributed objects. Its design is based on the principle of separation of concerns, namely separation of the coordination from computational aspects. SynchNet can be used in combination with any object-based language capable of expressing sequential behavior of objects. SynchNet, which is inspired by Petri nets, has a simple syntax and semantics, but is expressive enough to code many of the commonly used coordination patterns. The level of abstraction that it provides allows tools and techniques developed for Petri nets to be readily
applied to analysis and verification of the specified coordination patterns.

1.1 Logics for Asynchronously Communicating Objects

It is widely understood that automated verification of computer programs involves many challenging tasks. First, one has to determine the right semantic model for computation; a model that captures the essential aspects of the system one is interested in verifying. Then, it is important to precisely delineate the properties to be verified. This is usually followed by the design of a formal language (usually a logic) to express the specified properties. At the end, one has to determine decidability and computational complexity of corresponding verification problems.

This thesis addresses the problem of verification of distributed systems; a class of concurrent systems characterized by the absence of both a global clock and a global shared memory. As a consequence, processes run asynchronously and communicate via asynchronous message passing. Some distributed systems allow dynamic creation of processes, communication of channel names (mobility), and dynamic creation of communication channels. These characteristics have a great impact on the choice of both the semantic model and the set of properties to be verified.

We adopt the process calculus approach to model distributed systems and use asynchronous versions of process calculi CCS [42] and π-calculus [43]. More specifically, especially for the first part of the thesis, we consider a sub-calculus of asynchronous π-calculus called Lπ (“local” π-calculus). Lπ [3] is a reasonable linguistic framework to describe and specify the behavior of distributed object-based systems. Being a descendant of π-calculus, Lπ models mobility via communication and creation of channel names, which in object-oriented settings can be interpreted as identities of objects. Furthermore, Lπ restricts communication to asynchronous message passing: the fundamental mode of communication in distributed systems. Lπ further imposes the so called locality constraint that prevents a process from receiving messages on channel names which have already been received by the process. The locality property exists in many distributed programming languages such as Pict [49] and Join calculus [18]. Adding the uniqueness constraint, which requires every name be used in at most one input operation at any time, brings Lπ very close to the Actor model of computation [2] that has been practically used as a basis for distributed computing and concurrent object-oriented programming.
For verification purposes, we focus our attention on properties of processes that are observable by another process when composed concurrently. More specifically, we adopt the testing framework of Hennessy [26]. In this framework, a process \( P \) is tested against an observing process \( O \) by considering the set of computations that their composition \( P|O \) can perform. Usually, it is assumed that the observing process is capable of sending a distinguished message, and a computation of \( P|O \) is called successful if it leads to the emission of that message. Two processes \( P \) and \( Q \) are defined to be equivalent if for every process \( O \), the composed processes \( P|O \) and \( Q|O \) have “indistinguishable” sets of computations. There are two ways to define when two sets of computations are distinguishable. In may testing the two sets are distinguished if only one of the two sets has a successful computation. In must testing the two sets are distinguished if only one of them lacks successful computations. It is generally known that may testing equivalence distinguishes processes based on their safety properties and must testing based on their liveness properties.

Reasoning about testing equivalences using the definitions given above is difficult because the definitions involve quantification over all observing processes. Therefore, it is desirable to find characterizations that simplify the reasoning process. One such characterization is in terms of traces that a process exhibits and has been given for both synchronous [26] and asynchronous process calculi [6]. Using these trace based characterization, it is possible to obtain axiomatizations, which are sound and complete for finite state processes, and that they can be used in the implementation of symbolic verifiers. In Chapter 3, we extend the work of [6] and present a trace based characterization of may testing for \( L\pi \). We also use this characterization to obtain a sound and complete axiomatization for finite processes.

In recent years, model-checking has proved to be a successful approach to automated verification of concurrent systems. This motivates us to explore the formulation of a model-checking problem for asynchronous process calculi in Chapter 4. Considering that in model-checking concurrent systems, properties to be verified are described in some modal logics, we decided to look for a modal logic that expresses testing properties of processes described in an asynchronous language. More specifically, we explore a modal logic characterization of may testing equivalence. That is, we try to find a logic that best characterizes the class of properties that remain invariant under may testing equivalence. We explore the reasons why a new logic, which is more tightly coupled
with the testing semantics of processes should be looked for. Using the obtained logics we define the model-checking problem and investigate its decidability and computational complexity.

Part one of the dissertation consists of the following chapters. In Chapter 2, we define the syntax and semantics of asynchronous CCS and $\pi$-calculus together with the definition of may testing equivalences for them. In Chapter 3, we present the trace-based characterization of two variants of L$\pi$ and an axiomatization for finite processes. In Chapter 4, we present a modal logic that is expected to characterize testing equivalence for asynchronous CCS. Chapter 5 presents the syntax and semantics of A$\pi$ along with a trace based characterization of may preorder.

1.2 Meta-level Coordination of Objects Using Petri Nets

To manage the complexity of designing distributed object systems, many proposed frameworks advocate separation of coordination from computational aspects of systems [36]. One distinct group of solutions in this category may be called the two-level approach. A two-level framework consists of two languages: a base language in which the functionality of application processes and objects is described, and a meta language in which the developer specifies the coordination logic that governs the interaction among application level objects. Examples of such frameworks include the Synchronizers of [19], the reflective meta-architecture of [4], and the Two-Level Actor Machine (TLAM) of [62]. The use of ‘meta’ vs. ‘base’ terminology reflects the view that meta-level coordination policies are in fact modifications to the interaction semantics of the base application.

Two-level languages usually have an involved semantics. As a result, it may be difficult to understand programs written in these frameworks, and it is usually even harder to reason about them. This difficulty especially arises when the meta-level components are allowed to access the state of the base-level objects; such a permissive design creates a source of interference that is difficult to control. To counter these difficulties many proposed solutions disallow meta-level coordination components to access the states of base-level objects.

Frølund [19] has proposed a coordination language and framework in which a group of distributed objects are coordinated by entities called synchronizers. Each synchronizer is responsible for coordination of a group of objects: it decides when a message may or must be delivered to an object in the group. Synchronizers do not have access to the state of coordinated objects and maintain
their own independent state. The decision to approve a message delivery is based on predicates that refer to the state of the synchronizer and the information in the message. The state of the synchronizer is updated whenever an approved message is delivered. Therefore, the state of the synchronizer can be seen as an abstraction of some global snapshot of the states of the objects in the group. With this kind of abstraction, which provides a virtual local view of distributed actions, it is much simpler to solve coordination problems than with a language that only provides asynchronous message passing as a means of communication. Depending on the compiler for the synchronizer language, either centralized or distributed code may be generated.

Chapter 6 proposes a new language called SynchNet. The design of SynchNet follows the same design principles as a meta-level language called Synchronizers [19], but in contrast to Synchronizers, which are based on a Turing-complete language, SynchNet is based on Petri Nets [47]. Petri Nets is a formal modeling language for concurrent systems that has received wide academic and practical interest since its introduction by Carl Adam Petri in 1962 [47]. Its popularity is due to its rich and well-studied theory together with a friendly and easy-to-understand graphical notation. Petri Nets are less powerful than Turing machines, and therefore verification of many interesting properties is decidable [16]. In particular, reachability is decidable, which makes Petri Nets useful in the verification of safety properties such as deadlock-freedom.

Publications

The material in Chapter 3 is published in the proceedings of “Algebraic Methodology and Software Technology (AMAST02)”, Reunion Island, France, 2002 [58]. The material in Chapter 5 is published in the proceedings of “the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 02)”, Deventer, The Netherlands, 2002 [59]. The material in Chapter 6 is published in the proceedings of “the Second International Conference on Generative Programming and Component Engineering (ACM/GPCE03)”, Erfurt, Germany, 2003 [63].
Chapter 2

Asynchronous Process Calculi and May Testing

2.1 Introduction

Distributed systems consist of a group of processes that run concurrently. The main characteristics of distributed systems are the absence of a global clock and the absence of shared memory between any two processes. It is reasonable to assume that the most fundamental means of communication is *asynchronous* message passing, where sending of messages is non-blocking, and the sender continues its computation while the message is in transit to its destination. CCS (Calculus of Communicating Systems) [42] and its descendant $\pi$-calculus [43] are widely acknowledged formalisms for communication based concurrent systems. However, both formalisms are based on synchronous communication. More recently, asynchronous versions of CCS (ACCS) and $\pi$-calculus have been defined and studied [29, 33, 39, 48, 52]. More specifically, a theory of may testing for both calculi has been studied in [6]. In this chapter we introduce the syntax and operational semantics of ACCS and asynchronous $\pi$-calculus, and present a summary of the trace-based characterization of may-testing given in [6].

2.2 Asynchronous CCS

We let $\mathcal{N}$ be an infinite set of names ranging over by $a, b, c, \ldots$ and $\overline{\mathcal{N}} = \{ \overline{a} | a \in \mathcal{N} \}$ be the set of co-names ranging over by $\overline{a}, \overline{b}, \overline{c}, \ldots$. We also use names in $\mathcal{N}$ to model input actions and co-names to model output actions. $\mathcal{N} \cap \overline{\mathcal{N}} = \emptyset$ and the unary *complementation* operator ($\overline{\cdot}$), defined such
that \( (\overline{a}) = a \), is a bijection from \( \mathcal{N} \) to \( \overline{\mathcal{N}} \). Let \( \mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}} \) be the set of visible actions ranging over by \( l, l', \ldots \), and \( \mathcal{L}_\tau = \mathcal{L} \cup \{ \tau \} \) be the set of all actions or labels ranging over by \( \mu \), and \( \tau \) is the silent action. We let \( s \) range over \( \mathcal{L}^* \) and be an observable trace. We let \( X, Y, \ldots \) range over a countable set of process variable.

The set of asynchronous CCS processes (ACCS from now on), ranging over by \( P, Q, R \) is defined inductively:

\[
P ::= a | \sum_{i \in I} g_i.P_i | P_1|P_2 | P\\setminus L | P\{f\} | X | \text{rec } X.P
\]

with the following intuitive interpretation: The output action \( \overline{a} \) represents an output on channel name \( a \). The content is abstrated away, but will come back in \( \pi \)-calculus. The summation \( \sum_{i \in I} g_i.P_i \) represents non-deterministic choice among a collection of processes \( g_i.P_i \) \( (i \in I) \). \( g_i \) in \( g_i.P_i \) is either a label representing an input operation on a channel with that label, or is \( \tau \) (the silent action), in which case, it represent an internal operation of the process the detail of which is abstracted away. As a result of this abstraction, we do not distinguish between different \( \tau \) actions. We say that the process \( g_i.P_i \) is guarded by \( g_i \). Parallel composition \( P_1|P_2 \) represents the possibility of either processes \( P_1 \) or \( P_2 \) executing their actions concurrently and independently. The restriction \( P\\setminus L \), with \( L \) being a set of labels, prevents \( P \) from communicating on channel names in \( L \). Restriction has the effect of localizing channel names in the same way that blocks in imperative languages limit the scope of locally defined variables. Renaming \( P\{f\} \) allows renaming of labels used in \( P \) according to a given map \( f: \mathcal{N} \rightarrow \mathcal{N} \). The fixed point expression \( \text{rec } X.P \) represents a process \( P \) in which all occurrences of \( X \) are processes whose behavior is exactly the same as \( \text{rec } X.P \). This expression allows definition of recursive processes.

The operational semantics of asynchronous CCS is formalized as the labeled transition system \((\mathcal{P}, \mathcal{L}_\tau, \xrightarrow{\mu})\) defined by the rules in Table 2.1.

We use \( \xrightarrow{\tau} \) and \( \xrightarrow{\epsilon} \) to denote the reflexive and transitive closure of \( \xrightarrow{\tau} \) and use \( \xrightarrow{s} \) for \( \xrightarrow{I} \xrightarrow{s'} \) when \( s = ls' \). We also write \( P \xrightarrow{s} P' \) when there exists \( P' \) such that \( P \xrightarrow{s} P' \).

We now define may preorder and equivalence on ACCS process terms.

**Definition 1** An observer is an ACCS process that can perform a distinct output action \( \overline{a} \) (the success action). Let \( \mathcal{O} \) be the set of all observers ranging over by \( O, O', \ldots \). A computation from a
process \( P \) and an observer \( O \) is a sequence of transitions

\[
P|O = P_0|O_0 \xrightarrow{\tau} P_1|O_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_k|O_k \xrightarrow{\tau} \cdots
\]

which is either infinite or the last process term \( P_k|O_k \) cannot perform a silent action. The computation is successful if there exists some \( n \geq 0 \) such that \( O_n \xrightarrow{w} \). For a process \( P \) and an observer \( O \), we say \( P \) may \( O \) if and only if there exists a successful computation from \( P|O \). \( \square \)

**Definition 2 (may-testing preorder)** For processes \( P \) and \( Q \), we say \( P \sqsubseteq_m Q \) if and only if for each observer \( O \), \( P \) may \( O \) implies \( Q \) may \( O \). \( \square \)

We will use \( \simeq_m \) to denote the equivalence defined as \( \simeq_m = \sqsubseteq_m \cap \sqsubseteq_m^{-1} \).

### 2.3 May Testing for Asynchronous CCS

We present a summary of the fully-abstract trace-based characterization of may testing for asynchronous CCS given in [6].

**Definition 3 (A preorder on \( L^* \))** Let \( \preceq_0 \) be the least relation that satisfies the following laws.

1. **TO1** \( s_1s_2 \preceq_0 s_1as_2 \)
2. **TO2** \( s_1las_2 \preceq_0 s_1als_2 \)
3. **TO3** \( s_1s_2 \preceq_0 s_1a\bar{s}s_2 \)
where $s_1, s_2 \in \mathcal{N}^*$ are observable traces, $a$ is an input action, $\pi$ is an output action, and $l$ is either an input or an output action. $\preceq$ is the reflexive and transitive closure of $\succeq_0$.

The trace preorder defined above is used in defining the alternate characterization of may testing preorder between two processes.

**Definition 4 (alternative preorder)** For processes $P$ and $Q$, we write $P \preceq_m Q$ if whenever $P \xrightarrow{s} \Rightarrow$ then there exists $s'$ such that $s' \preceq s$ and $Q \xrightarrow{s'} \Rightarrow$.

Then the following theorem establishes the equivalence of the alternative preorder with the may testing preorder $\sim_m$.

**Theorem 1** For processes $P$ and $Q$, $P \sim_m Q$ if and only if $P \preceq_m Q$.

The following defines a set-based interpretation that maps may equivalent processes to the same semantic object.

**Definition 5** For a process $P$, its interpretation is defined as

$$[P]_m \overset{\text{def}}{=} \{ [s] | s \in L(P) \text{ and for } s' \in L(P) : s' \preceq s \text{ implies } [s] = [s'] \}$$

where $[s] \overset{\text{def}}{=} \{ s'| s \preceq s' \text{ and } s' \preceq s \}$.

The following theorem holds.

**Theorem 2** For processes $P$ and $Q$, $P \preceq_m Q$ if and only if for every $[s] \in [P]$ there is $[s'] \in [Q]$ such that $s' \preceq s$.

The relation $[s] \leq [s']$ defined as $s \preceq s'$ is a partial order on sets of traces. We say $[s]$ is $\leq$-minimal if it is minimal with respect to the $\leq$ partial order.

### 2.4 Asynchronous $\pi$-calculus

Now, we extend the syntax and semantics of CCS to that of the $\pi$-calculus. The $\pi$-calculus provides for communication of channel names as message content. Syntactically, this is achieved by extending
the syntax of the input and output operations to account for message contents. Sending of messages in π-calculus are represented by the syntax \( \overline{xy} \), which is interpreted as sending a message over a channel named \( x \) carrying the message content \( y \). The syntax for a process ready to receive a message is \( x(y).P \), where \( x \) is the name of the channel on which the input operation is performed and \( y \) is the name of a place holder for the content of the message communicated over the channel. The role \( y \) plays in the expression \( x(y).P \) is similar to that of formal parameters in functional languages: all occurrences of \( y \) in the process expression \( P \) refer to the argument \( y \) in \( x(y) \) and are replaced with the actual content of the message after a message is “delivered” to the receiving process.

In this section we formally define the syntax and semantics of π-calculus and then proceed to giving a definition for a may preorder over π-calculus processes and its trace-based characterisation.

We assume an infinite set of names \( \mathcal{N} \), and let \( u, v, w, x, y, z, \ldots \) range over \( \mathcal{N} \). Variables \( P, Q, \) and \( R \) range over the set of processes, which is defined by the following restricted π-calculus [43] grammar.

\[
P ::= \overline{xy} \mid \sum_{i \in I} \alpha_i.P_i \mid (\nu x)P \mid P_1P_2 \mid [x = y]P \mid !P
\]

where \( \alpha \) can be an input action \( x(y) \) or a silent action \( \tau \). The name \( x \) is said to be the subject of the output action \( \overline{xy} \) and the input action \( x(y).P \).

For a tuple \( \hat{x} \), we denote the set of names occurring in \( \hat{x} \) by \( \{\hat{x}\} \). We write \( \hat{x}, \hat{y} \) for the result of appending \( \hat{y} \) to \( \hat{x} \). We let \( \hat{z} \) range over \( \{\emptyset, \{z\}\} \). The term \( (\nu \hat{z})P \) is \( (\nu z)P \) if \( \hat{z} = \{z\} \), and \( P \) otherwise. The functions for free names, bound names and names, \( fn(\cdot) \), \( bn(\cdot) \) and \( n(\cdot) \), of a process, and alpha equivalence on processes are defined as usual. We use the usual definition and notational convention for name substitutions, and let \( \sigma \) range over them. Name substitution on processes is defined modulo alpha equivalence with the usual renaming of bound names to avoid captures. We write \( P\sigma \) and \( x\sigma \) to denote the result of applying \( \sigma \) to \( P \) and \( x \) respectively.

We use an early style labeled transition system for the operational semantics (see Table 2.2). The transition system is defined modulo alpha-equivalence on processes in that alpha-equivalent processes have the same transitions. The symmetric versions of \( \text{COM} \), \( \text{CLOSE} \), and \( \text{PAR} \) are not shown. Transition labels, which are also called actions, can be of five forms: \( \tau \) (a silent action), \( \overline{xy} \) (free output of a message with target \( x \) and content \( y \)), \( \overline{x}(y) \) (bound output), \( xy \) (free input of a
INP \[ \sum_{i \in I} \alpha_i \cdot P_i \xrightarrow{xy} P_j \{z/y\} \quad (j \in I, \alpha_j = \overline{xy}_j) \]

TAU \[ \sum_{i \in I} \alpha_i \cdot P_i \xrightarrow{\tau} P_i \quad (j \in I, \alpha_j = \tau) \]

PAR \[ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 | P_2 \xrightarrow{\alpha} P_1'|P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset \]

RES \[ \frac{P \xrightarrow{\alpha} P'}{(vy)P \xrightarrow{\alpha} (vy)P'} \quad y \notin n(\alpha) \]

CLOSE \[ \frac{P_1 \xrightarrow{(\overline{y})u} P_1' P_2 \xrightarrow{xy} P_2'}{P_1 | P_2 \xrightarrow{xy} (vy)(P_1'|P_2')} \quad y \notin fn(P_2) \]

REP \[ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \quad Matching \]

MATCH \[ \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha}, P'} \]

Table 2.2: An early style labeled transition system for asynchronous π-calculus.

message) and \( x(y) \) (bound input). The relation \( \overline{xy} \) is defined by the additional rule \( P \xrightarrow{\overline{xy}} Q \) if \( P \xrightarrow{xy} Q \) and \( y \notin fn(P) \). We denote the set of all visible (non-\( \tau \)) actions by \( \mathcal{L} \), let \( \alpha \) range over \( \mathcal{L} \), and let \( \beta \) range over all the actions. The functions \( fn(\cdot) \), \( bn(\cdot) \) and \( n(\cdot) \) are defined on \( \mathcal{L} \) the usual way. As a uniform notation for free and bound actions we adopt the following convention from [6]: 

\( (\emptyset)\overline{xy} = \overline{xy}, (\{y\})\overline{xy} = \overline{y}xy, \) and similarly for input actions. We define a complementation function on \( \mathcal{L} \) as \( (\overline{y})xy = (\overline{y})\overline{xy} = (\overline{y})xy \).

We let \( s, r, t \) range over \( \mathcal{L}^* \). The functions \( fn(\cdot) \), \( bn(\cdot) \) and \( n(\cdot) \) are extended to \( \mathcal{L}^* \) the obvious way. Complementation on \( \mathcal{L} \) is extended to \( \mathcal{L}^* \) the obvious way. Bound names can be alpha-converted to new names, in exactly the same way that bound names in lambda calculus may be renamed via alpha-conversion. Two traces obtained from one another via alpha-conversion are said to be alpha-equivalent. We will work with traces modulo alpha-equivalence. Furthermore, we will consider only normal traces \( s \in \mathcal{L}^* \) that satisfy the following hygiene condition: if \( s = s_1.\alpha.s_2 \), then \( (n(s_1) \cup fn(\alpha)) \cap bn(\alpha.s_2) = \emptyset \). For an action \( \alpha \) and a set of traces \( S \) we define \( \alpha.S = \{ \alpha.s | s \in S \} \).

We use \( \Longrightarrow \) to denote the reflexive transitive closure of \( \xrightarrow{\tau} \), and \( \overset{\beta}{\Longrightarrow} \) to denote \( \Longrightarrow^{\beta} \). For \( s = l.s' \) we use \( P \xrightarrow{s} Q \) to denote \( P \xrightarrow{l} s'Q \), and similarly \( P \overset{s}{\Longrightarrow} Q \) to denote \( P \overset{l}{\Longrightarrow} s'Q \). We write \( P \overset{s}{\Longrightarrow} \) if \( P \overset{s}{\Longrightarrow} Q \) for some \( Q \), and similarly for \( P \overset{s}{\Longrightarrow} \) and \( P \overset{\tau}{\Longrightarrow} \). We say \( P \) exhibits the trace \( s \) if \( P \overset{s}{\Longrightarrow} \).
The definition of may testing preorder and equivalence for asynchronous π-calculus is formally the same as the one defined for ACCS.

### 2.5 May Testing for Asynchronous π-calculus

Now, we present a summary of the trace-based characterization of may-testing preorder given in [6]. To avoid infinitary branching, a transition system with synchronous inputs instead of asynchronous inputs is used. To account for asynchrony, the trace semantics is modified using a trace preorder ≤ that is defined as the reflexive transitive closure of the laws shown in Table 2.3. The notation (\(\hat{y}\))· is extended to traces as follows.

\[
(\hat{y})s = \begin{cases} 
    s & \text{if } \hat{y} = \emptyset \text{ or } y \not\in \text{fn}(s) \\
    s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\
    s = s_1.xy.s_2 \text{ and } y \not\in \text{fn}(s_1) \cup \{x\} \\
    \bot & \text{otherwise}
\end{cases}
\]

The intuition behind the preorder is that if an observer accepts a trace \(s\), then it also accepts any trace \(r \leq s\). Laws L1-L3 capture asynchrony, and L4 captures the inability to mismatch names. Laws L1 and L2 state that an observer cannot force inputs on the process being tested. Since outputs are asynchronous, the actions following an output in a trace exhibited by an observer need not be causally dependent on the output. Hence the observer’s outputs can be delayed until a causally dependent action (L2), or dropped if there are no such actions (L1). Law L3 states that an observer can consume its own outputs unless there are subsequent actions that depend on the output. Law L4 states that without mismatch an observer cannot discriminate bound names from free names, and hence can receive any name in place of a bound name. The intuition behind the trace preorder is formalized in the following lemma.
Lemma 1 If $P \xrightarrow{\pi}$, then $r \preceq s$ implies $P \xrightarrow{r}$.

The may preorder $\subseteq_m$ in asynchronous $\pi$-calculus is then characterized according the following theorem.

Theorem 3 (Trace-based characterization) $P \subseteq_m Q$ if and only if $P \xrightarrow{s}$ implies $Q \xrightarrow{r}$ for some $r \preceq s$. 
Chapter 3

A Theory of May Testing for \( L_{\pi_{\equiv}} \) and \( L_{\pi} \)

3.1 Introduction

We now develop a theory of may testing for two subcalculi of asynchronous \( \pi \)-calculus [6]: \( L_{\pi_{\equiv}} \) and \( L_{\pi} \). \( L_{\pi_{\equiv}} \) is a variant of asynchronous \( \pi \)-calculus that follows the discipline of locality. Locality disallows a process from receiving messages on a channel name that has been received as message content by the same process. Locality is typical in systems based on an object paradigm [1]. \( L_{\pi} \) is a variant of \( L_{\pi_{\equiv}} \) in which the name matching construct, which is analogous to pointer comparison in imperative languages, is absent. The advantages of lack of name matching include the possibility of certain program transformations that would otherwise be unsound. In fact, name comparisons are disallowed by concurrent languages such as Pict [49]. In any case, comparing names is rarely useful in programming; the behavior observed while communicating at a name is all that matters, and the specific name used for communication is irrelevant. \( L_{\pi} \) is introduced in [39].

In this chapter, we first present a trace-based characterization of may testing for \( L_{\pi_{\equiv}} \) and \( L_{\pi} \). The work generalizes and is inspired by the work in [7]. In fact, we characterize a parameterized version of may testing where the parameter determines the set of observers that is used to decide may preorder between two processes. The usual notion of may testing is just a special case.

Second, we present complete axiomatizations of finitary \( L_{\pi_{\equiv}} \) and \( L_{\pi} \) (for processes with no recursion) based on the alternate characterizations for may testing. The axiomatizations highlight the differences that arise due to locality and lack of name matching. In addition to laws that are true for asynchronous \( \pi \)-calculus, we obtain laws that are true only for testing in the presence of
locality and the absence of name matching. Further, the inference rules for parameterized may testing generalize the ones for the usual may testing.

### 3.2 Trace Based Characterization for $L_{\pi_=}$

$L_{\pi_=}$ is a subcalculus of asynchronous $\pi$-calculus with its syntax defined by the following grammar.

$$P ::= 0 \mid \pi y \mid x(y).P \mid P | P \mid (\nu x)P \mid [x = y]P \mid !x(y).P$$

The locality property is enforced by requiring that for every subterm of the form $x(y).P$, the bound name $y$ does not occur as the subject of an input in $P$. The operational semantics of $L_{\pi_=}$ is defined as the labeled transition system in Table 3.1.

We now instantiate the testing framework [26] on $L_{\pi_=}$. In fact, as an extension to the notion of locality, we consider a generalized version of may testing that supports encapsulation. We define a parameterized may preorder $\subseteq_{\rho}$, where only observers that do not listen on names in $\rho$ are used to decide the order. The set of names $\rho$ can be interpreted as being “owned” by the process being tested, in that any context in which the process is executed in is assumed to have only the capability to send messages to these names. The reader may note that $\subseteq_{\emptyset}$ is the usual may preorder.

**Definition 6 (may testing)** An observer is a process that can emit a special message $\overline{\mu}$. We let
O range over the set of observers. We say \( O \) accepts a trace \( s \) if \( \overline{\mu}O \xrightarrow{s} \nu \). For \( P,O \), we say \( P \) may \( O \) if \( P|O \xrightarrow{m} \). Let \( rcp(P) \) be the set of all free names in \( P \) that occur as the subject of an input in \( P \). For any given \( \rho \) we say \( P \preccurlyeq_{\rho} Q \) if for every \( O \) such that \( rcp(O) \cap \rho = \emptyset \), \( P \) may \( O \) implies \( Q \) may \( O \). We say \( P \simeq_{\rho} Q \) if \( P \preccurlyeq_{\rho} Q \) and \( Q \preccurlyeq_{\rho} P \). Note that \( \preccurlyeq_{\rho} \) is reflexive and transitive, and \( \simeq_{\rho} \) is an equivalence relation. \( \square \)

The larger the parameter of a preorder, the smaller the observer set that is used to decide the order. Hence if \( \rho_1 \subset \rho_2 \), we have \( P \preccurlyeq_{\rho_1} Q \) implies \( P \preccurlyeq_{\rho_2} Q \). However, \( P \preccurlyeq_{\rho_2} Q \) need not imply \( P \preccurlyeq_{\rho_1} Q \). For instance, \( 0 \simeq_{\{x\}} \overline{xx} \), but only \( 0 \preccurlyeq_{\emptyset} \overline{xx} \) and \( \overline{xx} \nsubseteq_{\emptyset} 0 \). Similarly, \( \overline{xx} \simeq_{\{x,y\}} \overline{yy} \), but \( \overline{xx} \nsubseteq_{\emptyset} \overline{yy} \) and \( \overline{yy} \nsubseteq_{\emptyset} \overline{xx} \). However, \( P \preccurlyeq_{\rho_2} Q \) implies \( P \preccurlyeq_{\rho_1} Q \) if \( fn(P) \cup fn(Q) \subset \rho_1 \).

**Theorem 4** Let \( \rho_1 \subset \rho_2 \). Then \( P \preccurlyeq_{\rho_1} Q \) implies \( P \preccurlyeq_{\rho_2} Q \). Further, if \( fn(P) \cup fn(Q) \subset \rho_1 \) then \( P \preccurlyeq_{\rho_2} Q \) implies \( P \preccurlyeq_{\rho_1} Q \). \( \square \)

Note that \( L_{\pi_{\equiv}} \) is a subcalculus of asynchronous \( \pi \)-calculus. That is, every \( L_{\pi_{\equiv}} \) process is also a process of asynchronous \( \pi \)-calculus. Therefore, Lemma 1, that we proved to hold for asynchronous \( \pi \)-calculus also holds for \( L_{\pi_{\equiv}} \) without any modifications.

We now build on the trace-based characterization of may testing for asynchronous \( \pi \)-calculus presented in [6] to obtain a characterization of may testing in \( L_{\pi_{\equiv}} \). We note that \( L_{\pi_{\equiv}} \) is a proper subcalculus of the calculus in [6], that is, every \( L_{\pi_{\equiv}} \) term is also an asynchronous \( \pi \)-calculus term, and the transition systems of the two calculi match on the common terms. May testing in \( L_{\pi_{\equiv}} \) is weaker than in asynchronous \( \pi \)-calculus because the locality property reduces the number of observers that can be used to test processes. For example, the following two processes are distinguishable in asynchronous \( \pi \)-calculus but equivalent in \( L_{\pi_{\equiv}} \).

\[
P = (\nu x)(!x(z).0|\overline{xx}y) \quad Q = (\nu x)(!x(z).0|\overline{yy}x)
\]

The observer \( O = y(z).z(w).\overline{\mu} \mu \) can distinguish \( P \) and \( Q \) in asynchronous \( \pi \)-calculus, but is not a valid \( L_{\pi_{\equiv}} \) term as it violates locality. In fact, no \( L_{\pi_{\equiv}} \) term can distinguish \( P \) and \( Q \), because the message \( \overline{xx} \) is not observable.

To account for locality we need to consider only the traces that correspond to interaction between \( L_{\pi_{\equiv}} \) processes. Note that the transition system does not by itself account for locality.
For instance, in case of the example above, we have $P \xrightarrow{\overline{yx}} \overline{tx}$ although the message $\overline{tx}$ is not observable. To counter this deficiency, we define the notion of well-formed traces.

**Definition 7** For a set of names $\rho$ and trace $s$ we define $\text{rcp}(\rho, s)$ inductively as

$$
\text{rcp}(\rho, \epsilon) = \rho \\
\text{rcp}(\rho, s.(\hat{y})xy) = \text{rcp}(\rho, s) \\
\text{rcp}(\rho, s.(\hat{y})\overline{tx}) = \text{rcp}(\rho, s) \cup \hat{y}
$$

We say $s$ is $\rho$-well-formed if $s = s_1.(\hat{y})\overline{tx}.s_2$ implies $x \notin \text{rcp}(\rho, s_1)$. We say $s$ is well-formed if it is $\emptyset$-well-formed.

Only $\rho$-well-formed traces correspond to an interaction between a process and an L$_\pi$-observer $O$ such that $\text{rcp}(O) \cap \rho = \emptyset$. We are now ready to give the alternate characterization of $\equiv_{\rho}$ in L$_\pi$.

**Definition 8** We say $P \ll_{\rho} Q$, if for every $\rho$-well-formed trace $s$, $P \xrightarrow{s} \text{implies there is } r \preceq s \text{ such that } Q \xrightarrow{r}.$

To prove the characterization, we define an observer $O(s)$ for a well-formed trace $s$, such that $P \underline{\text{may}} O(s)$ implies $P \xrightarrow{r}$ for some $r \preceq s$. This construction is the same as the one used for asynchronous $\pi$-calculus [6].

**Definition 9 (canonical observer)** For a trace $s$, we define $O(s)$ as follows:

$$
O(\epsilon) \overset{\text{def}}{=} \overline{\pi} t \\
O((\hat{y})xy.t) \overset{\text{def}}{=} (v\hat{y})(\overline{ty}|O(t)) \\
O(\overline{ty}.t) \overset{\text{def}}{=} x(y).O(t) \\
O(\overline{ty}.s) \overset{\text{def}}{=} x(u).[u = y]O(s) \quad u \text{ fresh}
$$

Note that well-formedness of $s$ guarantees that $O(s)$ is an L$_\pi$ term. Furthermore, it is easy to show that if $s$ is $\rho$-well-formed, then $\text{rcp}(O(s)) \cap \rho = \emptyset$. Since the canonical observer constructions match and L$_\pi$ is a subcalculus of asynchronous $\pi$-calculus, the following lemma proved for asynchronous $\pi$-calculus [6], also holds in L$_\pi$.

**Lemma 2** For a well-formed trace $s$, $O(s) \xrightarrow{\overline{ty}}$ implies $r \preceq s$.

Theorem 5 proves the equivalence of $\equiv_{\rho}$ and $\ll_{\rho}$ in L$_\pi$. Before proving Theorem 5, we present a lemma that shows that the internal computation of the composed process $P|O$ can be seen as the composition of observable and complementary computations of processes $P$ and $O$. 18
Lemma 3  Let $\rho$ be a set of names where $\text{rcp}(O) \cap \rho = \emptyset$. Then $P|O \overset{\mu}{\Rightarrow}$ can be “unzipped” into $P \overset{s}{\Rightarrow}$ and $O \overset{\overline{r}\overline{\mu}}{\Rightarrow}$ for some $s$ that is $\rho$-well-formed. \hfill \Box

Theorem 5  $P \overset{\leq}{\sim}_\rho Q$ if and only if $P \leq_\rho Q$.

Proof: (if) Let $P \leq_\rho Q$ and $P \overset{\rho}{\text{may}} O$ for an observer $O$ such that $\text{rcp}(O) \cap \rho = \emptyset$. From $P \overset{\rho}{\text{may}} O$ we have $P|O \overset{\mu}{\Rightarrow}$. By Lemma 3, this computation can be unzipped into $P \overset{s}{\Rightarrow}$ and $O \overset{\overline{r}\overline{\mu}}{\Rightarrow}$ for some $\rho$-well-formed trace $s$. From $P \leq_\rho Q$, there is a trace $r \preceq s$ such that $Q \overset{r}{\Rightarrow}$. Moreover, $r \preceq s$ implies $r.\mu \mu \preceq s.\mu \mu$. Therefore, by Lemma 1, $O\overset{r}{\Rightarrow}$. We can zip this with $Q \overset{r}{\Rightarrow}$ to obtain $Q|O \overset{\overline{r}\overline{\mu}}{\Rightarrow}$, which means $Q \overset{\mu}{\text{may}} O$.

(only if): Let $P \overset{\leq}{\sim}_\rho Q$ and $P \overset{s}{\Rightarrow}$ where $s$ is $\rho$-well-formed. We have to show that there is a trace $r \preceq s$ such that $Q \overset{r}{\Rightarrow}$. Now, it is easy to show that $O(s) \overset{\overline{r}\overline{\mu}}{\Rightarrow}$. This can be zipped with $P \overset{s}{\Rightarrow}$ to get $P|O(s) \overset{\mu}{\Rightarrow}$, that is $P \overset{\rho}{\text{may}} O(s)$. From $P \overset{\leq}{\sim}_\rho Q$, we have $Q \overset{\mu}{\text{may}} O(s)$ and therefore $Q|O(s) \overset{\overline{r}\overline{\mu}}{\Rightarrow}$. This can be unzipped into $Q \overset{r}{\Rightarrow}$ and $O(s) \overset{\overline{r}\overline{\mu}}{\Rightarrow}$. From Lemma 2, it follows that $r \preceq s$. \hfill \Box

3.3 Trace Based Characterization for $L_\pi$

We now investigate the effect of lack of name matching capability. We remove the match operator from $L_{\pi=}$, to obtain the calculus $L_{\pi}$. The rules in Table 3.1 except the $\text{MATCH}$ rule, constitute the transition system for $L_{\pi}$.

The lack of name matching capability further weakens may testing equivalence. For example, the processes $(\nu u)(\overline{u}u|u)$ and $(\nu u, v)(\overline{u}u|u)$ are equivalent in $L_{\pi}$, but not in $L_{\pi=}$. For the alternate characterization of $P \overset{\leq}{\sim}_\rho Q$, it is too stringent to require that for any trace $s$ that $P$ exhibits, $Q$ exhibits a single trace $r$ such that any observer accepting $s$ also accepts $r$. In fact, there exist $L_{\pi}$ processes $P$ and $Q$ such that $P \overset{\leq}{\sim}_\rho Q$, and if $P$ exhibits $s$, then $Q$ exhibits different traces to satisfy different observers that accept $s$. For instance, let $P = \overline{u}u_1|\overline{j}u_1|u_1(w)\overline{w}w$ which can exhibit
\( s = \overline{x}u_1.\overline{y}u_1.u_1(w).\overline{w} \). The following L\( \pi \) observers accept \( s \).

\[
O_1 = (\nu w)(x(u).y(v).\overline{w}w|w(v).\overline{\rho}\overline{\mu}) \\
O_2 = (\nu w)(x(u).y(v).\overline{w}w|w(v).\overline{\rho}\overline{\mu}) \\
O_3 = (\nu w)(x(u).y(v).\overline{w}1w|w(v).\overline{\rho}\overline{\mu}) \\
O_4 = (\nu w)(x(u).y(v).|v\overline{v}u_1u_1|u_1(z).\overline{w}w| w(v).\overline{\rho}\overline{\mu})
\]

Now consider

\[
Q = (\nu v)(v(z).y(z').|v\overline{v}z\overline{z}|v\overline{u}u_1|\overline{u}u_2| !u_2(z).\overline{w}w| u_1(w).\overline{w}w)
\]

which can satisfy

\[
O_1 \text{ with } r_1 = \overline{x}u_1.\overline{y}u_2.u_1(w).\overline{w}w \\
O_2 \text{ with } r_2 = \overline{x}u_2.\overline{y}u_1.u_1(w).\overline{w}w \\
O_3 \text{ with } r_1 \text{ or } r_2, \text{ and } \\
O_4 \text{ with } r_4 = \overline{x}u_1.\overline{y}u_2.u_2u_2.\overline{u}1u_2.u_1(w).\overline{w}w
\]

but cannot exhibit a single trace that can satisfy all four observers. In fact, it is the case that

\( P \not\leq_\emptyset Q \). Intuitively, although unlike \( P \), \( Q \) always exports two different names at \( x \) and \( y \), for each possible dataflow pattern of the received names inside an observer that \( P \) satisfies, \( Q \) exhibits a corresponding trace that can lead the observer to a success.

For the alternate characterization, we define templates which are a special kind of traces that can be used to represent dataflows in an observer. A template is a trace in which all outputs are bound. The binding relation between arguments of outputs and their subsequent free occurrences, represents the relevant dependencies between the output argument that is received by an observer and its subsequent use in the observer’s computation. For a trace \( s \) and set of names \( \rho \), we define a set \( T(s, \rho) \) that has a template for each possible dataflow in a computation \( O \overline{\pi}\overline{\mu}^{\overline{\tau}} \) with \( rcp(O) \cap \rho = \emptyset \). Further, if \( t \) represents the dataflow in a computation \( O \overline{\pi}\overline{\mu}^{\overline{\tau}} \), then it will be the case that \( O \overline{\pi}\overline{\mu}^{\overline{\tau}} \). Thus, if an observer accepts a trace \( s \), then it also accepts a template in \( T(s, \rho) \). This template construction essentially captures the effect of lack of match operator. We will show that \( P \not\leq_\rho Q \) if and only if for every \( \rho \)-well-formed trace \( s \) that \( P \) exhibits and for each \( t \in T(s, \rho) \), \( Q \) exhibits some \( r \leq t \).

Following is an informal description of how the set \( T(s, \rho) \) can be obtained. Due to the lack of
name matching capability, an observer cannot fully discriminate between free inputs. Therefore, a process can satisfy an observer $O$ that exhibits $O \xrightarrow{\pi \mu} \Rightarrow$, by replacing free input arguments in $\pi$ with any name as long as it is able to account for changes to the subsequent computation steps that depend on the replaced name. Specifically, suppose $O \xrightarrow{\pi \mu}$ abbreviates the following computation:

$$O \xrightarrow{\pi} O_0 \xrightarrow{xy} O_1 \xrightarrow{\beta_1} O_2 \xrightarrow{\beta_2} \cdots O_n \xrightarrow{\beta_n} \mu \xrightarrow{\mu}$$

Because of the locality property, the name $y$ received in the input may be used only in output terms of $O_1$. We call such occurrences of $y$ as dependent on the input. During subsequent computation, these output terms may appear either as an output action or are consumed internally. In the latter case, $y$ may be the target of the internal communication, or the argument which in turn may generate further output terms with dependent occurrences of $y$. Therefore, $O$ can do the following computation when $y$ in the input is replaced with an arbitrary name $w$:

$$O \xrightarrow{\pi} O_0 \xrightarrow{(\hat{w})xw} O_1 \xrightarrow{\gamma_1} O_2 \xrightarrow{\gamma_2} \cdots O_n \xrightarrow{\gamma_n} \mu$$

where $\gamma_i$ is obtained from $\beta_i$ as follows. If $\beta_i$ is an output action, then $\gamma_i$ is obtained from $\beta_i$ by substituting dependent occurrences of $y$ with $w$. If $\beta_i$ is an internal delivery of a message $yz$ with target $y$ being a dependent occurrence, there are two possibilities. If $z$ is a private name, then $\gamma_i = \overline{w}(z).yz$ and the subsequent bound output $\beta_j$ ($j > i$) that exports $z$ for the first time (if any), is changed to a free output. If $z$ is not a private name, then $\gamma_i = \overline{w}z'.yz'$, where $z'$ is $w$ when $z$ is a dependent occurrence of $y$ and $z$ otherwise. For all other cases, $\gamma_i = \beta_i$. Note that, if $w$ is fresh, the input of $w$ could be a bound input.

Clearly, any computation obtained by repeated application of the above construction can be performed by $O$. In particular, if we always replace free inputs with bound inputs, we will eventually obtain a computation in which all inputs are bound and the construction can not be applied any further. Let $O \xrightarrow{\tau_{s,\rho}}$ abbreviate a computation thus obtained. The trace $t$ is a template that explicitly represents all dependencies between received names (bound input arguments) and subsequent computation steps (subsequent free occurrences of the argument). The set $T(s,\rho)$ consists of all the templates that can be obtained by this construction starting from arbitrary
computations of the form $O \xrightarrow{\pi} \mu$ with $\text{rcp}(O) \cap \rho = \emptyset$.

We now formalize the ideas presented above, leading to a direct inductive definition of $T(s, \rho)$. Let

$$O \xrightarrow{\pi} xy \xrightarrow{\mu} O_1 \xrightarrow{\mu} \mu$$

We first consider the simple case where $y \notin \text{rcp}(O_1)$. Due to locality, in the computation following input $xy$, there cannot be an internal message delivery with $y$ as the target. Therefore, the following computation is possible.

$$O \xrightarrow{(\bar{w})xw} O_1' \xrightarrow{s_2} \mu$$

where $s_2'$ is obtained from $\bar{s}_2$ by renaming dependent occurrences of $y$ in output actions to $w$. Specifically, it does not involve exposing internal actions that use dependent occurrences of $y$. When the computation steps above are not known, all we can say about $s_2'$ is that it is obtained from $\bar{s}_2$ by renaming some occurrences of $y$. Similarly, $O_1'$ is obtained from $O_1$ by renaming some occurrences of $y$ in output terms. These relations are formalized in Definition 10 and Lemma 4.

**Definition 10 (random output substitution)** For $\sigma = \{\bar{u}/\bar{v}\}$ we define random output substitution (from now on just random substitution) on process $P$, denoted by $P[\sigma]$, modulo alpha equivalence as follows. We assume $\text{bn}(P) \cap \{\bar{v}\} = \text{fn}(P)\sigma \cap \text{bn}(P) = \emptyset$. For a name $x$ we define $x[\sigma] = \{x, x\sigma\}.$

$$0[\sigma] = \{0\} \quad \quad (x(y).P)[\sigma] = \{x(y).P' \mid P' \in P[\sigma]\}$$

$$\{(\bar{y})y\}[\sigma] = \{\bar{x'}y' \mid x' \in x[\sigma], y' \in y[\sigma]\} \quad \quad (P|Q)[\sigma] = \{P'|Q' \mid P' \in P[\sigma], Q' \in Q[\sigma]\}$$

$$((\nu x)P)[\sigma] = \{(\nu x)P' \mid P' \in P[\sigma]\} \quad \quad (!x(y).P)[\sigma] = \{!x(y).P' \mid P' \in P[\sigma]\}$$

Random substitution on traces is defined modulo equivalence as follows. We assume $\text{bn}(s) \cap \{\bar{v}\} = \text{fn}(s)\sigma \cap \text{bn}(s) = \emptyset.

$$\epsilon[\sigma] = \{\epsilon\} \quad \quad (x(y).s)[\sigma] = \{x'(y).s' \mid x' \in x[\sigma], s' \in s[\sigma]\}$$

$$((\bar{y})y.s')[\sigma] = \{(\bar{y})y.s' \mid s' \in s[\sigma]\} \quad \quad (xy.s)[\sigma] = \{x'y'.s' \mid x' \in x[\sigma], y' \in y[\sigma], s' \in s[\sigma]\}$$

We will use $[\bar{u}/\bar{v}]$ as a short form for $[\{\bar{u}/\bar{v}\}]$. □

**Lemma 4** If $P \xrightarrow{\pi} P', P' \in P[w/y]$, and $y \notin \text{rcp}(P)$, then $P' \xrightarrow{\mu}$ for some $s' \in s[w/y].$ □
Now, suppose \( y \in rcp(O_1) \). Then, in the computation

\[
O \xrightarrow{xy} O_1 \xrightarrow{p} \]

certain internal transitions may involve a message with a dependent occurrence of \( y \) as the target. Then, the following computation which exposes such transitions is also possible

\[
O \xrightarrow{(w)xw} O'_1 \xrightarrow{s'_2} \]

where \( s'_2 \) is obtained from \( s_2 \) by not only renaming all dependent occurrences of \( y \) in output transitions to \( w \), but also exposing each internal message delivery with a dependent occurrence of \( y \) as the message target. If the computation steps are not known, we can only say \( s'_2 \) is obtained from some \( r \in s_2[w/y] \) by exposing arbitrary number of internal transitions at any point in \( r \). The relation between \( s_2 \) and \( s'_2 \) is formalized in Definition 11 and Lemma 5. To account for the situation where an exposed pair of actions \((\hat{z}wz.\hat{y}z)\) export a private name \( z \), we need the following function on traces.

\[
[\hat{y}]s = \begin{cases} 
  s & \text{if } \hat{y} = \emptyset \text{ or } y \notin n(s) \\
  s_1.x.y.s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\
  s = s_1.x(y).s_2 \text{ and } y \notin n(s_1) \cup \{x\} \\
  \bot & \text{otherwise} 
\end{cases} 
\]

**Definition 11** For a trace \( s \) and a pair of names \( w, y \), the set \( F(s, w, y) \) is the smallest set closed under the following rules:

1. \( \epsilon \in F(\epsilon, w, y) \)

2. \( (\hat{v})uv.s' \in F((\hat{v})uv.s, w, y) \) if \( s' \in F(s, w, y) \)

3. \( (\hat{v})wv.s' \in F((\hat{v})wv.s, w, y) \) if \( s' \in F(s, w, y) \)

4. \( (\hat{z})wz.\hat{y}z.[\hat{z}]s' \in F(s, w, y) \) if \( s' \in F(s, w, y) \) and \( [\hat{z}]s' \neq \bot \)

Note that \( s \in F(s, w, y) \). For a set of traces \( S \), we define \( F(S, w, y) = \cup_{s \in S} F(s, w, y) \).

**Lemma 5** If \( P \xrightarrow{\pi} \) and \( P' \in P[w/y] \), then \( P' \xrightarrow{\pi} \) for some \( s' \in F(s[w/y], w, y) \).
For a trace $s$ and a set of names $\rho$, we say $s$ is $\rho$-normal, if $s$ is normal and $\rho \cap bn(s) = \emptyset$. Now, let $O$ be an arbitrary observer such that $rcp(O) \cap \rho = \emptyset$. Suppose

$$O \xrightarrow{xy} O_1 \xrightarrow{\rho}$$

where $s_1.xy.s_2$ is $\rho$-normal. If $y \in \rho$ or $y$ is the argument of a bound input in $s_1$, then by locality $y \notin rcp(O_1)$. Otherwise, since $O$ is arbitrary, it is possible that $y \in rcp(O_1)$. From this observation, we have that for an arbitrary observer $O$ such that $rcp(O) \cap \rho = \emptyset$, if $O$ accepts the $\rho$-normal trace $s_1.xy.s_2$, then $O$ also accepts $s_1.(\hat{w})\tau w.s'_2$ where $w$ is an arbitrary name and $s'_2 \in s_2[w/y]$ if $y \in \rho$ or $y$ is the argument of a bound output in $s_1$, and $s'_2 \in F(s_2[w/y],w,y)$ otherwise. $T(s,\rho)$ is precisely the set of all traces with no free outputs, that can be obtained by repeated application of this reasoning. $T(s,\rho)$ is formally defined in Definition 12.

**Definition 12** For a trace $s$ and a set of names $\rho$, the set of templates $T(s,\rho)$ is defined modulo alpha equivalence as follows. We assume that $s$ is $\rho$-normal.

1. $\epsilon \in T(\epsilon,\rho)$.
2. $(\hat{y})xy.s' \in T((\hat{y})xy.s,\rho)$ if $s' \in T(s,\rho)$
3. $\tau(y).s' \in T(\tau(y).s,\rho)$ if $s' \in T(s,\rho \cup \{y\})$
4. $\tau(w).s' \in T(\tau(y).s,\rho)$ if $w$ fresh, $s' \in T(s'',\rho \cup \{w\})$, and
   $$s'' \in \begin{cases} s[w/y] & \text{if } y \in \rho \\ F(s[w/y],w,y) & \text{if } y \notin \rho \end{cases}$$

The reader may check that if $t \in T(s,\rho)$, then $s \preceq t$ using only $L3$ and $L4$. □

**Lemma 6** If $P \xrightarrow{\tau}$ and $\rho \cap rcp(P) = \emptyset$, then there is $t \in T(s,\rho)$ such that $P \xrightarrow{t}$. □

Lemma 7 states that template construction in Definition 12 preserves $\rho$-well-formedness.

**Lemma 7** If $s$ is $\rho$-well-formed then every $t \in T(s,\rho)$ is $\rho$-well-formed. □

We are now ready to give the alternate characterization of $\simeq_\rho$ in $L_{\pi}$. 24
Definition 13 We say \( P \preceq_\rho Q \) if for every \( \rho \)-well-formed trace \( s \), \( P \xrightarrow{s} \) implies for each \( t \in T(s, \rho) \) there is \( r \leq t \) such that \( Q \xrightarrow{r} \).

For \( t \in T(s, \rho) \), where \( s \) is a \( \rho \)-well-formed trace, let \( O(t) \) be the canonical observer as defined in Definition 9. By Lemma 7, since \( s \) is \( \rho \)-well-formed \( t \) is also \( \rho \)-well-formed. Hence \( O(t) \) satisfies the locality property, and \( rcp(O(t)) \cap \rho = \emptyset \). Further, since \( t \) is a template, the case \( t = xy.t' \) does not arise in the construction of the observer. Hence \( O(t) \) is an \( L_\pi \) term. Since \( L_\pi \) is a subcalculus of asynchronous \( \pi \)-calculus, Lemma 1 holds for \( L_\pi \). Further, since the canonical observer construction is unchanged, the following lemma (which is a weaker version of Lemma 2) holds for \( L_\pi \).

Lemma 8 For \( t \in T(s, \rho) \), where \( s \) is a \( \rho \)-well-formed trace, \( O(t) \xrightarrow{\pi, \mu} \) implies \( r \leq t \).

Lemma 3 holds for \( L_\pi \) with formally the same proof. Now, we are ready to prove that \( \preceq_\rho \) is an alternate characterization of \( \preceq_\rho \).

Theorem 6 \( P \preceq_\rho Q \) if and only if \( P \preceq_\rho Q \).

Proof: (if) Let \( P \preceq_\rho Q \) and \( P \text{ may } O \) for an observer \( O \) such that \( rcp(O) \cap \rho = \emptyset \). From \( P \text{ may } O \) we have \( P|O \xrightarrow{\mu} \). By Lemma 3, this computation can be unzipped into \( P \xrightarrow{s} \) and \( O \xrightarrow{\pi, \mu} \) for some \( \rho \)-well-formed trace \( s \). From Lemmas 1 and 6 we deduce there is a \( t' \in T(s, \mu \mu, \rho) \) such that \( r' \leq t' \) implies \( O \xrightarrow{r} \). It is easy to show that \( t' \in T(s, \mu \mu, \rho) \) implies \( t' = t, \mu \mu \) for some \( t \in T(s, \rho) \).

From \( P \preceq_\rho Q \), there is a trace \( r \leq t \) such that \( Q \xrightarrow{r} \). Moreover, \( r \leq t \) implies \( r, \mu \mu \leq t, \mu \mu = t' \).

Therefore, \( O \xrightarrow{\pi, \mu} \). We can zip this with \( Q \xrightarrow{r} \) to obtain \( Q|O \xrightarrow{\pi, \mu} \), which means \( Q \text{ may } O \).

(only if): Let \( P \preceq_\rho Q \) and \( P \xrightarrow{s} \) where \( s \) is \( \rho \)-well-formed. We have to show for every \( t \in T(s, \rho) \) there is a trace \( r \leq t \) such that \( Q \xrightarrow{r} \). It is easy to show that if \( t \in T(s, \rho) \), then \( O(t) \xrightarrow{\pi, \mu} \). This can be zipped with \( P \xrightarrow{s} \) to get \( P|O(t) \xrightarrow{\pi, \mu} \), that is \( P \text{ may } O(t) \). From \( P \preceq_\rho Q \), we have \( Q \text{ may } O(t) \) and therefore \( Q|O(t) \xrightarrow{\pi, \mu} \). This can be unzipped into \( Q \xrightarrow{r} \) and \( O(t) \xrightarrow{\pi, \mu} \).

From Lemma 8, it follows that \( r \leq t \).

For finitary processes we can obtain a simpler characterization based on a modified version of Definition 12 as given below.

Definition 14 For a trace \( s \) and a set of names \( \rho \), the set \( T_f(s, \rho) \) is defined inductively using the first three rules of Definition 12 and the following two.
The main difference from Definition 12 is that output arguments \( y \) that are not in \( \rho \) are not converted to bound arguments. According to rule 4 of Definition 12, such conversions introduce arbitrary number of pairs of input/output actions. But, since the length of traces that a finite process can exhibit is bounded, the only way the process can exhibit a trace \( r \preceq t \) for each of the resulting templates, is by emitting the same name \( y \), so that \( L4 \) and \( L3 \) can be applied to annihilate some of these input/output pairs. The following lemma helps formalize this observation.

**Lemma 9** For a trace \( s \), a set of names \( \rho \), and a prefixed closed set \( R \) of traces with bounded length, if for every \( t \in T(s, \rho) \) there exists \( r \in R \) such that \( r \preceq t \), then for every \( t_f \in T_f(s, \rho) \) there exists \( r \in R \) such that \( r \preceq t_f \).

Using this lemma, we can show that for finitary processes we can use \( T_f(s, \rho) \) in Definition 13 instead of \( T(s, \rho) \). The resulting characterization is equivalent to the earlier one for the following reason. Suppose \( P \Rightarrow \tilde{\sim} \) implies, for every \( t \in T(s, \rho) \), there exists \( r \preceq t \) such that \( Q \Rightarrow \tilde{r} \). Then, let \( R \) be the set of all traces that \( Q \) exhibits. Note that \( R \) is prefix closed. Further, since \( Q \) is finite, there is a bound on the length of traces in \( R \). By Lemma 9, for every \( t_f \in T_f(s, \rho) \), there exists \( r \preceq t_f \) such that \( Q \Rightarrow \tilde{r} \). Conversely, suppose \( P \Rightarrow \tilde{\sim} \) implies that for every \( t \in T_f(s, \rho) \) there exists \( r \preceq t \) such that \( Q \Rightarrow \tilde{r} \). It is easy to verify that for every \( t \in T(s, \rho) \) there exists a \( t_f \in T_f(s, \rho) \) such that \( t_f \preceq t \), where the relation can be derived using only \( L3 \) and \( L4 \). From transitivity of \( \preceq \), it follows that \( P \Rightarrow \tilde{\sim} \) implies for every \( t \in T(s, \rho) \) there exists \( r \preceq t \) such that \( Q \Rightarrow \tilde{r} \).

### 3.4 An Axiomatization of Finitary Lπ= and Lπ

We first give a sound and complete proof system for \( \tilde{\sim}_\rho \) for the finitary fragment of \( L\pi \), i.e. for \( L\pi \) processes that do not use replication. A simple adaptation of the proof system gives us one for finitary \( L\pi= \). The proof system consists of the laws given in Table 3.2 and the rules for reflexivity and transitivity. For a finite index set \( I \), we use the macro \( \sum_{i \in I} P_i \) to denote, \( (\nu u)(\langle |i \in I | u \rangle . P_i)|u \rangle \) for \( u \) fresh if \( I \neq \emptyset \), and 0 otherwise. For an index set that is a singleton, we omit \( I \) and simply
write $\sum P$ instead of $\sum_{i \in I} P_i$. We let the variable $G$ range over processes of form $\sum_{i \in I} P_i$. We write $\sum_{i \in I} P_i + \sum_{j \in J} P_j$ to denote $\sum_{k \in I \cup J} P_k$. We write $\sqsubseteq$ as a shorthand for $\sqsubseteq_\emptyset$, and $=$ for $=_\emptyset$.

Random input substitution on processes $P[w/y]$ is defined similar to random output substitution (Definition 10), except that only the occurrences of $y$ at the subject of input prefixes in $P$ are randomly substituted with $w$.

While axioms $A1$ to $A19$ all hold in asynchronous $\pi$-calculus [6], axioms $A20$ and $A21$ are unique to $L\pi$. $A20$ captures the fact that a message targeted to a name that an environment is prohibited from listen to, cannot escape to the environment. The axiom states that there are only two ways such a message can be handled in the next transition step: it can be consumed internally or delayed for later. The axiom also accounts for delaying the message forever by including dropping of the message as one of the possibilities. As an application of this axiom, if $x \in \rho$, we can prove $\exists y \sqsubseteq_\rho 0$ as follows. For $w$ fresh, $\exists y \sqsubseteq_\rho (\nu x)(w(w).0)$.

1. $\sqsubseteq_\rho (\nu w)(\exists y|w(w).0)$ (A3, A11, I1)
2. $\sqsubseteq_\rho (\nu w)(\exists y|w(w).0)$ (A8)
3. $\sqsubseteq_\rho (\nu w)(\sum w(w).0 + \sum w(w) \exists y + \sum 0)$ (A20, I1)
4. $\sqsubseteq_\rho \sum (\nu w)(w(w).0) + \sum (\nu w)w(w) \exists y + \sum (\nu w)0$ (A7)
5. $\sqsubseteq_\rho 0$ (A1, A11, A14, I3)

Axiom $A21$ captures the effect of lack of match operator. It is directly motivated from rule 4 of Definition 14 for template construction.

The inference rules extend the rules for asynchronous $\pi$-calculus to handle parameterization of the may preorder. In fact, the rules for asynchronous $\pi$-calculus presented in [6] can be obtained by setting $\rho = \emptyset$ in $I1$, $I2$ and $I3$. $I4$ is a new rule that is motivated by Theorem 4. We make a few remarks about $I1$ which is significantly different from its analogue for asynchronous $\pi$-calculus.

First, using $\exists y \sqsubseteq_\{x\} 0$ (proved above) and $II$, we get $(\nu x)\exists y \sqsubseteq (\nu x)0$, and by axiom $A19$ we have $(\nu x)0 \sqsubseteq 0$. Therefore, $(\nu x)\exists y \sqsubseteq 0$. Note the use of the ability to contract the parameter $\rho$ of the may preorder after applying a restriction. Second, the following example illustrates the necessity of the side condition $rcp(R) \cap \rho = \emptyset$ for composition: $\exists y \sqsubseteq_\{x\} 0$ but not $\exists y|x(y)\exists y \sqsubseteq_\{x\} x(y)\exists y$, for the LHS can satisfy the observer $y(u).\exists \mu$ and the RHS can not.

The soundness of rules $II-I4$ can be easily proved directly from Definition 6. We only show the argument for $II$, which is given in Lemma 10. Soundness of axioms $A1-A21$ is easy to check. For
### Inference Rules

1. If $P \subseteq \rho Q$ and $rcp(R) \cap \rho = \emptyset$, then $\langle \nu x \rangle P \subseteq_{\rho - \{x\}} \langle \nu x \rangle Q$, $P[R] \subseteq_{\rho} Q[R]$.
2. If for each $z \in fn(P, Q)$, $P[z/y] \subseteq_{\rho} Q[z/y]$ then $x(y).P \subseteq_{\rho} x(y).Q$
3. If for each $i \in I$, $P_i \subseteq_{\rho} \sum_{j \in J} Q_{ij}$ then $\sum_{i \in I} P_i \subseteq_{\rho} \sum_{i \in I, j \in J} Q_{ij}$
4. If $\rho_1 \subseteq \rho_2$ and $P \subseteq_{\rho_1} Q$ then $P \subseteq_{\rho_2} Q$.

### Axioms

\begin{align*}
A1 & \quad G + G = G \\
A2 & \quad G \subseteq G + G' \\
A3 & \quad P|0 = P \\
A4 & \quad P|Q = Q|P \\
A5 & \quad (P|Q)|R = P|(Q|R) \\
A6 & \quad (\nu x)(\sum_{i \in I} P_i) = \sum_{i \in I}(\nu x)P_i \quad x \notin n(P) \\
A7 & \quad (\nu x)(\nu y|\alpha.P) = \alpha.(\nu x)(\nu y|P) \quad x \notin n(\alpha) \\
A8 & \quad (\nu x)(\nu y|\alpha.P) = \nu x(\nu y|P) \quad x \notin n(\alpha) \\
A9 & \quad (\nu x)(\nu y|\alpha.P) = \nu x(\nu y|P) \quad x \notin n(\alpha) \\
A10 & \quad (\nu x)(\nu y|\alpha.P) = \nu x(\nu y|P) \quad x \notin n(\alpha) \\
A11 & \quad (\nu x)(y(z).P) = \begin{cases} y(z).(\nu x)P & \text{if } x \neq y, x \neq z \\ 0 & \text{if } x = y \end{cases} \\
A12 & \quad (\nu y|\sum_{i \in I} P_i) = \sum_{i \in I}(\nu y|P_i) \quad I \neq \emptyset \\
A13 & \quad \alpha.(\nu y|\sum_{i \in I} P_i) = \sum_{i \in I}(\alpha.P_i) \quad I \neq \emptyset \\
A14 & \quad P = \sum P \\
A15 & \quad x(y).(|w|P) \subseteq |w|x(y).P \quad y \neq u, y \neq v \\
A16 & \quad P(y/z) \subseteq |w|x(z).P \\
A17 & \quad x(u).y(v).P \subseteq y(v).x(u).P \quad u \neq y, u \neq v \\
A18 & \quad x(y).(|w|P) \subseteq P \quad y \notin n(P) \\
A19 & \quad (\nu x)P \subseteq P\{y/x\} \\
A20 & \quad \text{If } x \in \rho, w \neq x \text{ and } w \neq y, \text{ then} \\
& \quad (\nu w)|z(w).P \subseteq_{\rho} \sum z(w)(\nu x|P) + \sum z(w).P + \sum Q, \text{ where } Q = \begin{cases} P\{y/w\} & \text{if } x = z \\ 0 & \text{otherwise} \end{cases} \\
A21 & \quad \nu w|P \subseteq_{\rho} (\nu w|\sum_{P' \in P\{w/y\}, P'} \quad w \text{ fresh, } y \in \rho.
\end{align*}

Table 3.2: Laws for $L\pi$. 

28
A1-A19, whenever $P \sqsubseteq Q$, we have $P \xrightarrow{s} r$, implies $Q \xrightarrow{r}$ such that $r \leq s$. For A20, both LHS and RHS exhibit the same $\rho$-well-formed traces. Proof of soundness of axiom A21 is more involved, and is established in Lemma 10. The reader can verify that A20 and A21 would also be sound as equalities. For instance, the converse of A21 can be shown using A19, A1, and I1.

Lemma 10

1. If $P \sqsubseteq_{\rho} Q$ and $\text{rep}(R) \cap \rho = \emptyset$, then $(\nu x)P \sqsubseteq_{\rho-\{x\}} (\nu x)Q$, $P|R \sqsubseteq_{\rho} Q|R$.

2. For $y \in \rho$ and $w$ fresh, $\pi_{y}|P \sqsubseteq_{\rho} (\nu w)(\pi_{w}|\sum_{P'} \in P[w/y]} P')$. □

We prove that the laws presented constitute a complete proof system for finite processes, i.e. for finite processes $P, Q$, $P \sqsubseteq_{\rho} Q$ if $P \sqsubseteq_{\rho} Q$. Inspired by the alternate characterization, the proof relies on existence of canonical forms for processes.

Definition 15 If $s$ is a template, then we call $s$ a cotemplate. Thus, a cotemplate is a trace with no free inputs. If $s$ is well-formed, we say $s$ is cowell-formed.

1. For a cowell-formed cotemplate $s$, the process $e(s)$ is defined inductively as follows.

\[
\begin{align*}
e(\epsilon) & \overset{\text{def}}{=} 0 \\
e(\pi_{y}.s') & \overset{\text{def}}{=} (\nu y)(\pi_{y}|e(s')) \\
e(y.s') & \overset{\text{def}}{=} x(y).e(s')
\end{align*}
\]

Note that cowell-formedness of $s$ implies that $e(s)$ is an $L_{\pi}$ term. From now on we follow the convention that whenever we write $e(s)$ it is implicit that $s$ is a cowell-formed cotemplate.

2. The process $\sum_{s \in S} e(s)$, for a set of traces $S$, is said to be in canonical form. □

The proof of completeness relies on the following four lemmas. The first lemma states that every process has an equivalent canonical form.

Lemma 11 For every process $P$ there is a canonical form $C$ such that $P = C$. □

Lemma 12 (1) If $e(s) \xrightarrow{r}$, then $e(r) \sqsubseteq e(s)$. (2) If $s \leq r$ then $e(r) \sqsubseteq e(s)$. □

29
The proofs of the two lemmas above are formally the same as the proofs of the corresponding lemmas for asynchronous π-calculus [6]. This is because, the proofs of \( P = C \) and \( e(r) \subseteq e(s) \) constructed using the proof system of [6], can be transformed into proofs in our proof system. This claim is justified by the following observations. First, every L\( \pi \) term is also an asynchronous π-calculus term. Second, starting from L\( \pi \) terms, every term that appears in the proofs of [6] is also an L\( \pi \) term. Note that any summation that appears is finite and can be interpreted as our macro. Finally, every axiom and inference rule used in their proof is derivable in our proof system.

**Lemma 13** Let \( R \) contain all the cowell-formed cotemplates \( r \) such that \( e(s) \Rightarrow r \) and \( r \) is \( \rho \)-well-formed. Then \( e(s) \subseteq \rho \sum_{r \in R} e(r) \). □

**Lemma 14** \( e(s) \subseteq \rho \sum_{t \in T} e(t) \). □

Note that the summations in the two lemmas above are finite because \( R \) and \( T \) are finite modulo alpha equivalence. For instance, finiteness of \( R \) is a direct consequence of the following two observations. For every \( r \in R \), we have \( \text{fn}(r) \subseteq \text{fn}(e(s)) \), and since \( e(s) \) is a finite process, the length of traces in \( R \) is bounded.

We are now ready to establish the completeness of the proof system.

**Theorem 7** For finite L\( \pi \) processes \( P, Q \) and a set of names \( \rho \), \( P \subseteq \rho \ Q \) if and only if \( P \preceq_{\rho} Q \).

**Proof**: The only-if part follows from the soundness of laws in Table 3.2. We prove the if part. By lemma 11 and soundness of the proof system, without loss of generality, we can assume that both \( P \) and \( Q \) are in canonical form, i.e. \( P \) is of form \( \sum_{s \in S_1} e(s) \) and \( Q \) is of form \( \sum_{s \in S_2} e(s) \). Using Lemma 13, and laws I3, A1, we get \( P \subseteq \rho \sum_{r \in R} e(r) \), where \( R \) is the set of \( \rho \)-well-formed cowell-formed cotemplates that \( P \) exhibits. Using Lemma 14 and laws I3, A1, we have \( \sum_{r \in R} e(r) \subseteq \rho \sum_{t \in T} e(t) \), where \( T = \cup_{r \in R} T_f(r, \rho) \). Note that since every \( r \in R \) is a cotemplate, so is every \( t \in T \). Let \( t \in T \). Then \( t \in T_f(r, \rho) \) for some \( \rho \)-well-formed \( r \) that \( P \) exhibits. Using the characterization of may preorder based on \( T_f(r, \rho) \), we have \( P \underrel{\rho} Q \) implies there is \( s' \preceq t \) such that \( Q \Rightarrow s' \). It follows that for some \( s \in S_2, e(s) \Rightarrow s' \). Since \( Q \Rightarrow s' \), by locality, \( s' \) is cowell-formed. From the facts that \( s' \preceq t \) and \( t \) is a cotemplate, it follows that \( s' \) is a cotemplate. Then by Lemma 12.2 and law I4, \( e(t) \subseteq \rho e(s') \). Further, by Lemma 12.1 and law I4, \( e(s') \subseteq \rho e(s) \). Hence by transitivity...
of $\preceq_{\rho}$, we have $e(t) \preceq_{\rho} e(s)$. Since $t \in T$ is arbitrary, using laws I3, A1, and A2, we deduce $\sum_{t \in T} e(t) \preceq_{\rho} \sum_{s \in S_2} e(s)$. The result follows from transitivity of $\preceq_{\rho}$. \hfill $\Box$

We obtain a complete proof system for $L\pi$ by dropping axiom A2I and adding the following two for the match operator: $[x = x]P = P$, and $[x = y]P = 0$ if $x \neq y$. Completeness of the resulting proof system can be established by simple modifications to the proofs above, which we do not elaborate further due to space limitation.

### 3.5 Related Work

We have provided alternate characterization of a parameterized version of may testing for asynchronous variants of $\pi$-calculus with locality and no name matching. We have exploited the characterizations to obtain complete axiomatizations of the may preorder for finitary fragments of the calculi. Our results extend the ones obtained by Boreale, De Nicola, and Pugliese for asynchronous $\pi$-calculus [6]. We now compare our work with other related research.

Hennessy and Rathke [27] study typed versions of three behavioral equivalences, namely may and must equivalences, and barbed congruence in a typed $\pi$-calculus where the type system allows names to be tagged with input/output capabilities. In the typed calculus, one can express processes that selectively distribute different capabilities on names. The locality property is a special case in which only the output capability on names can be passed. A novel labeled transition system is defined over configurations which are process terms with two typed environments, one that constrains the process and the other the environment. It is shown that the standard definitions of trace and acceptance sets [26] defined over the new transition system characterize may and must preorders respectively. In comparison to our work, the typed calculus of Hennessy and Rathke is synchronous and is equipped with name matching, whereas $L\pi_=$ is asynchronous, and $L\pi$ is asynchronous with no name matching. Further, $L\pi_=$ has no capability types and hence we obtain a simpler characterization of may testing for it, which is based on the usual early style labeled transition system. Finally, we have also given an axiomatization of may testing, which is not pursued by Hennessy and Rathke.

There have been extensive investigations of bisimulation-based behavioral equivalences on $L\pi$.
and related variants of $\pi$-calculus, which are properly contained in may testing which is trace based. Merro and Sangiorgi [39] investigate barbed congruence in $L\pi$, and show that a variant of asynchronous early bisimulation provides an alternate characterization for the congruence. Boreale and Sangiorgi [8] study typed barbed equivalence for typed (synchronous) $\pi$-calculus with capability types and no name matching, and show that the equivalence is characterized by a typed variant of bisimulation. Merro [38] characterizes barbed congruence in the more restricted setting of asynchronous $\pi$-calculus with no name matching (no capability types, and no locality in particular). He defines synonymous bisimulation and shows that it characterizes barbed congruence in this setting.
Chapter 4

A Modal Logic for ACCS

4.1 Introduction

This chapter introduces and discusses the design of a modal logic for asynchronous CCS. Once a modal logic and its corresponding satisfiability relation are defined, we can explore the decidability and computational complexity of the induced model-checking problem. The obtained results will help develop insights into how to build automatic verification tools.

Many forms of modal and temporal logics for concurrent systems exist today. The question, then, is “why a new logic?” Logics differ from one another for several reasons, one is the expressive power. For instance, while a linear time temporal logic can express properties of computation sequences of a system, it fails to express properties about the structural parts of a program that correspond to non-deterministic choices. Similarly, a branching time logic such as CTL fails to express fairness properties of computation sequences because it is not possible to compose path quantifiers without the intervention of branching quantifiers.

In our particular case, we are interested in a class of properties that are observable by a may-testing observer. More precisely, we are interested in the class of properties that are invariant under may equivalence. Suppose we write $P \simeq_m Q$ to denote may equivalence of processes $P$ and $Q$, and $P \models \phi$ to mean that the process $P$ has property $\phi$. A set of properties $\Phi$ is said to be may equivalence invariant (or simply may invariant) if for every property $\phi \in \Phi$ the following holds

\[
\text{if } P \simeq_m Q \text{ then } P \models \phi \text{ if and only if } Q \models \phi.
\]
The usual modal logics for processes, both branching time logics, such as Hennessy-Milner, and linear time logics, such as linear time mu-calculus, fail to provide a characterization for the may invariant properties in which we are interested. For instance, branching time logics can distinguish $a.b + a.c$ from $a.(b + c)$, which are may testing equivalent in both a synchronous and an asynchronous calculus.

Another reason why conventional modal logics are too discriminating to characterize may equivalence is that in asynchronous setting, the order in which a process receives messages cannot be observed. For instance, consider the (may-equivalent) processes $a.b.P$ and $b.a.P$, with $P$ an arbitrary process. These two processes are distinguished by the Hennessy-Milner logic formula $\langle a \rangle \langle b \rangle \texttt{tt}$ and linear time mu-calculus formula $(a).(b).\texttt{tt}$. In general, any logic with modalities capable of expressing order of events has the power of distinguishing these two processes.

Another problem is that these logics can express properties that are meaningless in a distributed system. For instance the process $a \parallel b$ satisfies both

\[
\phi_1 \overset{\text{def}}{=} \langle a \rangle \langle b \rangle \texttt{tt} \quad \text{and} \quad \phi_2 \overset{\text{def}}{=} \langle b \rangle \langle a \rangle \texttt{tt}.
\]

Each of these two formulas specify some order on emission of messages $\overline{a}$ and $\overline{b}$, which does not make sense in a distributed system by virtue of unordered asynchronous communication.

After justifying the need for a new logic, we have to decide on the form of the modalities of the logic. To develop some intuition, let’s recall the informal interpretation of may preorder between two processes: $P \preceq_m Q$ if, by consuming the same messages, $Q$ can produce at least the same messages as $P$. Therefore, it is reasonable to express properties as (sequences of) pairs $(I,O)$ of multisets of messages with the intended meaning that if the process consumes some or all of the messages in $I$, it can produce at least all the messages in $O$. More elaborate properties can be formed by putting these pairs in sequences or use propositional connectives such as conjunction or disjunction to combine properties.

We choose the following syntax for the new modality: $\langle I \rightarrow O \rangle \phi$ where $I$ is a multiset of inputs, $O$ is a multiset of outputs, and $\phi$ is a formula of the logic. Intuitively, a trace $s$ has property
\langle \{a\} \rightarrow \{b, b\}\rangle tt if it has a prefix that contains at least two instances of \(b\) and at most one instance of \(a\). In other words, if the observed trace exhibits \(O = \{b, b\}\) either without waiting for an input or at worst after waiting for an input on channel \(a\), then the trace satisfies the property described by the formula.

With this modality and propositional connectives we can define a modal logic for asynchronous CCS. In the following section we present the syntax and semantics of the logic, discuss its expressive power, and study its properties including characterization of may testing equivalence.

### 4.2 Two Modal Logics for Asynchronous CCS

It is often said that may testing is related to safety properties. Surprisingly, this relationship has not been sufficiently explored. By presenting a modal logic that characterizes may preorder, we investigate such a relationship in the context of Asynchronous CCS.

We note that a logic that characterizes may-preorder must be a logic of “unsafety”. Let’s remember the definition of may preorder: we say \(P \preceq_m Q\) if \(P \text{ may } O\) implies \(Q \text{ may } O\). This definition is based on the observation that an observer \(O\) verifies a process \((P\) or \(Q)\) to the best of its interaction power to tell something about that process’s behavior. For instance, the observer could tell whether the process under observation exhibits an “unsafe” trace.

How does all this relates may preorder to verification of safety? Suppose we know that \(Q\) is safe, that is, it does not have any “unsafe” observable traces, then it can not be the case that \(P\) has “unsafe” observable trace, for in that case there would be an observer \(O\) such that \(P \text{ may } O\) but not \(Q \text{ may } O\). Hence \(P \preceq_m Q\) would be violated. In other words, if \(Q\) does not contain any “unsafe” observable trace, then establishing \(P \text{ may } O\) guarantees \(P\)’s safety.

Before presenting our logic of may preorder, we start with the observation that the set of may invariant properties is not closed under logical complementation. In other words, if \(\Phi\) is the set of may-invariant properties, there are formulas in \(\Phi^c\) that are not may invariant. As a result, a modal logic that characterizes may-preorder will not have a negation operator.

Based on this observation, we will present two modal logics: \(\mathcal{L}_+^{\text{ACCS}}\) that specifies “unsafe” properties, and \(\mathcal{L}_-^{\text{ACCS}}\) that specifies “safe” properties of Asynchronous CCS. After presenting the syntax and semantics of the two logics, we prove that \(\mathcal{L}_+^{\text{ACCS}}\) characterizes may preorder.
Furthermore, and that $\mathcal{L}_{\text{ACCS}}^+$ and $\mathcal{L}_{\text{ACCS}}^-$ are dual, in the sense that for every formula $\phi$ in $\mathcal{L}_{\text{ACCS}}^+$ there exists a formula $\phi'$ in $\mathcal{L}_{\text{ACCS}}^-$ such that for a process $P$, $P \models \phi$ if and only if $P \not\models \phi'$.

We use the notation and definitions introduced in Chapter 2 plus the following. For a sequence of actions $s$ we write $\{s\}$ to denote the multiset of actions in $s$. For multiset operations union, intersection, difference, $\ldots$, we use the same symbols used for sets: $\cup, \cap, -, \ldots$. For a trace $s$, we write $I(s)$ ($O(s)$) to denote the multisets of names used in input (output) actions of $s$.

**Definition 16** For a finite trace $s$ and a multiset of messages $B$, we write $\text{exec}(s, B)$ to denote the buffer resulting from execution of trace $s$ when started with all the messages in $B$. $\text{exec}(s, B)$ is defined inductively:

1. $\text{exec}(\epsilon, B) = B$
2. $\text{exec}(as, B \cup \{a\}) = \text{exec}(s, B)$
3. $\text{exec}(as, B) = \text{undefined}$ if $a \notin B$
4. $\text{exec}(\overline{as}, B) = \text{exec}(s, B \cup \{a\})$.

By induction on the length of the first parameter, we can easily show that the following properties hold for $\text{exec}$:

- If $\text{exec}(s, B)$ is defined, then $\text{exec}(s, B) = (B \cup O(s)) - I(s)$.
- If $\text{exec}(sa\overline{as}', B)$ is defined, then $\text{exec}(sa\overline{as}', B) = \text{exec}(ss', B)$.
- If $\text{exec}(sa, B)$ is defined, then $\text{exec}(sa, B) = \text{exec}(s, B) - \{a\}$.
- If $\text{exec}(\overline{as}, B)$ is defined, then so is $\text{exec}(s, B)$.

### 4.2.1 Syntax and Semantics of $\mathcal{L}_{\text{ACCS}}^+$and $\mathcal{L}_{\text{ACCS}}^-$

We let $\phi, \phi_1, \ldots$ to range over the formulas of the modal logic $\mathcal{L}_{\text{ACCS}}^+$. The syntax of $\mathcal{L}_{\text{ACCS}}^+$ is defined as follows.

$$\phi ::= \texttt{tt} \mid \phi \land \phi \mid \phi \lor \phi \mid \langle I \rightarrow O \rangle \phi$$

where $I$ and $O$ range over multisets of actions.
The semantics of $\mathcal{L}_{\text{ACCS}}^+$ is given in terms of the relation $s, B \models \phi$ over a trace $s$, a multiset of messages $B$ (which we shall call buffer), and a $\mathcal{L}_{\text{ACCS}}^+$ formula $\phi$, is defined as:

1. $s, B \models \texttt{tt}$.

2. $s, B \models \phi_1 \land \phi_2$ iff $s, B \models \phi_1$ and $s, B \models \phi_2$.

3. $s, B \models \phi_1 \lor \phi_2$ iff $s, B \models \phi_1$ or $s, B \models \phi_2$.

4. $s, B \models \langle I \rightarrow O \rangle \phi$ iff $s = s_1s_2$, $B' = \text{exec}(s_1, B \cup I)$ is defined, $O \subseteq B'$, and $s_2, (B' - O) \models \phi$.

(1-3) are the usual rules for propositional operators and constants. Rule 4 says that, a trace $s$ satisfies $\langle I \rightarrow O \rangle \phi$ when it can be divided into two parts: the first part can provide all the messages in $O$ and in doing so it uses only messages provided in $I$ and those already in the initial buffer $B$, and the second part satisfies $\phi$ after the messages in $O$ have been consumed from the resulting buffer $B'$.

We write $s \models \phi$ for $s, \{\} \models \phi$. For a process $P$ and a formula $\phi$, we write $P \models \phi$ if there exists a trace $s$ such that $P \xrightarrow{s} \Rightarrow$ and $s \models \phi$.

**Some Laws**

The following laws hold in $\mathcal{L}_{\text{ACCS}}(\phi$ is a $\mathcal{L}_{\text{ACCS}}$ formula):

1. $\langle I \rightarrow \{\} \rangle \texttt{tt}$ is equivalent to $\texttt{tt}$. More generally, $\langle I \rightarrow \{\} \rangle \phi$ is equivalent to $\phi$.

2. $\langle I \rightarrow O \rangle (\phi_1 \land \phi_2)$ implies $\langle I \rightarrow O \rangle \phi_1 \land \langle I \rightarrow O \rangle \phi_2$.

3. $\langle I \rightarrow O \rangle \langle I' \rightarrow O' \rangle \phi$ implies $\langle I \rightarrow O \rangle \texttt{tt} \land \langle I \cup I' \rightarrow O \cup O' \rangle \phi$.

**4.2.2 A Logic of Safety: $\mathcal{L}_{\text{ACCS}}^-$**

The syntax of $\mathcal{L}_{\text{ACCS}}^-$ is defined as follows.

$$
\phi ::= \texttt{ff} \mid \phi \land \phi \mid \phi \lor \phi \mid [I \not\rightarrow O] \phi
$$

where $I$ and $O$ range over multisets of actions.
The semantics of $\mathcal{L}_{\text{ACCS}}^-$ is given in terms of the relation $s, B \models \phi$ over a trace $s$, a multiset of messages $B$ (which we shall call buffer), and a $\mathcal{L}_{\text{ACCS}}^-$ formula $\phi$. The relation $\models$ is the smallest relation that satisfies the following:

1. $s, B \models \phi_1 \land \phi_2$ iff $s, B \models \phi_1$ and $s \models \phi_2$.
2. $s, B \models \phi_1 \lor \phi_2$ iff $s, B \models \phi_1$ or $s \models \phi_2$.
3. $s, B \models [I \not\rightarrow O] \phi$ iff $s = s_1s_2$ and $B' = \text{exec}(s_1, B \cup I)$ imply $B'$ is undefined, $O \not\subseteq B'$, or $s_2, (B' - O) \not\models \phi$.

We write $s \models \phi$ for $s, \{\} \models \phi$. For a process $P$ and a $\mathcal{L}_{\text{ACCS}}^-$ formula $\phi$, we write $P \models \phi$ if for every trace $s$ such that $P \xrightarrow{s} s'$, we have $s \models \phi$.

It is easy to see that for every $\mathcal{L}_{\text{ACCS}}^+$ formula $\phi$, there exists a $\mathcal{L}_{\text{ACCS}}^-$ formula $\phi'$ such that $P \models \phi$ if and only if $P \not\models \phi'$. And for every $\mathcal{L}_{\text{ACCS}}^-$ formula $\phi$, there exists a $\mathcal{L}_{\text{ACCS}}^+$ formula $\phi'$ such that $P \models \phi$ if and only if $P \not\models \phi'$.

We now present a series of technical lemmas that demonstrate several properties of the two logics and their relationship. These lemmas will be used in the characterization theorem.

**Lemma 15** If $B \subseteq B'$ and $\phi$ is a $\mathcal{L}_{\text{ACCS}}^+$ formula then $s, B \models \phi$ implies $s, B' \models \phi$.

**Proof:** Suppose $B \subseteq B'$ and $s, B \models \phi$. We use induction on the size of $\phi$. The base case $\phi = \text{tt}$ is trivial. The induction step for the case $\phi = \phi_1 \land \phi_2$ is also trivial. Now suppose $\phi = (I \rightarrow O)\phi'$ for some $I, O$, and $\phi'$. Since $s, B \models \phi$, it follows that there exist $s_1, s_2$ such that $s = s_1s_2$, $B_1 = \text{exec}(s_1, B \cup I)$ is defined, $O \subseteq B_1$, and $s_2, (B_1 - O) \models \phi'$. $B \subseteq B'$ implies that $B \cup I \subseteq B' \cup I$ and then from $B_1 = \text{exec}(s_1, B \cup I)$ and Definition 16, it follows that $B'_1 = \text{exec}(s_1, B' \cup I)$ is defined and that $B_1 \subseteq B'_1$, from which it follows that $O \subseteq B'_1$ and (after applying the induction step) $s_2, (B'_1 - O) \models \phi'$. Therefore $s, B' \models \phi$ and the proof is complete.

**Lemma 16** For a $\mathcal{L}_{\text{ACCS}}^+$ formula $\phi$ and a multiset of names $B$, $s_1as_2, B \models \phi$ implies $s_1s_2 \models \phi$.

**Proof:** We use induction on the structure of $\phi$:

1. $\phi = \text{tt}$: $s_1s_2, B \models \phi$ trivially.
2. \( \phi = \phi_1 \land \phi_2 \): Suppose \( s_1a s_2, B \models \phi_1 \land \phi_2 \). Then \( s_1a s_2, B \models \phi_1 \) and \( s_1a s_2, B \models \phi_2 \). Using induction hypothesis, \( s_1s_2, B \models \phi_1 \) and \( s_1s_2, B \models \phi_2 \). Therefore, \( s_1s_2, B \models \phi_1 \land \phi_2 \) and the lemma follows.

3. \( \phi = (I \rightarrow O)\phi_1 \): Suppose \( s_1a s_2, B \models (I \rightarrow O)\phi_1 \). Then \( s_1a s_2 = s's'' \) for some \( s', s'' \) such that \( B' = \text{exec}(s', I \cup B) \) is defined, \( O \subseteq B' \), and \( s'', (B' - O) \models \phi_1 \). There are two cases:

(a) \( s' = s_1 a s_3 \) and \( s_2 = s_3s'' \) for some \( s_3 \): By definition if \( \text{exec}(s_1a s_3, I \cup B) \) is defined, then so is \( \text{exec}(s_1a s_3, I \cup B) \). Furthermore, \( \text{exec}(s_1a s_3, I \cup B) \subseteq \text{exec}(s_1a s_3, I \cup B) \) (dropping inputs can not decrease the number of sent messages), and consequently \( O \subseteq \text{exec}(s_1a s_3, I \cup B) \).

And since \( \text{exec}(s_1a s_3, I \cup B) - O \subseteq \text{exec}(s_1a s_3, I \cup B) - O \) and \( s'' = \text{exec}(s_1a s_3, I \cup B) - O \models \phi_1 \), by Lemma 15 we have \( s'', \text{exec}(s_1a s_3, I \cup B) - O \models \phi_1 \). Therefore, \( s_1a s_3s'', B \models (I \rightarrow O)\phi_1 \). In other words, \( s_1s_2, B \models (I \rightarrow O)\phi_1 \).

(b) \( s'' = s_3a s_2 \) and \( s_1 = s's \) for some \( s_3 \): Then by induction hypothesis \( s_3s_2, (B' - O) \models \phi_1 \) and therefore \( s's_3s_2, B \models (I \rightarrow O)\phi_1 \), or \( s_1s_2, B \models (I \rightarrow O)\phi_1 \). ∎

**Lemma 17** For a \( \mathcal{L}_{\text{ACCS}}^+ \) formula \( \phi \) and a multiset of names \( B, s_1a \overline{a}s_2, B \models \phi \) implies \( s_1s_2, B \models \phi \).

**Proof:** Again proof by induction on the structure of \( \phi \) for which there are three cases, the first two, including the base case, being similar to the previous lemma. Suppose \( \phi = (I \rightarrow O)\phi_1 \). It must be the case that \( s_1a \overline{a}s_2 = s's'' \) for some \( s' \) and \( s'' \) such that \( B' = \text{exec}(s', B \cup I) \) is defined, \( O \subseteq B' \), and \( s'', (B' - O) \models \phi \). Now, if \( a \overline{a} \) is contained in either \( s' \) or \( s'' \), then we apply the induction step and the lemma follows immediately. For the case where \( s' = s_1a \) and \( s'' = \overline{a}s_2 \), we proceed as follows. From the properties of \( \text{exec} \) it follows that \( \text{exec}(s_1, B \cup I) = B' \cup \{a\} \). It is easy to see that \( O \subseteq B' \cup \{a\} \). Finally, from \( \overline{a}s_2, (B' - O) \models \phi \), we can prove by induction on the length of \( s_2 \), that \( s_2, (B' \cup \{a\} - O) \models \phi \). This completes the induction step and we have \( s_1s_2, B \models \phi \). ∎

**Lemma 18** For a \( \mathcal{L}_{\text{ACCS}}^+ \) formula \( \phi \) and a multiset of names \( B, s_1a s_2, B \models \phi \) implies \( s_1a \overline{a}s_2, B \models \phi \).

**Proof:** Again we use induction on the structure of \( \phi \). There are three cases and the proofs of the first two is the same as the ones for Lemma 16. Now suppose \( \phi = (I \rightarrow O)\phi_1 \). Again, it must be
the case that \( s_1 \alpha s_2 = s's'' \) for some \( s' \) and \( s'' \) such that \( B' = \text{exec}(s', B \cup I) \) is defined, \( O \subseteq B' \), and \( s'', (B' - O) \models \phi. \) The rest is similar to the proof of Lemma 17. \( \square \)

The following examples show that the reverse direction of the above lemmas do not hold.
Suppose \( \phi = \langle \{ \} \rightarrow \{a\} \rangle \tt \).

- \( \bar{a} \models \phi \) but \( b \bar{a} \not\models \phi. \)
- \( \bar{a}b \models \phi \) but \( b \bar{a} \not\models \phi. \)
- \( \bar{a} \models \phi \) but \( b \bar{a} \not\models \phi. \)

**Lemma 19** For every \( \mathcal{L}^+_{\text{ACCS}} \) formula \( \phi, \) if \( s' \preceq s \) and \( s \models \phi \) then \( s' \models \phi. \)

**Proof:** Proof is by induction on \( n, \) where \( s'(\leq_0)^n s, \) and using Lemmas 16, 17, and 18. \( \square \)

**Lemma 20** For every \( \mathcal{L}^-_{\text{ACCS}} \) formula \( \phi, \) if \( s' \preceq s \) and \( s' \models \phi \) then \( s \models \phi. \)

**Proof:** Since \( \phi \) is a \( \mathcal{L}^-_{\text{ACCS}} \) formula, there exists a formula \( \phi' \) in \( \mathcal{L}^+_{\text{ACCS}} \) such that \( s \models \phi \) if and only if \( s \not\models \phi'. \) Now, suppose \( s' \preceq s, \) according to Lemma 19 \( s \models \phi' \) implies \( s' \models \phi'. \) Writing in contrapositive form, we have \( s' \not\models \phi' \) implies \( s \not\models \phi'. \) Or, equivalently, \( s' \models \phi \) implies \( s \models \phi. \) \( \square \)

The following lemma shows that for a process \( P \) to satisfy a \( \mathcal{L}^-_{\text{ACCS}} \) formula \( \phi \) it is sufficient that the finite traces of \( P \) satisfy the formula.

**Lemma 21** \( P \models \phi \) if and only if for every finite trace \( s, \) \( P \xrightarrow{s} \) implies \( s \models \phi. \)

**Proof:** We only prove the 'if' direction, the other direction is trivial. By induction on \( \phi, \) we can show that if \( P \xrightarrow{s} \) and \( s \models \phi \) for a finite \( s, \) then for any trace \( ss', \) it is the case that \( ss' \models \phi. \) From this, it follows that for every infinite trace \( s \) such that \( P \xrightarrow{s} \), we have \( s \models \phi. \) The lemma follows immediately. \( \square \)

### 4.2.3 Logical Characterization of May-Invariant Properties by \( \mathcal{L}_{\text{ACCS}} \)

The following theorem shows that properties expressed as \( \mathcal{L}_{\text{ACCS}} \) formulas are invariant under may equivalence.
Theorem 8 For processes $P$ and $Q$, if $P \cong_m Q$ then for every $L_{ACCS}^-$ formula $\phi$, $P \models \phi$ implies $Q \models \phi$.

Proof: Suppose $P \models \phi$. That is, there exists a trace $s$ such that $P \rightarrow s$ and $s \models \phi$. Since $P \cong_m Q$, From $P \rightarrow s$, it follows that there exists a trace $s' \preceq s$ such that $Q \Rightarrow s'$. Since $s \models \phi$ and $s' \preceq s$, by Lemma 19 we have $s' \models \phi$. Since $Q \Rightarrow s'$, we can conclude that $Q \models \phi$. □

The following lemmas and theorem complete the characterization.

Lemma 22 For a trace $s$, a multiset of action names $B$, and a $L_{ACCS}^+$ formula $\phi$, if $s, B \models \phi$, then $bs \models \phi$ where $b$ is trace consisting only of output actions and that $B = \{b\}$.

Proof: Proof is by induction on the size of $B$ and repeated application of the definition for exec:
$\text{exec}(\overline{a}s, B) = \text{exec}(s, B \cup \{a\})$. □

Lemma 23 For every finite trace $s$ there exists a formula $\phi_s$ such that the following hold:

1. $s \models \phi_s$.

2. Whenever $Q \Rightarrow s'$ and $s' \models \phi_s$, there exists a trace $s''$ such that $Q \Rightarrow s''$ and $s'' \preceq s$.

Proof: We define $\phi_s$ inductively as follows:

- $\phi_\epsilon = \text{tt}$

- $\phi_{ios} = (\{i\} \rightarrow \{o\})\phi_s$, where $i$ is a sequence of input actions, followed by a sequence of output actions $o$ such that both $i$ and $o$ can not be empty and that $s$ does not begin with an output action.

Part 1 is easy to show.

We prove part 2 by induction on $s$. For the base case, where $\phi_\epsilon = \text{tt}$, we let $s'' = \epsilon$ and we are done because $s'' \preceq s'$.

For the induction step, suppose $s = ios_1$ and that $\phi_{ios_1} = (I \rightarrow O)\phi_{s_1}$, and that the lemma (part 2) holds for $s_1$ and $\phi_{s_1}$. Let’s start with the assumption that $Q \Rightarrow s'$ and $s' \models \phi_s$. From $s' \models \phi_s$ we can write $s' = s'_1s'_2$, where $B = \text{exec}(s'_1, I)$, $O \subseteq B$, and $s'_2, B \models \phi_{s_1}$. Let $Q'$ be the process resulting from $Q$ after executing $s'_1$, that is, $Q \Rightarrow s'_1 \Rightarrow Q' \Rightarrow s'_2$. Now suppose $b$ is the set of extra
output actions in $s'_1$ that are not in $io$ (that is $b = \text{exec}(s'_1, \{\}) - \text{exec}(io, \{})$). Since output actions can be delayed, there must exists a trace $t$ such that $Q \xrightarrow{tb} Q'$ and that $\text{exec}(io, \{}) = \text{exec}(t, \{})$.

We show further that there exists a trace $t'$ such that $Q \xrightarrow{tb} Q'$ and $t' \preceq io$.

We already know that both $\text{exec}(io, \{})$ and $\text{exec}(t, \{})$ are defined and they are equal. If $t \preceq io$ does not hold, it must be for the following reasons:

1. $t$ has output actions that must be delayed.
2. $t$ contains additional $\overline{a}$ for some name $a$.

We can annihilate action pairs and delay outputs by applying a Lemma in [6] that relates $\preceq$ and the operational semantics of Asynchronous CCS. As a result we can obtain a trace $t'$ such that $Q \xrightarrow{t'} Q$ and $t' \preceq io$.

On the other hand, since $s'_2, B |\models \phi_{s_1}$, by Lemma 22 there exists a trace $b$ such that $bs'_2 |\models \phi_{s_1}$. Since $Q' \xrightarrow{s'_2}$ and $Q \xrightarrow{tb} Q'$, there must be a process $Q'$ such that $Q \xrightarrow{t'} Q'' \xrightarrow{bs''_2}$. By applying the induction hypothesis, we can conclude that there exists a trace $s''_2$ such that $Q'' \xrightarrow{s''_2}$ and $s''_2 \preceq s_1$.

We have $Q \xrightarrow{ts''_2}$ and since $\preceq$ is both suffix closed and prefix closed, we can deduce $ts''_2 \preceq ios''_2 \preceq ios_1 = s$. By letting $s'' = ts''_2$ the induction step is complete.

**Theorem 9** For processes $P$ and $Q$, if $P \models \phi$ implies $Q \models \phi$ for every $\mathcal{L}_{\text{ACC}}$ formula $\phi$, then $P \preceq_m Q$.

**Proof:** We need to prove that for every trace $s$ such that $P \xrightarrow{s}$, there exists a trace $s'$ such that $Q \xrightarrow{s'}$ and $s' \preceq s$.

Suppose $P \xrightarrow{s}$. Let $\phi_s$ be the formula whose existence is shown in Lemma 23. By definition, $P \models \phi_s$. This, in turn, implies that $Q \models \phi_s$. Hence, there exists a trace $s'$ such that $Q \xrightarrow{s'}$ and $s' \models \phi_s$. By Lemma 23, there exists a prefix $s''$ of $s'$ such that $s'' \preceq s$. Since $Q \xrightarrow{s'}$, it can also perform any prefix of $s'$, hence $Q \xrightarrow{s''}$, and this completes the proof.

We have shown that $P \preceq_m Q$ is equivalent to the following:

For every $\mathcal{L}_{\text{ACC}}$ formula $\phi$, $P \models \phi$ implies $Q \models \phi$.

By exploiting the duality of $\mathcal{L}_{\text{ACC}}$ and $\mathcal{L}_{\text{ACC}}^+$, we also infer that $P \preceq_m Q$ is equivalent to:
For every $\mathcal{L}_{\text{ACCS}}^+$ formula $\phi$, $Q \models \phi$ implies $P \models \phi$.

This completes our modal logic characterization of may-invariant properties.

### 4.2.4 Expressive Power

To illustrate the expressivity of the logic, let’s see how we can specify mutual exclusion of several processes sharing a critical section. Let $T$ be the following process:

$$T \overset{\text{def}}{=} NCS; \text{ENT}; (\overline{a})(CS; a.EXT; T))$$

Where $P; Q$ is the sequential composition of two processes $P$ and $Q$ and defined in the usual way [42]. The process variable $NCS$ stands for “non-critical section”, $\text{ENT}$ for “enter the critical section”, $CS$ for “critical section”, and $EXT$ for “exit the critical section”. This process performs its non-critical section part $NCS$. The process sends a message to channel $a$ immediately after entering the critical section, and reads a message from channel $a$ immediately before leaving the critical section. This guarantees that the number of message in channel $a$ is equal to the number of processes in the critical section.

Now, consider the process $TP \overset{\text{def}}{=} T \cdot \cdots \cdot T$ that consists of parallel composition of two or more $Ts$. We can express the mutual exclusion property – that no two processes enter the critical section at the same time – as the $\mathcal{L}_{\text{ACCS}}^-$ formula $[\{\} \not\rightarrow \{a,a\}]^\mathbf{ff}$. We can also express the (dual) unsafe property of two or more processes being in the critical section by the $\mathcal{L}_{\text{ACCS}}^+$ formula $\langle\{} \rightarrow \{a,a\}\rangle^\mathbf{tt}$. The composed process $TP$ satisfies the mutual exclusion property if and only if there exists no trace $s$ such that $TP \xrightarrow{s}$ and $s \models \langle\{} \rightarrow \{a,a\}\rangle^\mathbf{tt}$.

Note that the trace $\overline{a}a\overline{a}$ which is possible in an interaction with an observer which steals the message $a$ and gives it back does not satisfy the property $\langle\{} \rightarrow \{a,a\}\rangle^\mathbf{tt}$ because the second instance of $\overline{a}$ is used to annihilate the input action $a$, and hence can not be used to satisfy the requirement that two messages must exist on channel $a$ at the end of the interaction.

Please also note that that regardless of how many components $TP$ is composed of, the specification would still be the same: $[\{\} \not\rightarrow \{a,a\}]^\mathbf{ff}$. 

43
4.3 Summary and Discussion

We presented a modal logic of safety ($L^-_{\text{ACCS}}$) and a modal logic of unsafety ($L^+_{\text{ACCS}}$) for Asynchronous CCS, such that they characterize the class of properties preserved by may-preorder. By a simple example, we showed how this logic can be used for specification of properties such as mutual exclusion.

Even though the logics are proposed for Asynchronous CCS, they are independent of the actual calculus, and can be used for any calculus with asynchronous communication. All that is needed, is that the semantics captures duality of 'input' vs 'output' operations and that the order of receiving messages is undetectable.

The proposed logics are different from usual modal logics for process calculi (such as modal $\mu$-calculus) because of their ontological commitment to asynchrony. The multiset nature of actions in the modality $\langle I \rightarrow O \rangle$ is another feature that distinguishes $L_{\text{ACCS}}$ from other modal logics. Interestingly, the idea of consumption and production can be related to Linear Logic. However, the temporal nature of $L_{\text{ACCS}}$ is lacking in Linear Logic. It would be useful to investigate the relationship between the proposed logics and Linear Logic, and explore the possibility of having a “modal” Linear Logic.

More Expressive Logics

The logics $L^-_{\text{ACCS}}$ and $L^+_{\text{ACCS}}$ are expressive enough to characterize may-preserving properties. However, a modal logic for ACCS can be more expressive. An example is the linear temporal logic, which can express, but is not limited to, safety properties.

A possible candidate for a more expressive logic is one with modalities from both $L^-_{\text{ACCS}}$ and $L^+_{\text{ACCS}}$. Using this logic we can express the property that a trace produces exactly one instance of message $a$ for the critical section of the example in the previous section. The first part corresponds to the production of one or more $a$, and the second part expresses the inability of producing two or more $a$’s.

$$\langle \{} \rightarrow \{a\} \rangle \texttt{tt} \land \neg \langle \{} \rightarrow \{a,a\} \rangle \texttt{tt}$$

To express the property that messages of type $a$ are produced but no messages of other types
can be produced, we can use the following formula. We shall assume that there is a finite set of possible message types $M$.

$$\langle \{ \} \rightarrow \{ a \} \rangle \top \land \neg \bigwedge_{m \in M \setminus \{ a \}} \langle \{ \} \rightarrow \{ m \} \rangle \top$$

Note that since these formulas are interpreted over finite traces, they can only express unsafe properties. For a process to be safe with respect to a formula $\phi$, it is sufficient that the process does not have any finite trace satisfying $\phi$.

Future direction of research includes exploring a proof system for $\mathcal{L}_{\overline{\text{ACCS}}}$ and $\mathcal{L}_{\overline{\text{ACCS}}}$, decidability of the satisfiability relation, asynchronous pi-calculus, and a logic that characterizes must testing.
Chapter 5

Actors: From Lπ to Distributed Objects

5.1 Introduction

This chapter takes a step forward from a model of distributed computation to a model of distributed object based computation by introducing a new ontological commitments: *encapsulation* and *object identity*. With these notions, we obtain a calculus based formulation of the Actor model that is faithful to its standard definition in [1]. Here we introduce a basic calculus for the Actor model, called Aπ, by imposing a type system on the asynchronous version of π-calculus. This work bridges the long-standing gap between the Actor model and the process calculi tradition. We also present a trace based characterization of may testing [26] for Aπ. Both testing theories [2] and trace based models [55] have been studied for actors but the relation between them has not been investigated.

Aπ is a typed asynchronous π-calculus [9, 29, 43], where the type system enforces properties specific to the Actor model. Since the operational semantics of π-calculus is unchanged, Aπ can be seen as an embedding of the Actor model in π-calculus. This embedding not only provides a direct basis for comparison between the two models, but also enables us to apply concepts and techniques developed for π-calculus to Actors. Many formalisms for the Actor model have been proposed in the past [2, 21, 30, 51, 55] and various notions of equivalence have been considered for them [2, 21, 55]. However, none of these formalisms is directly comparable to π-calculus. On the other hand, we believe reusing a well-known formalism provides some advantages over adopting a fresh approach.

We define a labeled transition system for Aπ that reflects what is observable to an environment.
that interacts with an actor configuration. We use this transition system to derive an alternate characterization of may-testing in terms of the set of traces that an actor configuration can exhibit. The approach we adopt for establishing the characterization is similar to that used for asynchronous \(\pi\)-calculus [5]. The two characterizations differ in several respects due to differences in the notion of observability in actors and \(\pi\)-calculus, which are reflected in the labeled transition systems for the two calculi.

Due to space limitation we do not present the proofs, for which the reader is referred to [57]. We have also considered variants of A\(\pi\) that differ in the name matching capabilities in [57], and sketched the key ideas behind trace based characterizations of may testing for them. These characterizations involve radical changes to the one presented in this paper.

5.2 The Actor Model

A computational system in the Actor Model, called a configuration, consists of a collection of concurrently executing actors and a collection of messages in transit [1]. Each actor has a unique name (the uniqueness property) and a behavior, and communicates with other actors via asynchronous messages. Actors are reactive in nature, i.e. they execute only in response to messages received. An actor’s behavior is deterministic in that its response to a message is uniquely determined by the message contents. Message delivery in the Actor model is fair [11]. The delivery of a message can only be delayed for a finite but unbounded amount of time.

An actor can perform three basic actions on receiving a message: (a) create a finite number of actors with universally fresh names, (b) send a finite number of messages, and (c) assume a new behavior. Furthermore, all actions performed on receiving a message are concurrent; there is no ordering between any two of them. The following observations are in order here. First, actors are persistent in that they do not disappear after processing a message (the persistence property). Second, actors cannot be created with well known names or names received in a message (the freshness property).
5.3 The Calculus $\pi$

We assume an infinite set of names $\mathcal{N}$, and a set $\mathcal{B}$ of behavior identifiers. We let $u, v, w, x, y, z, \ldots$ range over $\mathcal{N}$, and $B$ range over $\mathcal{B}$. We write $\bar{x}$ for a tuple of names, and $\text{len}(\bar{x})$ for the length of the tuple. For $\bar{x}$ of length $n$, $x_i$ for $i \leq n$ denotes the $i^{th}$ component of the tuple. We let $C$ range over the set of preterms $\mathcal{C}$, which is defined by the following context-free grammar.

\[
C := 0 \mid x(y).C \mid \bar{x}y \mid [x = y](C_1, C_2) \mid (\nu x)C \mid C_1|C_2 \mid B(\bar{x}; \bar{y})
\]

The order of precedence of combinators is the order in which they appear. The nil term $0$, represents an empty configuration. The output term $xy$, represents a configuration with a single message targeted to $x$ and with contents $y$. We call $x$ the subject of the output term. The input term $x(y).C$ represents a configuration with an actor $x$ whose behavior is $(y)C$. We call $x$ the subject of the input term. The composition $C_1|C_2$ is a configuration containing all the actors and messages in $C_1$ and $C_2$. The conditional $[x = y](C_1, C_2)$ is $C_1$ if $x$ and $y$ are the same names, and $C_2$ otherwise. The restriction $(\nu x)C$ is the same as $C$, except that $x$ is now private to $C$. The term $B(\bar{u}; \bar{v})$ is a behavior instantiation. The identifier $B$ has a single defining equation of the form $B \overset{\text{def}}{=} (\bar{x}; \bar{y})x_1(z).C$, where $\bar{x}$ is a tuple of distinct names of length 1 or 2, and $\bar{x}, \bar{y}$ together contain exactly the free names in $x_1(z).C$. The definition provides a template for an actor behavior. For an instantiation $B(\bar{u}; \bar{v})$ we assume $\text{len}(\bar{u}) = \text{len}(\bar{x})$, and $\text{len}(\bar{v}) = \text{len}(\bar{y})$.

Notational Conventions and Definitions For a tuple $\bar{x}$, we denote the set of names occurring in $\bar{x}$ by $\{\bar{x}\}$. We write $\bar{x}, \bar{y}$ for the result of appending $\bar{y}$ to $\bar{x}$. We let $\bar{z}$ range over $\{\emptyset, \{z\}\}$. By $\bar{x}, \bar{z}$ we mean $\bar{x}, z$ if $\bar{z} = \{z\}$, and $\bar{x}$ otherwise. The term $(\bar{z})C$ is $(\nu z)C$ if $\bar{z} = \{z\}$, and $C$ otherwise. We write $(\nu x_1, \ldots, x_n)C$ instead of $(\nu x_1)(\nu x_n)C$.

The functions $fn(\cdot)$, $bn(\cdot)$, $n(\cdot)$ are defined on preterms the obvious way. Alpha equivalence on preterms, $\equiv_\alpha$, is defined as usual. We also use the usual definition and notational convention for name substitution, and let $\sigma$ range over substitutions. For a name $x$ we write $\sigma(x)$ for the name to which $x$ is mapped to by $\sigma$, and for a set of names $S$, we write $\sigma(S)$ to denote the set obtained by applying $\sigma$ to each element of $S$. Name substitutions on configurations are defined modulo alpha equivalence, with the usual renaming convention to avoid captures. We write $C\sigma$ to denote the result of applying the simultaneous substitution $\sigma$ to $C$. 

48
Let $X \subset \mathcal{N}$. We assume $\bot, \ast \notin \mathcal{N}$, and define $X^* = X \cup \{\bot, \ast\}$. For $f : X \to X^*$, we define $f^* : X^* \to X^*$ as $f^*(x) = f(x)$ for $x \in X$ and $f(\bot) = f(\ast) = \bot$. Further, if $\sigma$ is a substitution which is one-to-one on $X$, we define $f\sigma : \sigma(X) \to \sigma(X)^*$ as $f\sigma(\sigma(x)) = \sigma(f(x))$, where we let $\sigma(\bot) = \bot$ and $\sigma(\ast) = \ast$.

### 5.4 Type System

Not all preterms represent actor configurations. Unlike $\pi$-calculus where names denote communication channels, a name in the Actor model uniquely denotes a persistent agent. To capture this object paradigm we need to impose a certain discipline on the use of names, which we do using a type system. Well-typed preterms, called terms, will represent actor configurations.

Strictly enforcing all actor properties would make $\Lambda\pi$ too weak to express certain communication patterns. One such scenario is where, instead of assuming a new behavior immediately after receiving a message (as required by persistence property), an actor has to wait until certain synchronization conditions are met before processing the next message. For example, such a delaying mechanism is required to express polyadic communication, where an actor has to delay the assumption of a behavior and processing of other messages until all the arguments are transferred.

We therefore relax the persistence requirement, and allow actors to temporarily assume a series of fresh names, one at a time, and resume the old name at a later point. Basically, the synchronization task is delegated from one new name to another until the last one releases the actor after certain synchronization conditions are met.

A typing judgment is of the form $\rho; f \vdash C$, where $\rho$ is the set of free names in $C$ that denote actors in $C$, and $f : \rho \to \rho^*$ is a function that relates actors in $C$ to the temporary names they have assumed currently. Specifically, $f(x) = \bot$ means that $x$ is a regular actor name and not a temporary one, $f(x) = \ast$ means $x$ is the temporary name of an actor with a private name (bound by a restriction), and $f(x) = y \notin \{\bot, \ast\}$ means that actor $y$ has assumed the temporary name $x$. The function $f$ has the following properties: for all $x, y \in \rho$, $f(x) \neq x$, $f(x) = f(y) \notin \{\bot, \ast\}$ implies $x = y$, and $f^*(f(x)) = \bot$. While the first property is obvious, the second states that an actor cannot assume more than one temporary name at the same time, and the third states that temporary names are not like regular actor names in that they themselves cannot temporarily
assume new names but can only delegate their capability of releasing the original actor to new
names.

We define the following functions and relations that will be used in defining the type rules.

Definition 17 Let \( f_1 : \rho_1 \to \rho_1^* \) and \( f_2 : \rho_2 \to \rho_2^* \).

1. We define \( f_1 \oplus f_2 : \rho_1 \cup \rho_2 \to (\rho_1 \cup \rho_2)^* \) as

\[
(f_1 \oplus f_2)(x) = \begin{cases} 
  f_1(x) & \text{if } x \in \rho_1, \text{ and } f_1(x) \neq \bot \text{ or } x \notin \rho_2 \\
  f_2(x) & \text{otherwise}
\end{cases}
\]

Note that \( \oplus \) is associative.

2. If \( \rho \subset \rho_1 \) we define \( f|_\rho : \rho \to \rho^* \) as

\[
(f|_\rho)(x) = \begin{cases} 
  * & \text{if } f(x) \in \rho_1 - \rho \\
  f(x) & \text{otherwise}
\end{cases}
\]

3. We say \( f_1 \) and \( f_2 \) are compatible if \( f = f_1 \oplus f_2 \) has following properties: \( f = f_2 \oplus f_1 \), and for all \( x, y \in \rho_1 \cup \rho_2 \), \( f(x) \neq x \), \( f^*(f(x)) = \bot \), and \( f(x) = f(y) \notin \{ \bot, * \} \) implies \( x = y \). \( \square \)

Definition 18 For a tuple \( \tilde{x} \), we define \( ch(\tilde{x}) : \{ \tilde{x} \} \to \{ \tilde{x} \}^* \) as \( ch(\epsilon) = \{ \} \), and if \( \text{len}(x) = n \), \( ch(\tilde{x})(x_i) = x_{i+1} \) for \( 1 \leq i < n \) and \( ch(\tilde{x})(x_n) = \bot \). \( \square \)

The type rules are shown in Table 5.1. Rules \( \text{NIL} \) and \( \text{MSG} \) are obvious. In the \( \text{ACT} \) rule, if \( \hat{z} = \{ z \} \) then actor \( z \) has assumed temporary name \( x \). The condition \( y \notin \rho \) ensures that actors are not created with names received in a message. In the terminology of [48], only output capability of names can be passed in messages. The conditions \( y \notin \rho \) and \( \rho - \{ x \} = \hat{z} \) together guarantee the freshness property by ensuring that new actors are created with fresh names. Note that it is possible for \( x \) to be a regular name, i.e. \( \rho - \{ x \} = \emptyset \), and disappear after receiving a message, i.e. \( x \notin \rho \). We interpret this as the actor \( x \) assuming a \( \text{Sink} \) behavior that simply consumes all messages it receives. With this interpretation the persistence property is not violated.

The compatibility check in \( \text{COND} \) rule prevents errors such as two actors, each in a different branch, assuming the same temporary name, or the same actor assuming different temporary names in different branches. The \( \text{COMP} \) rule guarantees the uniqueness property by ensuring that the
two composed configurations do not contain actors with the same name. In the RES rule, \( f \) is updated so that if \( x \) has assumed a temporary name \( y \) in \( C \), then \( y \)'s role as a temporary name is remembered but \( x \) is forgotten. The INST rule assumes that if \( \text{len}(\bar{x}) = 2 \) then \( B(\bar{x}; \bar{y}) \) denotes an actor \( x_2 \) that has assumed temporary name \( x_1 \).

Type checking a preterm involves checking the accompanying behavior definitions. For INST rule to be sound, for every definition \( B \overset{\text{def}}{=} (\bar{x}; \bar{y})x_1(z).C \) and substitution \( \sigma = \{ \bar{u}, \bar{v}/\bar{x}, \bar{y} \} \) that is one-to-one on \( \{ \bar{x} \} \), the judgment \( \{ \bar{u} \}; \text{ch}(\bar{u}) \vdash (x_1(z).C)\sigma \) should be derivable. From Lemma 25, it follows that this constraint is satisfied if \( \{ \bar{x} \}; \text{ch}(\bar{x}) \vdash x_1(z).C \) is derivable. Thus, a preterm is well-typed only if for each accompanying behavior definition \( B \overset{\text{def}}{=} (\bar{x}; \bar{y})x_1(z).C \), the judgment \( \{ \bar{x} \}; \text{ch}(\bar{x}) \vdash x_1(z).C \) is derivable.

The following lemma states a soundness property of the type system.

**Lemma 24** If \( \rho; f \vdash C \) then \( \rho \subseteq \text{fn}(C) \), and for all \( x, y \in \rho \), \( f(x) \neq x \), \( f^*(f(x)) = \bot \), and \( f(x) = f(y) \notin \{ \bot, * \} \) implies \( x = y \). Further, if \( \rho'; f' \vdash C \) then \( \rho = \rho' \) and \( f = f' \).

**Proof:** By structural induction on \( C \). \( \square \)

Not all substitutions on a term \( C \) yield terms. A substitution \( \sigma \) may identify distinct actor names in \( C \) and therefore violate the uniqueness property. But, if \( \sigma \) renames different actors in \( C \)
Lemma 25 If \( \rho; f \vdash C \) and \( \sigma \) is one-to-one on \( \rho \) then \( \sigma(\rho); f \sigma \vdash C \sigma \).

Proof: Since the type system respects alpha equivalence, without loss of generality, we may assume the hygiene condition that \( \sigma(x) = x \) for all \( x \in bn(C) \), and \( bn(C) \cap \sigma(fn(C)) = \emptyset \).

The proof is by induction on the length of a derivation of \( \rho; f \vdash C \). It is straightforward to verify the base cases where the derivation is a direct application of \textsc{nil}, \textsc{msg} or \textsc{inst}. For the induction step, we consider only two cases; the others are simple.

1. \( C = x(y).C' \): Then the last step of derivation is

\[
ACT: \quad f'; f \vdash C' \quad \frac{}{\{x\} \cup \hat{z}; ch(x, \hat{z}) \vdash x(y).C'}
\]

if \( f' = \begin{cases} 
ch(x, \hat{z}) & \text{if } x \in \rho' \\
ch(\epsilon, \hat{z}) & \text{otherwise}
\end{cases} \)

\( \rho' - \{x\} = \hat{z}, y \notin \rho' \), and

Note that by hygiene condition \( C\sigma = \sigma(x)(y).(C'\sigma) \). Since \( \sigma \) is an injection on \( \{x\} \cup \hat{z} \) so it is on \( \rho' \), and thus by induction hypothesis \( \sigma(\rho') ; f'\sigma \vdash C'\sigma \). This, together with \( \rho' - \{x\} = \hat{z} \), also implies \( \sigma(\rho') - \sigma(\{x\}) = \sigma(\hat{z}) \). By Lemma 24, we have \( \rho' \subseteq fn(C') \), and hence by the hygiene condition \( y \notin \sigma(\rho') \). Since \( \sigma \) is an injection on \( \{x\} \cup \hat{z} \) and \( \rho' \subseteq \{x\} \cup \hat{z} \), we have \( f'\sigma = ch(x, \hat{z})\sigma = ch(\sigma(x), \sigma(\hat{z})) \) if \( \sigma(x) \in \sigma(\rho') \), and \( ch(\epsilon, \hat{z})\sigma = ch(\epsilon, \sigma(\hat{z})) \) otherwise. We can now apply the \textsc{act} rule to get

\( \sigma(\{x\}) \cup \sigma(\hat{z}); ch(\sigma(x), \sigma(\hat{z})) \vdash \sigma(x)(y).(C'\sigma) \), i.e.

\( \sigma(\{x\}) \cup \hat{z}; ch(x, \hat{z})\sigma \vdash \sigma(x)(y).(C'\sigma) \).

2. \( C = [x = y](C_1, C_2) \): Then the last step of the derivation is

\[
COND: \quad f_1; f_1 \vdash C_1 \quad f_2; f_2 \vdash C_2 \quad \frac{}{f_1 \cup f_2; f_1 \oplus f_2 \vdash [x = y](C_1, C_2)}
\]

if \( f_1 \) and \( f_2 \) are compatible

Since \( \sigma \) is an injection on \( \rho_1 \cup \rho_2 \), so it is on \( \rho_1 \) and \( \rho_2 \). Then, by induction hypothesis \( \sigma(\rho_1); f_1\sigma \vdash C_1\sigma \) and \( \sigma(\rho_2); f_2\sigma \vdash C_2\sigma \). The reader may verify that the facts \( f_1 \) and \( f_2 \) are compatible, and \( \sigma \) is an injection on \( \rho_1 \cup \rho_2 \), together imply \( f_1\sigma \) and \( f_2\sigma \) are compatible. We can now apply the \textsc{cond} rule to get

\( \sigma(\rho_1) \cup \sigma(\rho_2); f_1\sigma \oplus f_2\sigma \vdash [\sigma(x) = \sigma(y)](C_1\sigma|C_2\sigma) \).
result follows from the following fact that $f_1 \sigma \oplus f_2 \sigma = (f_1 \oplus f_2) \sigma$, which can be verified easily.

\[ \square \]

### 5.5 Reduction Semantics

Reduction semantics of $A\pi$ is the same as that of $\pi$-calculus with mismatch. It is defined in terms of the usual structural congruence over preterms and reduction rules shown in Definition 19 and Table 5.2. We use $\implies$ to denote the reflexive transitive closure of $\to$.

**Definition 19 (structural congruence)** The relation $\equiv$ is the smallest congruence relation on preterms closed under the following laws:

1. If $C_1 \equiv_\alpha C_2$ then $C_1 \equiv C_2$.
2. The combinator, $\mid$, is commutative and associative with 0 as identity.
3. $(\nu x, y) C \equiv (\nu y, x) C$, $(\nu x) 0 \equiv 0$.
4. If $x \notin fn(C_2)$ then $(\nu x) C_1 | C_2 \equiv (\nu x) (C_1 | C_2)$.
5. If $B \overset{\text{def}}{=} (\bar{x}; \bar{y}) x_1(z). C$, $\text{len}(\bar{u}) = \text{len}(\bar{x})$, and $\text{len}(\bar{v}) = \text{len}(\bar{y})$ then

$$B(\bar{u}; \bar{v}) \equiv (x_1(z). C)\{((\bar{u}, \bar{v})/(\bar{x}, \bar{y}))\}.$$

\[ \square \]

<table>
<thead>
<tr>
<th>Table 5.2: Reduction rules for $A\pi$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RECV:</strong> $x(y). C</td>
</tr>
<tr>
<td><strong>IF:</strong> $[x = x](C_1, C_2) \to C_1$</td>
</tr>
<tr>
<td><strong>HIDE:</strong> $C \to C'$</td>
</tr>
<tr>
<td>$(\nu x) C \to (\nu x) C'$</td>
</tr>
<tr>
<td><strong>REQV:</strong> $\frac{C_1 \equiv C_1'}{C_1' \to C_2'}$ if $C_1 \equiv C_1'$</td>
</tr>
</tbody>
</table>

Lemma 26 and Theorem 10 state that type system respects both the structural congruence and reduction rules.

**Lemma 26** Let $C_1 \equiv C_2$. Then $\text{fn}(C_1) = \text{fn}(C_2)$, and $\rho; f \vdash C_1$ if and only if $\rho; f \vdash C_2$.

53
**Theorem 10 (subject reduction 1)** Let \( \rho; f \vdash C \) and \( C \rightarrow C' \). Then \( \rho'; f' \vdash C' \), for some \( \rho' \subset \rho \), and \( f' : \rho' \rightarrow \rho'^* \) satisfying the following conditions: \( f'(x) = \bot \) if \( f(x) = \bot \), \( f'(x) \in \{ f(x), \bot \} \) otherwise.
Proof: The proof is by induction on length of a derivation of $C \rightarrow C'$. There are three base cases, depending on whether the derivation is a direct application of $RECV$, $IF$ or $ELSE$ rules. We only consider the $RECV$ and $ELSE$ cases.

1. $RECV$: Then we have $C = x(y).C_1|\tau z$ and $C' = C_1\{z/y\}$, for some $x,y,z,C_1$. Then by $ACT$, $MSG$, and $COMP$ rules, for some $\rho_1,f_1$ we have $\rho_1; f_1 \vdash C_1$, $y \notin \rho_1$, $\rho = \rho_1 \cup \{x\}$, and $f_1 = f|\rho_1$. Then $\sigma = \{z/y\}$ is an injection on $\rho_1$, and by Lemma 25 it follows that $\sigma(\rho_1); f_1\sigma \vdash C_1\sigma$. But since $y \notin \rho_1$, we have $\sigma(\rho_1) = \rho_1$, $f_1\sigma = f_1$. Thus, we have $\rho_1; f_1 \vdash C'$, and the theorem follows from the fact that $\rho_1 \subset \rho$, and $f_1(x) = f(x)$ for $x \in \rho_1$.

2. $ELSE$: For some $x,C_1,C_2$, we have $C = [x = x](C_1,C_2)$ and $C' = C_2$. By $COND$ rule, $\rho_1; f_1 \vdash C_1$, $\rho_2; f_2 \vdash C_2$, and $\rho = \rho_1 \cup \rho_2$, $f = f_1 \oplus f_2$, and $f_1,f_2$ are compatible. From the definition of $\oplus$ it is clear that $f_2(y) = \bot$ if $f(y) = \bot$. Since, $f_1,f_2$ are compatible, we have $f_2(y) \in \{f(y), \bot\}$ if $f(y) \neq \bot$, because otherwise $(f_1 \oplus f_2)(y) \neq (f_2 \oplus f_1)(y)$, and $f_1,f_2$ will not be compatible. The theorem thus follows.

For the induction step, there are three cases depending on which rule is used in the last step of the derivation.

1. $HIDE$: For some $x,C_1,C_1'$, we have $C = (\nu x)C_1$ and $C' = (\nu x)C_1'$ and the last derivation step is

$$HIDE: \quad \frac{C_1 \rightarrow C_1'}{(\nu x)C_1 \rightarrow (\nu x)C_1'}$$

By $RES$ rule, for some $\rho_1,f_1$, we have $\rho_1; f_1 \vdash C_1$, $\rho = \rho_1 - \{x\}$, and $f = f_1|\rho$. By induction hypothesis, $\rho_1'; f_1' \vdash C_1'$ for some $\rho_1' \subset \rho_1$, $f_1'$. Then by $RES$ rule $\rho_1' - \{x\}; f_1'|\{\rho_1' - \{x\}\} \vdash (\nu x)C_1'$. Note that $\rho_1' - \{x\} \subset \rho_1 - \{x\} = \rho$. Now, the result follows if we show that $f_1'|\{\rho_1' - \{x\}\}$ satisfies the conditions in theorem statement. For $y \in \rho_1' - \{x\}$ we consider only the case where $f(y) = *$; the other cases where $f(y) = \bot$ and $f(y) \in \rho$ are easier. Since $f = f_1|\{\rho_1 - \{x\}\}$, it follows that $f_1(y) \in \{x,*\}$. We consider the case where $f_1(y) = x$ and leave the other easier case to the reader. By induction hypothesis, $f_1'(y) \in \{x,\bot\}$, and hence $(f_1'|\{\rho_1' - x\})(y) \in \{*,\bot\}$, and the theorem follows.
2. \textit{PAR}: For some $C_1, C'_1, C_2$, we have $C = C_1|C_2$, $C' = C'_1|C_2$ and the last derivation step is

\[
\begin{array}{c}
\text{PAR:} \\
C_1 \rightarrow C'_1 \\
C_1|C_2 \rightarrow C'_1|C_2
\end{array}
\]

By \textit{COMP} rule, for some $\rho_1, \rho_2, f_1, f_2$, we have $\rho_1; f_1 \vdash C_1$, $\rho_2; f_2 \vdash C_2$, $\rho_1 \cap \rho_2 = \emptyset$, $\rho = \rho_1 \cup \rho_2$, and $f = f_1 \oplus f_2$. By induction hypothesis, we have $\rho'_1; f'_1 \vdash C'_1$ for some $\rho'_1 \subset \rho_1$, and $f'_1$. Then $\rho'_1 \cap \rho_2 = \emptyset$, and by \textit{COMP} rule it follows that $\rho'_1 \cup \rho_2 \vdash f'_1 \oplus f_2 \vdash C'_1|C_2$. Note that $\rho'_1 \cup \rho_2 \subset \rho$.

Now the result follows if we show that $f'_1 \oplus f_2$ satisfies the required conditions. For $y \in \rho'_1 \cup \rho_2$, we consider only the case where $f(y) = x \in \rho$, and leave the other similar cases to the reader.

If $y \in \rho_1 - \rho_2$ or $y \in \rho_2 - \rho_1$ the argument is simple. We consider only the case $y \in \rho_1 \cap \rho_2$. Since $f_1$ and $f_2$ are compatible, we have the following possible cases: $f_1(y) = f_2(y) = x$, or $f_1(y) = x, f_2(y) = \bot$, or $f_1(y) = \bot, f_2(y) = x$. We consider only the first. Since $f_1(y) = x$, by induction hypothesis it follows that $f'_1(y) \in \{x, \bot\}$. In any case we have $(f'_1 \oplus f_2)(y) = x$ which satisfies the condition. The theorem thus follows.

3. \textit{REQV}: The result follows by a simple application of Lemma 26. \hfill $\Box$

Since well-typed terms are closed under reduction, it follows that actor properties are preserved during a computation. However, note that the source and the target of a transition need not have the same typing judgment. This is because of two reasons. First, actors may disappear. As the reader may recall, this is interpreted as the actor assuming a sink behavior. Second, an actor with a temporary name may re-assume its original name, or decide to never assume it.

We show how the ability to temporarily assume a fresh name can be used to encode polyadic communication in $A\pi$. We assume that the subject of a polyadic receive is not a temporary name. In particular, in the encoding below, $x$ cannot be a temporary name. The idea behind translation is to let $x$ temporarily assume a fresh name $z$ which is used to receive all the arguments without
any interference from other messages, and re-assume $x$ after the receipt. For fresh $u, z$ we have

\[
\begin{align*}
[\overline{\pi}(y_1, \ldots, y_n)] & = (\nu u)(\overline{\pi} u \mid S_1(u; y_1, \ldots, y_n)) \\
S_i & \overset{\text{def}}{=} (u; y_i, \ldots, y_n) u(z)(\overline{\pi} y_i \mid S_{i+1}(u; y_{i+1}, \ldots, y_n)) & 1 \leq i < n \\
S_n & \overset{\text{def}}{=} (u; y_n) u(z).\overline{\pi} y_n \\
[x(y_1, \ldots, y_n).C] & = x(u).(\nu z)(\overline{\pi} z \mid R_1(z, \hat{x}; u, \tilde{a})) \\
R_i & \overset{\text{def}}{=} (z, \hat{x}; u, \tilde{a}) z(y_i)(\overline{\pi} z \mid R_{i+1}(z, \hat{x}; u, \tilde{a})) & 1 \leq i < n \\
R_n & \overset{\text{def}}{=} (z, \hat{x}; u, \tilde{a}) z(y_n)(\overline{\pi} z \mid [C])
\end{align*}
\]

where $\tilde{a} = fn(x(y_1, \ldots, y_n).C) - \{x\}$, and $\hat{x} = \{x\}$ if for some $\rho, f$, we have $\rho \cup \{x\}; f \vdash [C]$, and $\hat{x} = \emptyset$ otherwise.

The formalism thus far does not account for fairness in message deliveries that is required by the Actor model. We do not consider fairness, as it does not make a difference to the may testing theory we are concerned with. The reader is referred to Section 3.5 for further discussion about this.

### 5.6 May Testing

We now instantiate the general notion of may testing [26] on $A\pi$. As in any typed calculus, testing in $A\pi$ takes typing into account; an observer $O$ can be used to test $C$ only if $C|O$ is well typed. Since the set of valid tests varies between configurations, we parameterize the may preorder with the set of observers that is used to decide the order.

**Definition 20 (may testing)** Observers are actor configurations that can emit a special message $\overline{\pi} \mu$. We let $O$ range over the set of observers. For $C, O$ such that $C|O$ is well-typed, we say $C \overline{\mu} O$ if $C|O \Rightarrow C'|\overline{\pi} \mu$ for some $C'$. Let $\rho_1; f_1 \vdash C_1$ and $\rho_2; f_2 \vdash C_2$. Then for $\rho$ such that $\rho_1, \rho_2 \subset \rho$ we say $C_1 \overset{\rho}{\leq}_C C_2$ if for every $O$ such that $\rho'; f' \vdash O$ and $\rho' \cap \rho = \emptyset$, $C_1 \overline{\mu} O$ implies $C_2 \overline{\mu} O$. We say $C_1 \overset{\rho}{\leq}_C C_2$ if $C_1 \overset{\rho}{\leq}_C C_2$ and $C_2 \overset{\rho}{\leq}_C C_1$. Note that $\overset{\rho}{\leq}_C$ is reflexive and transitive, and $\overset{\rho}{\sim}_C$ is an equivalence relation. □
The parameter of a preorder indicates the size of the observer set that is used to decide the order; the larger the parameter, the smaller the observer set. From this observation, it is easy to see that when \( \rho_1 \subseteq \rho_2 \), we have \( C_1 \preceq_{\rho_1} C_2 \) implies \( C_1 \preceq_{\rho_2} C_2 \), but not the converse. To see why the converse doesn’t hold, we have \( x \sim_{\{x\}} \bar{y} \), but only \( x \not\sim_{\emptyset} \bar{y} \) and \( \bar{y} \not\sim_{\emptyset} 0 \). Similarly, \( \bar{y} \sim_{\{x,y\}} \bar{y} \), but \( \bar{y} \not\sim_{\emptyset} \bar{y} \) and \( \bar{y} \not\sim_{\emptyset} \bar{x} \). However, the converse holds if \( \text{fn}(C_1) \cup \text{fn}(C_2) \subset \rho_1 \).

**Lemma 27** Let \((\nu x)C_1 \longrightarrow C_2\). Then \( C_2 \equiv (\nu x)C'_2 \) for some \( C'_2 \) such that \( C_1 \longrightarrow C'_2 \).

**Proof:** By induction on the length of a derivation of \((\nu x)C_1 \longrightarrow C_2\). \(\square\)

**Theorem 11** Let \( \rho_1 \subset \rho_2 \). Then \( C_1 \preceq_{\rho_1} C_2 \) implies \( C_1 \preceq_{\rho_2} C_2 \). Further, if \( \text{fn}(C_1) \cup \text{fn}(C_2) \subset \rho_1 \) then \( C_1 \preceq_{\rho_2} C_2 \) implies \( C_1 \preceq_{\rho_1} C_2 \).

**Proof:** Let \( C_1 \preceq_{\rho_1} C_2 \). Suppose \( \rho; f \vdash O, \rho \cap \rho_2 = \emptyset \), and \( C_1 \) may \( O \). Since \( \rho_1 \subset \rho_2 \), we have \( \rho \cap \rho_1 = \emptyset \). Then since \( C_1 \preceq_{\rho_1} C_2 \), we have \( C_2 \) may \( O \). Hence \( C_1 \preceq_{\rho_2} C_2 \).

Let \( \text{fn}(C_1) \cup \text{fn}(C_2) \subset \rho_1 \) and \( C_1 \preceq_{\rho_2} C_2 \). For \( O \) such that \( \rho; f \vdash O, \rho \cap \rho_1 = \emptyset \), let \( C_1 \) may \( O \). We have to show \( C_2 \) may \( O \). Now, for \( \bar{x} \) such that \( \{\bar{x}\} = \rho \), we have \( \emptyset; \{\} \vdash (\nu \bar{x})O, \text{fn}(C_1) \cap \{\bar{x}\} = \emptyset \), and \( \text{fn}(C_2) \cap \{\bar{x}\} = \emptyset \). Then \( C_1|O \implies C'|\bar{\mu} \) implies \( C_1|(\nu \bar{x})O \equiv (\nu \bar{x})(C_1|O) \implies (\nu \bar{x})(C'|\bar{\mu}) \equiv (\nu \bar{x})C'|\bar{\mu} \). Hence \( C_1 \) may \( (\nu \bar{x})O \). Now, since \( C_1 \preceq_{\rho_2} C_2 \) we have \( C_2 \) may \( (\nu \bar{x})O \). Since \( \text{fn}(C_2) \cap \{\bar{x}\} = \emptyset \), we have \( (\nu \bar{x})(C_2|O) \equiv C_2|(\nu \bar{x})O \implies C_3|\bar{\mu}, \) for some \( C_3 \). From Lemma 27, we deduce \( C_2|O \implies C_4 \) such that \( (\nu \bar{x})C_4 \equiv C_3|\bar{\mu} \). From this we deduce that \( C_4 \equiv C_5|\bar{\mu} \) for some \( C_5 \). It follows that \( C_2 \) may \( O \). \(\square\)

## 5.7 Labeled Transition System

We give an alternate characterization of may testing which does not involve quantification over observing contexts. The characterization is trace-based, i.e. it is in terms of sequences of observable actions, namely the message exchanges, that a configuration may perform while interacting with its environment.

The set of possible message exchanges at any time is determined by the current ownership of names, i.e. which names denote actors in the configuration and which those in the environment.

The configuration can input only messages targeted to one of its actors that is not hidden from the
environment (a receptionist), and can emit only messages targeted to an actor in the environment (an external actor). We call this the *encapsulation* property. Note that the information about ownership of names is in general not contained in the syntax of a configuration as internal actors may disappear (assume sink behavior) and external names may be forgotten as the configuration evolves. We therefore define the notion of a configuration interface that records the history of ownership of names, and use it to define a labeled transition system that characterizes observable actions.

**Definition 21 (interfaces)** An interface is a pair of sets of names written as $[\rho, \chi]$, where $\rho \cap \chi = \emptyset$. We let $I$ range over interfaces. We define an ordering on interfaces as $[\rho_1, \chi_1] \leq [\rho_2, \chi_2]$ if $\rho_1 \subseteq \rho_2$ and $\chi_1 \subseteq \chi_2 \cup \rho_2$.

**Lemma 28** The relation $\leq$ on interfaces is a partial order.

**Proof of Lemma 28**: Reflexivity and transitivity of $\leq$ is immediate from Definition 21. Let $[\rho_1, \chi_1] \leq [\rho_2, \chi_2]$ and $[\rho_2, \chi_2] \leq [\rho_1, \chi_1]$. Since $\rho_1 \subseteq \rho_2$ and $\rho_2 \subseteq \rho_1$ we have $\rho_1 = \rho_2$. From $\chi_1 \subseteq \rho_2 \cup \chi_2$, $\rho_1 = \rho_2$, and $\chi_1 \cap \rho_1 = \emptyset$, it follows that $\chi_1 \subseteq \chi_2$. By a similar argument $\chi_2 \subseteq \chi_1$, and hence $\chi_1 = \chi_2$. So $\leq$ is antisymmetric, and thus a partial order.

We associate a configuration with interface $[\rho, \chi]$ to mean that names in $\rho$ denote receptionists of the configuration and those in $\chi$ denote its external actors. Thus, the configuration can input only messages with target in $\rho$ and emit messages with target in $\chi$. Note that since the computational history of a configuration is not contained in its syntax, a configuration can have several possible interfaces. The idea behind partial order on interfaces is that if $I_1 \leq I_2$ and $I_1$ is a possible interface of a configuration then so is $I_2$.

**Definition 22** Let $\rho; f \vdash C$, and $\chi = \text{fn}(C) - \rho$. Then we say $[\rho', \chi']$ is a possible interface of $C$ and write $C : [\rho', \chi']$ if $[\rho, \chi] \leq [\rho', \chi']$. We call $[\rho, \chi]$ the minimal interface of $C$.

**Remark**: Note that as a direct consequence of Lemma 26, if $C_1 \equiv C_2$ then $C_1 : [\rho, \chi]$ if and only if $C_2 : [\rho, \chi]$.

We define labeled transitions over configurations with interfaces, which are written as $\llangle C \rrangle^\rho_\chi$. We say $\llangle C \rrangle^\rho_\chi$ is well-formed if and only if $C : [\rho, \chi]$. The transition rules are given in Table 5.3.
Transition labels can be of five forms: \( \tau \) (a silent action), \( \overline{xy} \) (free output of a message with target \( x \) and content \( y \)), \( x(y) \) (bound output), \( xy \) (free input of a message) and \( x(y) \) (bound input). We denote the set of all visible actions (non-\( \tau \)) actions by \( L \), and let \( \alpha \) range over \( L \). The functions \( fn(\cdot) \), \( bn(\cdot) \) and \( n(\cdot) \) are defined on \( L \) the usual way. To have a convenient uniform notation for free and bound actions we use the following convention: \((\emptyset)xy = xy\), \((\{y\})xy = \overline{xy}\), and similarly for input actions. We define a complementation function on \( L \) as \((\hat{y})xy = (\hat{y})\overline{xy}\), \((\hat{y})xy = (\hat{y})\overline{xy}\).

Table 5.3: Labelled transition system for A\( \pi \)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IN:</strong></td>
<td>[C]^{\rho}<em>x \xrightarrow{(\hat{y})xy} [C \mid \overline{xy}]^{\rho}</em>{x \cup {y} \cup \rho} \quad \text{if } \hat{y} \cap (\rho \cup \chi) = \emptyset, \ x \in \rho</td>
<td></td>
</tr>
<tr>
<td><strong>OUT:</strong></td>
<td>[C]^{\rho}<em>x \xrightarrow{(y)\overline{xy}} [C \mid \overline{xy}]^{\rho}</em>{\hat{y} \cup \chi} \quad \text{if } \hat{y} \cap (\rho \cup \chi) = \emptyset, \ x \in \chi</td>
<td></td>
</tr>
<tr>
<td><strong>TAU:</strong></td>
<td>[C \mid \overline{C}]^{\rho}<em>{x} \xrightarrow{\tau} [C' \mid \overline{C'}]^{\rho}</em>{x}</td>
<td></td>
</tr>
<tr>
<td><strong>LEQV:</strong></td>
<td>[C_1]^{\rho}_x \xrightarrow{\alpha} [C_2]^{\rho}_x \quad \text{if } C_1 \equiv C'_1 \ \ x \in \chi</td>
<td></td>
</tr>
</tbody>
</table>

The labeled transition system is essentially a simple extension of the reduction system to include observable actions. The \( IN \) and \( OUT \) rules together capture the encapsulation property we described earlier. The \( IN \) rule states that a configuration can receive only a message targeted to one of its receptionists. The message is asynchronous and is added to the pool of messages in the configuration. The external actor set of the interface may expand as the received message may contain new external actor names. The \( OUT \) rule states that only messages targeted to an external actor can leave the configuration. The receptionist set may expand because names of hidden actors can be exported in the message. Note that fresh names are chosen for these new receptionists so that uniqueness property is preserved.

Lemma 29 states that the transition system is consistent with the reduction system. Theorem 12 states the soundness of the transition system and also characterizes the evolution of configuration interfaces.

**Lemma 29** If \( C : [\rho, \chi] \) then \( C \longrightarrow C' \) if and only if \[C\]^{\rho}_x \xrightarrow{\tau} \[C'\]^{\rho}_{x}.

**Theorem 12 (subject reduction 2)** If \( C_1 : [\rho_1, \chi_1] \) and \[C_1\]^{\rho}_x \xrightarrow{\alpha} \[C_2\]^{\rho}_x then \( C_2 : [\rho_2, \chi_2] \), \( \rho_1 \subset \rho_2 \) and \( \chi_1 \subset \chi_2 \).

**Remark:** Note that \([\rho_1, \chi_1] \leq [\rho_2, \chi_2] \).
Proof: All the transition rules in Table 5.3 monotonically increase \( \rho \) and \( \chi \). Therefore \( \rho_1 \subseteq \rho_2 \) and \( \chi_1 \subseteq \chi_2 \). We show \( C_2 : [\rho_2, \chi_2] \) by induction on the length of a derivation of a \( \langle C_1 \rangle_{\chi_1}^{\rho_1} \stackrel{\alpha}{\longrightarrow} \langle C_2 \rangle_{\chi_2}^{\rho_2} \).

We are given that \( \rho ; f \vdash C_1 \) and \( [\rho, fn(C_1) - \rho] \subseteq [\rho_1, \chi_1] \).

There are two base cases. The derivation is by a direct application of

1. **IN**: We have

\[
\text{IN: \quad } \langle C_1 \rangle_{\chi_1}^{\rho_1} \stackrel{\{y\}x}{\longrightarrow} \langle C_1 \mid xy \rangle_{(\chi_1 \cup \{y\}) - \rho_1}^{\rho_1}
\]

where \( \hat{y} \cap (\rho_1 \cup \chi_1) = \emptyset \), \( x \in \rho_1 \), \( C_2 = C_1 \mid xy \), \( \rho_2 = \rho_1 \), and \( \chi_2 = (\chi_1 \cup \{y\}) - \rho_1 \).

Then by MSG and COMP rules we have \( \rho ; f \vdash C_2 \). Also, since \( fn(C_1) \subseteq \rho_1 \cup \chi_1 \), we have \( fn(C_2) = fn(C_1) \cup \{x, y\} \subseteq \rho_1 \cup \chi_1 \cup \{x, y\} = \rho_2 \cup \chi_2 \). The last equality is because \( x \in \rho_1 \).

Thus, \( C_2 : [\rho_2, \chi_2] \).

2. **OUT**: We have \( C_1 = (\nu y)(C_2 | xy) \),

\[
\text{OUT: \quad } \langle (\nu y)(C_2 | xy) \rangle_{\chi_1}^{\rho_1} \stackrel{\{y\}x}{\longrightarrow} \langle C_2 \rangle_{\chi_1}^{\rho_1 \cup \hat{y}}
\]

\( \rho_2 = \rho_1 \cup \hat{y} \), \( \chi_2 = \chi_1 \), where \( \hat{y} \cap (\rho_1 \cup \chi_1) = \emptyset \), and \( x \in \chi_1 \). By MSG, COMP, and RES rules, it follows that \( \rho' ; f' \vdash C_2 \) where \( \rho' \subseteq \rho \cup \hat{y} \) and \( f = f' \mid \rho \). Since \( \rho \subseteq \rho_1 \), we have \( \rho' \subseteq \rho_1 \cup \hat{y} = \rho_2 \).

Also, \( fn(C_2) \subseteq fn(C_1) \cup \hat{y} \). Since \( fn(C_1) \subseteq \rho_1 \cup \chi_1 \), we have \( fn(C_2) \subseteq \rho_1 \cup \hat{y} \cup \chi_1 = \rho_2 \cup \chi_2 \).

Thus, \( C_2 : [\rho_2, \chi_2] \).

For the induction step there are two cases depending on the rule used for the last derivation step.

1. **TAU**: We have \( \rho_1 = \rho_2, \chi_1 = \chi_2 \) and

\[
\text{TAU: } \quad C_1 \rightarrow C_2 \quad \frac{C_1}{\langle C_1 \rangle_{\chi_1}^{\rho_1} \stackrel{r}{\longrightarrow} \langle C_2 \rangle_{\chi_1}^{\rho_1}}
\]

By Theorem 10, we have \( \rho' ; f' \vdash C_2 \) for some \( \rho' \subseteq \rho \). Hence \( \rho' \subseteq \rho \subseteq \rho_1 \). Further, it is easy to show that free names in the target of a transition also occur free in the source. So \( fn(C_2) \subseteq fn(C_1) \). Now, since \( fn(C_1) \subseteq \rho_1 \cup \chi_1 \), we have \( fn(C_2) \subseteq \rho_1 \cup \chi_1 \). Hence, \( fn(C_2) - \rho' \subseteq \rho_1 \cup \chi_1 \). We thus have \( C_2 : [\rho_1, \chi_1] \).

61
We let $s, r, t$ range over $L^*$. For $s = \alpha_1 \ldots \alpha_i \ldots \alpha_n$, we define $\text{len}(s) = n$, and $s(i) = \alpha_i$, for $1 \leq i \leq \text{len}(s)$. The functions $\text{fn}(.), \text{bn}(.)$ and $n(.)$ are defined on $L^*$ the obvious way. The complementation function on $L$ is extended to $L^*$ the obvious way. We use $\rightarrow$ to denote the reflexive transitive closure of $\rightarrow$, and $\rightarrow^\alpha$ to denote $\rightarrow^\alpha \rightarrow \rightarrow$. Note that $\rightarrow$ is overloaded to denote both sequences of reductions and $\rightarrow^\alpha \rightarrow$ transitions, but its context of use will always clarify which one is being used. For $s = l.s'$ we use $\llangle C_1 \rrangle^{\rho_1}_{\chi_1} \triangleright s \llangle C_2 \rrangle^{\rho_2}_{\chi_2}$ to denote $\llangle C_1 \rrangle^{\rho_1}_{\chi_1} \overset{1}{\rightarrow} \llangle C_2 \rrangle^{\rho_2}_{\chi_2}$, and similarly $\llangle C_1 \rrangle^{\rho_1}_{\chi_1} \triangleright s \llangle C_2 \rrangle^{\rho_2}_{\chi_2}$ to denote $\llangle C_1 \rrangle^{\rho_1}_{\chi_1} \overset{1}{\rightarrow} \llangle C_2 \rrangle^{\rho_2}_{\chi_2}$. We write $\llangle C \rrangle^\rho_1 \triangleright s \llangle C \rrangle^\rho_2$ if $\llangle C \rrangle^\rho_1 \rightarrow^\rho \llangle C' \rrangle^\rho_2$, for some $C', \rho', \chi'$, and similarly for $\llangle C \rrangle^\rho_0 \rightarrow^\rho \llangle C \rrangle^\rho_0$.

The sequences of observable actions, called interaction paths, that a configuration $C$ with interface $[\rho, \chi]$ may perform are precisely $s \in L^*$ such that $\llangle C \rrangle^\rho_i \rightarrow^s \llangle C \rrangle^\rho_j$. The following lemma, which is true in $\text{A}_\pi$, relates a computation involving two composed configurations to the interaction paths that each exhibits during the computation.

**Lemma 30 (zip-unzip)** Let $C_1 : [\rho_1, \chi_1], C_2 : [\rho_2, \chi_2]$, and $\rho_1 \cap \rho_2 = \emptyset$. Then $C_1 \triangleright C_2 \triangleleft C$ if and only if for some $s$, $\llangle C_1 \rrangle^\rho_1 \rightarrow^s \llangle C'_1 \rrangle^\rho_1 \rightarrow^s \llangle C'_2 \rrangle^\rho_2 \rightarrow^\tau \llangle C \rrangle^\rho_2$ and $C \equiv (\nu \tilde{z})(C'_1 | C'_2)$, where $\{ \tilde{z} \} = \text{bn}(s)$. \[\Box\]

We only sketch the proof of Lemma 30 as its complete version is very tedious. Before the proof, a few definitions and lemmas are in order. For a term $C$, a subterm of $C$ is said to be at the top level in $C$ if it does not occur under an input prefix or inside a conditional construct. If $\rho; f \vdash C$ we define $\text{rcp}(C) = \rho$ and $\text{ext}(C) = \text{fn}(C) - \rho$. The derivation tree for a reduction step $C \rightarrow C'$ has exactly one leaf $C_0 \rightarrow C'_0$ that is an instance of $\text{RECV}, \text{IF}$, or $\text{ELSE}$. We call $C_0 \rightarrow C'_0$ the core of the derivation. We can show that $C_0$ appears (with bound names possibly renamed) at the top level in $C$, and similarly $C'_0$ (with possible renaming) appears at the top level in $C'$. A context is a configuration with a hole. Structural congruence is extended to contexts the obvious way.

**Lemma 31 (core)** If $C_0 \rightarrow C'_0$ is the core of reductions $C \rightarrow C_1$ and $C \rightarrow C_2$, then $C_1 \equiv C_2$.

**Proof:** By induction on the length of the derivation tree of $C \rightarrow C_1$, we can show that $C \equiv C[C_0]$ and $C_1 \equiv C[C'_0']$. Similarly there is a $C'$ such that $C \equiv C'[C_0]$ and $C_2 \equiv C'[C'_0']$. From the fact that $C[C_0] \equiv C \equiv C'[C_0]$, it follows that $C$ and $C'$ are congruent. Therefore, $C[C'_0] \equiv C'[C'_0']$. \[\Box\]
Lemma 32 (one-step-unzip) If $\text{rcp}(C_1) \cap \text{rcp}(C_2) = \emptyset$ and $C_1 | C_2 \rightarrow C'$ then one of the following holds:

1. $C' \equiv C'_1 | C'_2$, where $C_1 \rightarrow C'_1$ and $C_2 = C'_2$.
2. $C' \equiv C'_1 | C'_2$, where $C_2 \rightarrow C'_2$ and $C_1 = C'_1$.
3. $C' \equiv C'_1 | C'_2$, where $C_1 \equiv C'_1 | \overline{\gamma} y$, $C_2 | \overline{\gamma} y \rightarrow C'_2$, and $x \in \text{rcp}(C_2)$.
4. $C' \equiv C'_1 | C'_2$, where $C_2 \equiv C'_2 | \overline{\gamma} y$, $C_1 | \overline{\gamma} y \rightarrow C'_1$, and $x \in \text{rcp}(C_1)$.
5. For any $y \notin fn(C_1) \cup fn(C_2)$, $C' = (\nu y)(C'_1 | C'_2)$, where $C_1 \equiv (\nu y)(C'_1 | \overline{\gamma} y)$, $C_2 | \overline{\gamma} y \rightarrow C'_2$, and $x \in \text{rcp}(C_2)$.
6. For any $y \notin fn(C_1) \cup fn(C_2)$, $C' = (\nu y)(C'_1 | C'_2)$, where $C_2 \equiv (\nu y)(C'_2 | \overline{\gamma} y)$, $C_1 | \overline{\gamma} y \rightarrow C'_1$, and $x \in \text{rcp}(C_1)$.

**Proof:** Let $C_0 \rightarrow C'_0$ be the core of $C_1 | C_2 \rightarrow C'$. There are three cases depending on whether the core is an instance of $\text{IF}$, $\text{ELSE}$ or $\text{RECV}$. Suppose the core is an instance of $\text{IF}$ or $\text{ELSE}$. Then it follows that $C_0$ (with its bound names possibly alpha renamed) occurs at the top level in either $C_1$ or $C_2$. If it occurs in $C_1$, case 1 of the lemma statement applies, else case 2 applies. If the core is an instance $\text{RECV}$, we have $C_0 \equiv C'_{01} | C'_{02}$, where $C'_{01} = \overline{\gamma} y$ and $C'_{02} = x(z).C'_0$. There are four cases depending on where $C'_{01}$ and $C'_{02}$ appear (with their bound names possibly alpha-renamed) in $C_1 | C_2$:

1. **$(C'_{01}$ and $C'_{02}$ are subterms of $C_1$):** Since $C'_{01}$ and $C'_{02}$ are at the top level in $C_1$, it follows $C_1 \equiv (\nu \overline{\gamma})(C'|C'_{01} | C'_{02})$ for some $\overline{\gamma}$ and $C'$. Using the rules of Table 5.3, we can derive $C_1 | C_2 \rightarrow (\nu \overline{\gamma})(C'|C'_{0}) | C_2$ with $C_0 \rightarrow C'_0$ as its core. Then, by Lemma 31 we have $C' \equiv C'_1 | C'_2$, where $C'_{1} = (\nu \overline{\gamma})(C'|C'_{0})$ and $C'_{2} = C_2$. Thus, statement 1 of the lemma applies.

2. **$(C'_{01}$ and $C'_{02}$ are subterms of $C_2$):** Similar to case 1.

3. **$(C'_{01}$ is a subterm of $C_1$ and $C'_{02}$ a subterm of $C_2$):** Since $C'_{01}$ ($C'_{02}$) is at the top level in $C_1$ ($C_2$). Depending on whether $y$ is restricted in $C_1$, we have two subcases:
• \( C_1 \equiv (\nu \tilde{u})(C'_1|\overline{x}y) \) and \( C_2 \equiv (\nu \tilde{v})(x(z).C'_{02}|C''_2) \) such that \( x, y \notin \{\tilde{u}, \tilde{v}\} \). Therefore, \( x \in \text{rcp}(C_2) \). We have \( C_1|C_2 \equiv (\nu \tilde{u})(C'_1|\overline{x}y)(\nu \tilde{v})(x(z).C'_{02}|C''_2) \). Thus, we can derive \( C_1|C_2 \longrightarrow (\nu \tilde{u})(C'_1|\overline{x}y)(\nu \tilde{v})(C''_0|C''_2) \) with \( C_0 \longrightarrow C'_0 \) as its core. Then by Lemma 31, \( C' \equiv C'_1|C'_2 \) where \( C'_1 \equiv (\nu \tilde{u})C'_1 \) and \( C'_2 \equiv (\nu \tilde{v})(C''_0|C''_2) \). Hence, statement 3 of the lemma applies.

• For any \( y \notin \text{fn}(C_1) \cup \text{fn}(C_2) \), we can write

\[
C_1 \equiv (\nu \tilde{u}, y)(C'_1|\overline{x}y) \quad \text{and} \quad C_2 \equiv (\nu \tilde{v})(x(z).C'_{02}|C''_2)
\]

such that \( x, y \notin \{\tilde{u}, \tilde{v}\} \). Therefore, \( x \in \text{rcp}(C_2) \). We have

\[
C_1|C_2 \equiv (\nu y)((\nu \tilde{u})(C'_1|\overline{x}y)(\nu \tilde{v})(x(z).C'_{02}|C''_2)).
\]

Thus, we can derive \( C_1|C_2 \longrightarrow (\nu y)((\nu \tilde{u})(C'_1|\overline{x}y)(\nu \tilde{v})(C''_0|C''_2)) \) with \( C_0 \longrightarrow C'_0 \) as its core. Then by Lemma 31, \( C' \equiv (\nu y)(C'_1|C'_2) \) where \( C'_1 \equiv (\nu \tilde{u})C'_1 \) and \( C'_2 \equiv (\nu \tilde{v})(C''_0|C''_2) \). Hence, statement 5 of the lemma applies.

4. \( \text{(C}_{01} \text{ is a subterm of } C_2 \text{ and } C_{02} \text{ a subterm of } C_1) \): Similar to case 3. \( \square \)

**Proof of Lemma 30 (zip-unzip):**

**(if : zip)** The proof is by induction on length of \( s \).

For the base case, we have \( s = \epsilon \), and

\[
\langle C_1 \rangle^\epsilon_{\chi_1} \rightarrow \langle C'_1 \rangle^\epsilon_{\chi_1}, \quad \langle C_2 \rangle^\epsilon_{\chi_2} \rightarrow \langle C'_2 \rangle^\epsilon_{\chi_2}
\]

By Lemma 29, we have \( C_1 \longrightarrow C'_1 \) and \( C_2 \longrightarrow C'_2 \). Then by repeated application of \( \text{PAR} \) we have

\( C_1|C_2 \longrightarrow C'_1|C_2 \longrightarrow C'_1|C'_2 \). The result follows from the fact that \( \text{bn}(\epsilon) = \emptyset \).

For the induction step, there are two cases out of which we only consider the case \( s = s'.(\tilde{y})xy \).

From \( \text{IN}, \text{OUT}, \) and \( \text{LEQV} \) we have

\[
\begin{align*}
\langle C_1 \rangle_{\chi_1}^{\rho_1} & \rightarrow_{s'} \langle C_3 \rangle_{\chi_1}^{\rho_1} \langle \overline{y} \rangle_{\chi_1}^{\rho_1} \langle C_3 \rangle_{\chi_1}^{\rho_1} \langle \overline{x} \rangle_{\chi_1}^{\rho_1} \rightarrow_{\rho_1} \langle C'_1 \rangle_{\chi_1}^{\rho_1} \langle \overline{x} \rangle_{\chi_1}^{\rho_1} \left( \tilde{y} \bigcap (\rho_1 \cup \chi_1) = \emptyset, x \in \rho_1 \right) \\
\langle C_2 \rangle_{\chi_2}^{\rho_2} & \rightarrow_{\overline{y}} \langle (\nu \tilde{y})(C_4 \overline{y}) \rangle_{\chi_2}^{\rho_2} \langle C_4 \rangle_{\chi_2}^{\rho_2} \rightarrow_{\rho_2} \langle C'_2 \rangle_{\chi_2}^{\rho_2} \left( \tilde{y} \bigcap (\rho_2 \cup \chi_2) = \emptyset, x \in \chi_2 \right)
\end{align*}
\]
From Lemma 29, $C_3 \models y \Rightarrow C'_1$ and $(\nu y)(C_4 \models y) \Rightarrow C'_2$. Let $\tilde{z} = bn(s')$. By induction hypothesis $C_1|C_2 \Rightarrow C'$ where $C'' \equiv (\nu \tilde{z})(C_3|\nu y)(C_4|\nu y)$. By Theorem 12, we have $C_3 \models [\rho'_1, \chi'_1]$. It follows $fn(C_3) \subset \rho'_1 \cup \chi'_1$. Now, since $\hat{y} \cap (\rho'_1 \cup \chi'_1) = \emptyset$, it follows that $\hat{y} \cap fn(C_3) = \emptyset$. Using this, and by repeated application of PAR and RES we have $C'' \equiv (\nu \tilde{z}, \hat{y})(C_3|\nu y|C_4) \Rightarrow (\nu \tilde{z}, \hat{y})(C'_1|C'_2)$. The lemma follows from the observation that $\{\tilde{z}, \hat{y}\} = bn(s)$.

(only if : unzip) Proof by induction on the length of the computation path $C_1|C_2 \Rightarrow C$. For the base case where the length is zero, the result holds with $s = \epsilon$. For the induction step, let the length be $n > 0$. By induction hypothesis, we can unzip the first $n - 1$ steps of $C_1|C_2 \Rightarrow C'' \rightarrow C$ as

\[
\langle\langle C_1 \rangle\rangle_{\chi_1}^{\rho_1} \xrightarrow{s'} \langle\langle C_1 \rangle\rangle_{\chi_1}^{\rho'_1} \text{ and } C'' \equiv (\nu \tilde{z})(C'_1|C'_2), \{\tilde{z}\} = bn(s').
\]

It is easy to show $\rho'_1 \cap \rho'_2 = \emptyset$. We are done if we show $(\nu \tilde{z})(C'_1|C'_2) \rightarrow C$ can be unzipped. From Lemma 27 we have $C'_1|C'_2 \rightarrow C'$ where $C \equiv (\nu \tilde{z})(C')$. Now, there are four cases according to Lemma 32 out of which we only consider cases 1 and 5 (the others are similar).

1. (case 1): We have $C' \equiv C'_1|C'_2$, $C_1 \rightarrow C''$, $C'' = C'_2$. From $C_1 \rightarrow C''$ we construct

\[
\langle\langle C_1 \rangle\rangle_{\chi} \xrightarrow{s'} \langle\langle C'_1 \rangle\rangle_{\chi_1}^{\rho'_1} \text{ and } C'' \equiv (\nu \tilde{z})(C'_1|C'_2), \{\tilde{z}\} = bn(s').
\]

Then we have

\[
\langle\langle C_1 \rangle\rangle_{\chi_1}^{\rho'_1} \approx_{\chi_1} \langle\langle C_1'' \rangle\rangle_{\chi_1}^{\rho'_1} \cup \{y\} \text{ and } \langle\langle C_2 \rangle\rangle_{\chi_2}^{\rho'_2} \approx_{\chi_2} \langle\langle C_2'' \rangle\rangle_{\chi_2}^{\rho'_2} \cup \{y\}.
\]

Then $C \equiv (\nu \tilde{z})C' \equiv (\nu \tilde{z}, y)(C'_1|C'_2)$. The result follows by setting $s = s'.y$ and noting that $bn(s) = \{\tilde{z}, y\}$.

$\square$
5.8 Alternate Characterization of May Testing

We present an alternate characterization of may testing in $A\pi$ that is based on interaction paths. We follow the same approach used for asynchronous $\pi$-calculus by Boreale [5]. However, differences between the two calculi, such as in name matching capabilities and notions of observability, lead to changes in the characterization and require different proofs to establish it. To demonstrate these differences, we follow Boreale’s proof layout and highlight the differences as they arise.

Definition 23 and Lemma 33 demonstrate the relation between interaction paths exhibited by a configuration and the evolution of its interface.

**Definition 23** We define the following functions on interaction paths

$$rcp([\rho, \chi], \epsilon) = \rho$$

$$rcp([\rho, \chi], s.(\hat{y})\overline{xy}) = \hat{y} \cup rcp([\rho, \chi], s)$$

$$rcp([\rho, \chi], s.(\hat{y})xy) = rcp([\rho, \chi], s)$$

$$ext([\rho, \chi], \epsilon) = \chi$$

$$ext([\rho, \chi], s.(\hat{y})\overline{xy}) = ext([\rho, \chi], s)$$

$$ext([\rho, \chi], s.(\hat{y})xy) = (\{y\} \cup ext([\rho, \chi], s)) - rcp([\rho, \chi], s)$$

**Lemma 33** If $\ll C \rr_\chi^\rho \xrightarrow{s} \ll C' \rr_\chi'^{\rho'}$ then

1. $\rho' = rcp([\rho, \chi], s)$ and $\chi' = ext([\rho, \chi], s)$,

2. $s = s_1.(\hat{y})xy.s_2$ implies $x \in rcp([\rho, \chi], s_1)$, and $\hat{y} \cap (rcp([\rho, \chi], s_1) \cup ext([\rho, \chi], s_1)) = \emptyset$.

3. $s = s_1.(\hat{y})\overline{xy}.s_2$ implies $x \in ext([\rho, \chi], s_1)$, and $y \in rcp([\rho, \chi], s_1) \cup ext([\rho, \chi], s_1)$ if and only if $\hat{y} = \emptyset$.

4. $rcp([\rho, \chi], s) \cup ext([\rho, \chi], s) = n(s) \cup \rho \cup \chi$, and

5. $s = s_1.\alpha.s_2$ implies $bn(\alpha) \cap (n(s_1) \cup \rho \cup \chi) = \emptyset$.

**Proof:**

1. The proof is by induction on the length of $s$. For the base case $s = \epsilon$, by $TAU$ rule we have $\rho' = \rho, \chi' = \chi$, and the result follows. For the induction step we consider only the case $s = s'.(\hat{y})xy$, and leave the case $s = s'.(\hat{y})\overline{xy}$ to the reader. We have

$$\ll C \rr_\chi^\rho \xrightarrow{s'} \ll C_1 \rr_{\chi_1}^{\rho_1} (\hat{y}) \overline{xy} \ll C_2 \rr_{(\chi_1 \cup \{y\})-\rho_1}^{\rho_1} \xrightarrow{\cdot} \ll C' \rr_{(\chi_1 \cup \{y\})-\rho_1}^{\rho_1}$$

66
and \( \rho' = \rho_1, \chi' = (\chi_1 \cup \{y\}) - \rho_1 \). From induction hypothesis, \( \rho_1 = rcp([\rho, \chi], s') \) and \( \chi_1 = ext([\rho, \chi], s') \). Then \( \rho' = \rho_1 = rcp([\rho, \chi], s') = rcp([\rho, \chi], s'.(\bar{y})xy) \), and \( \chi' = (\chi_1 \cup \{y\}) - \rho_1 = (\{y\} \cup ext([\rho, \chi], s')) - rcp([\rho, \chi], s') = ext([\rho, \chi], s'.(\bar{y})xy) \).

2. From part 1 of this lemma and IN rule.

3. We have \( s = s'.(\bar{y})xy \) and 
   \[
   \langle C \rangle^p_\chi \xrightarrow{s_1} \langle (\nu \bar{y})(C_1[\bar{y}]y) \rangle^{\rho_1}_{\chi_1} (\bar{y}xy)
   \]
   By part 1 of this lemma, \( \rho_1 = rcp([\rho, \chi], s') \) and \( \chi_1 = ext([\rho, \chi], s') \). By OUT rule, \( x \in \chi_1 = ext([\rho, \chi], s') \). Let \( C'_1 = (\nu \bar{y})(C_1[\bar{y}]y) \). By Theorem 12, \( \rho'_1: f'_1 \vdash C'_1 \), for some \( [\rho'_1, fn(C'_1) - \rho'_1] \leq [\rho_1, \chi_1] \). If \( \bar{y} = 0 \) then either \( y \in \rho'_1 \) or \( y \in fn(C'_1) - \rho'_1 \). In either case, from \( [\rho'_1, fn(C'_1) - \rho'_1] \leq [\rho_1, \chi_1] \) it follows that \( y \in \rho_1 \cup \chi_1 = rcp([\rho, \chi], s') \cup ext([\rho, \chi], s') \). Conversely, if \( y \in rcp([\rho, \chi], s') \cup ext([\rho, \chi], s') \) then by the side condition of OUT rule, we have \( \bar{y} = 0 \).

4. The proof is by induction on length of \( s \). The base case is obvious. For the induction step there are two cases out of which we only consider \( s = s'.(\bar{y})xy \). By Definition 23, we have 
   \[
   rcp([\rho, \chi], s) \cup ext([\rho, \chi], s) = rcp([\rho, \chi], s') \cup ((\{y\} \cup ext([\rho, \chi], s')) - rcp([\rho, \chi], s')) = \{y\} \cup rcp([\rho, \chi], s') \cup ext([\rho, \chi], s') = \{x, y\} \cup rcp([\rho, \chi], s') \cup ext([\rho, \chi], s').
   \]
   The last equality follows from part 2 of this lemma. From induction hypothesis we have 
   \[
   rcp([\rho, \chi], s') \cup ext([\rho, \chi], s') = n(s') \cup \chi \cup \rho.
   \]
   Using this, we get 
   \[
   rcp([\rho, \chi], s) \cup ext([\rho, \chi], s) = \{x, y\} \cup n(s') \cup \chi \cup \rho.
   \]
   Now, the result follows from the observation that \( \{x, y\} \cup n(s') = n(s) \).

5. We have 
   \[
   \langle C \rangle^p_\chi \xrightarrow{s_1} \langle C_1 \rangle^p_{\chi_1} \xrightarrow{\alpha} \langle C_2 \rangle^p_{\chi_2} \xrightarrow{s_2}
   \]
   By IN and OUT rules, we have \( bn(\alpha) \cap (\rho_1 \cup \chi_1) = \emptyset \). By parts 1 and 4 of this lemma \( \rho_1 \cup \chi_1 = rcp([\rho, \chi], s_1) \cup ext([\rho, \chi], s_1) = n(s_1) \cup \rho \cup \chi \), and the result follows.

For \( \rho \cap \chi = \emptyset \), we define \( L^*[\rho, \chi] \) as the set of all \( s \in L^* \) that satisfy conditions 2 and 3 of Lemma 33. We define alpha equivalence on paths the obvious way, and work modulo alpha equivalence. Note that \( L^*[\rho, \chi] \) is not closed under alpha renaming, that is for \( s \in L^*[\rho, \chi] \) there may be \( r \equiv_\alpha s \) but \( r \notin L^*[\rho, \chi] \). Therefore, we will only consider alpha renaming that does not result in such ill-formed paths.
Table 5.4: A relation on interaction paths.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(L1)</td>
<td>( s.(\hat{y}) \bar{y}x \prec s )</td>
</tr>
<tr>
<td>(L2)</td>
<td>( s \prec s.(\hat{y})xy )</td>
</tr>
<tr>
<td>(L3)</td>
<td>( s_1.(M)uv.(N)xy.s_2 \prec s_1.(\hat{y})xy.(\hat{v})uv.s_2 )</td>
</tr>
<tr>
<td>(L4)</td>
<td>( s_1.(M)\bar{y}v.(N)xy.s_2 \prec s_1.(\hat{y})x\bar{y}.(\hat{v})uv.s_2 )</td>
</tr>
<tr>
<td>(L5)</td>
<td>( s_1.(\hat{v})\bar{v}v.(\hat{y})xy.s_2 \prec s_1.(\hat{y})xy.(\hat{v})uv.s_2 ) if ( u, v \notin \hat{y} )</td>
</tr>
</tbody>
</table>

In L3 and L4, \( M = \hat{v}, N = \hat{y} \) if \( v \notin \hat{y} \), and \( M = \hat{y}, N = \emptyset \) otherwise.

**Definition 24 (path transformation)** We define a relation \( \preceq \) on \( L^* \) as the reflexive transitive closure of the relation defined in Table 5.4. □

The intuition behind laws in Table 5.4, which is stated formally in Lemma 35, is that, for \( r, s \in L^*[\rho, \chi] \) and \( r \prec s \), if a configuration exhibits \( \overline{r} \) then it can also exhibit \( \overline{s} \). L3 states that two consecutive outputs can be commuted, while L4 states that two consecutive inputs can be commuted. L5 states that an output can be postponed to after an input, provided the input doesn’t use the bound name exported by the output. L3 and L5 can be used to postpone outputs, and L4 and L5 to prepone inputs. These two rules capture the essence of asynchrony. L1 states that additional inputs may be appended, and L2 states that a tailing output can be removed as there is no interaction after it that depends on it.

**Lemma 34** Let \( r, s \in L^*[\rho, \chi] \), and \( r = r_0 \prec r_1 \prec \ldots \prec r_n \prec s \). Then \( r_i \in L^*[\rho, \chi] \) for \( 1 \leq i \leq n \).

**Proof:** Suppose for some \( 1 \leq i \leq n, r_i \notin L^*[\rho, \chi] \). Then \( r_i \) violates either property 2 or 3 of Lemma 33. We consider only the later, the former is similar. We have \( r_i = t_1.(\hat{y})\overline{y}.t_2 \) and at least one of the following conditions holds.

1. \( x \notin ext([\rho, \chi], t_1) \): There are four subcases depending on which law \( r_{i-1} \prec r_i \) is an instance of.
   - L1 or L2: Then \( r_{i-1} = t_1.(\hat{y})\overline{y}.t_3 \) for some \( t_3 \), and hence \( r_{i-1} \notin L^*[\rho, \chi] \).
   - L3: Then \( r_{i-1} = t'_1.(\hat{y})\overline{y}.t'_2 \) such that \( ext([\rho, \chi], t'_1) = ext([\rho, \chi], t_1) \). Then \( r_{i-1} \notin L^*[\rho, \chi] \).
   - L4: Then from the observation that output actions do not change the set of external names it follows that \( r_{i-1} \notin L^*[\rho, \chi] \).
• L5: Then we have \( t_1 = t'_1.(\hat{v})uv \), and \( r_{i-1} = t'_1.(\hat{y})\varphi y.(\hat{v})uv.t_2 \). But then again \( r_{i-1} \notin \mathcal{L}^*[\rho, \chi] \) because \( \text{ext}([\rho, \chi], t'_1) \subset \text{ext}([\rho, \chi], t_1) \).

Thus, in all cases \( r_{i-1} \notin \mathcal{L}^*[\rho, \chi] \). When this argument is applied repeatedly we get the contradiction that \( r \notin \mathcal{L}^*[\rho, \chi] \).

2. \( y \notin \text{rcp}([\rho, \chi], t_1) \cup \text{ext}([\rho, \chi], t_1) \) and \( \hat{y} = \emptyset \): A similar case analysis as in case 1 shows that \( r_{i-1} \notin \mathcal{L}^*[\rho, \chi] \), which again when repeatedly applied leads to the contradiction that \( r \notin \mathcal{L}^*[\rho, \chi] \).

3. \( y \in \text{rcp}([\rho, \chi], t_1) \cup \text{ext}([\rho, \chi], t_1) \) and \( \hat{y} = \{ y \} \): By property 1 of Lemma 37, \( y \in n(t_1) \cup \rho \cup \chi \).

Then, since \( \hat{y} = \{ y \} \) property 2 of Lemma 37 is violated. Contradiction.

In all cases we have arrived at a contradiction. Thus \( r_{i-1} \in \mathcal{L}^*[\rho, \chi] \) for all \( 1 \leq i \leq n \). (For the case where we start with the initial assumption that \( r_1 \) violates property 2 of Lemma 33, we use a similar argument as above, but move up the path to arrive at the contradiction that \( s \notin \mathcal{L}^*[\rho, \chi] \).)

\[ \square \]

**Lemma 35** If \( \langle C \rangle^\rho_{\chi} \xrightarrow{\varphi} r \preceq s \) and \( \varphi \in \mathcal{L}^*[\rho, \chi] \), then \( \langle C \rangle^\rho_{\chi} \xrightarrow{\varphi} \).

**Proof:** We have \( \varphi, \varphi \in \mathcal{L}^*[\rho, \chi], r \preceq s \). Let \( r \prec r_1 \prec \ldots \prec r_n \prec s \). By Lemma 38, \( \varphi \prec \varphi_1 \prec \ldots \prec \varphi_n \prec \varphi \). Then by Lemma 34, we have \( \varphi_i \in \mathcal{L}^*[\rho, \chi] \) for \( 1 \leq i \leq n \). So the lemma follows by a simple induction on \( n \) if we prove it just for the case \( r \prec s \).

Let \( r \prec s \). There are five cases one for each law in Table 5.4. We consider only L1, L3 and L5.

1. L1: Let \( r = s.(\hat{y})\varphi y \). We know that \( \langle C \rangle^\rho_{\chi} \xrightarrow{\varphi} \langle C_1 \rangle^\rho_{\chi_1} \). By Lemma 33, we have \( \rho_1 = \text{rcp}([\rho, \chi], \varphi), \chi_1 = \text{ext}([\rho, \chi], \varphi) \). Then from \( \varphi \in \mathcal{L}^*[\rho, \chi] \) it follows \( x \in \rho_1, \hat{y} \cap (\rho_1 \cup \chi_1) = \emptyset \).

Now, by IN rule we have \( \langle C \rangle^\rho_{\chi} \xrightarrow{\varphi} \langle C_1 \rangle^\rho_{\chi_1} \xrightarrow{(\hat{y})\varphi y} \).

2. L3: Let \( s = s_1.\hat{y}xy.(\hat{v})uv.s_2 \) and \( r = s_1.(M)uv.(N)xy.s_2 \) where \( M \) and \( N \) are as defined in the side condition of L3. We know that

\[
\langle C \rangle^\rho_{\chi} \xrightarrow{\varphi} \langle C_1 \rangle^\rho_{\chi_1} \xrightarrow{(\hat{y})\varphi y} \langle C_2 \rangle^\rho_{\chi_1} \xrightarrow{(\hat{v})\varphi y} \langle C_3 \rangle^\rho_{\chi_1} \xrightarrow{(\hat{v})\varphi y} \langle C_4 \rangle^\rho_{\chi_1} \xrightarrow{\varphi} .
\]

69
From \( \text{OUT} \) we deduce \( C_1 \equiv (\nu^y)(C_2|\overline{x} y) \) and \( C_3 \equiv (\nu^\emptyset)(C_4|\overline{\pi}v) \). Then we have

\[
\langle C \rangle^\rho_x \xrightarrow{\overline{x} y} \langle (\nu^y)(C_2|\overline{x} y) \rangle^\rho_1 \xrightarrow{(\nu^\emptyset, \hat{\nu})} \langle (\nu^y, \hat{\nu})(C_4|\overline{\pi}v) \rangle^\rho_1 \xrightarrow{(M|\overline{\pi}v)} \langle (\nu N)(C_4|\overline{\pi}v) \rangle^\rho_1 \xrightarrow{(N|\overline{\pi}v)} \langle C_4 \rangle^\rho_1 \xrightarrow{(\nu^y|\nu\hat{\nu}^y)} \frac{s_2}{s_2}.
\]

because \( M \cup N = \hat{y} \cup \hat{\nu} \).

3. \( L5 \): Let \( s = s_1.\langle \hat{y} \rangle xy.(\hat{v})\overline{\pi}v.s_2 \) and \( r = s_1.\langle \pi \rangle xy.(\hat{y})xy.s_2 \), where \( u, v \not\in \hat{y} \). We know that

\[
\langle C \rangle^\rho_x \xrightarrow{\overline{y} y} \langle C_1 \rangle^\rho_1 \xrightarrow{(\nu^y|\nu\hat{y}^y)} \langle C_2 \rangle^\rho_1 \xrightarrow{(\nu^\emptyset, \hat{\nu})} \langle C_3 \rangle^\rho_1 \xrightarrow{(\nu^y|\nu\hat{y}^y)} \langle C_4 \rangle^\rho_1 \xrightarrow{(\hat{y}|\nu)} \frac{s_2}{s_2}.
\]

From \( \text{IN} \) we deduce \( C_4 \equiv (C_3|\overline{\pi}v) \), and from \( \text{OUT} \) we deduce \( C_1 \equiv (\nu^y)(C_2|\overline{x} y) \). \( \hat{y} \cap (\rho_1 \cup \chi_1) = \emptyset \). Then we have

\[
\langle C \rangle^\rho_x \xrightarrow{\overline{s}_1} \langle (\nu^y)(C_2|\overline{x} y) \rangle^\rho_1 \xrightarrow{(\nu^\emptyset, \hat{\nu})} \langle (\nu^y)(C_2|\overline{x} y|\overline{\pi}v) \rangle^\rho_1 \xrightarrow{(\hat{y}|\nu)} \langle C_4 \rangle^\rho_1 \xrightarrow{(\hat{y}|\nu)} \frac{s_2}{s_2}.
\]

because \( (\chi_1 \cup \hat{\nu}) \) - \( \rho_1 = (\chi_1 \cup \hat{\nu}) \) - \( \rho_1 \cup \hat{y} \).

We now compare our laws with those of Boreale. The mismatch capability in \( \text{A}\pi \) enables distinguishing bound names from free names. Thus, Boreale’s law that allows replacing bound names in an input action with free names is not applicable in \( \text{A}\pi \). Furthermore, Boreale’s annihilation law, which states that a configuration can consume a pair of complementary interactions, is not needed in \( \text{A}\pi \) for two reasons. First, due to the encapsulation property a configuration can never exhibit complementary actions. Second, because we do not have a law that substitutes bound names with fresh names, no path with complementary actions is related to a path in \( L^*[\rho, \chi] \). Finally, as opposed to asynchronous inputs allowed by the \( \text{IN} \) rule, Boreale’s LTS uses synchronous inputs. As a consequence, \( L4 \) is not applicable there. These differences lead to a stronger characterization of may preorder for \( \text{A}\pi \) (see below).

Using \( \preceq \) we define the following preorder on configurations, which we will prove to be an alternate characterization of may preorder.
Definition 25 Let \([\rho_1, \chi_1]\) be the minimal interface of \(C_1\) and \([\rho_2, \chi_2]\) that of \(C_2\). For \(\rho\) such that \(\rho_1, \rho_2 \subseteq \rho\), we say \(C_1 \preceq \rho \ C_2\) if for \(\chi = (\chi_1 \cup \chi_2) - \rho\), \(\langle C_1 \rangle^\rho \overset{s}{\Rightarrow} \langle C_2 \rangle^\rho \overset{r}{\Rightarrow} \) for some \(r \triangleq s\).

Although it is easy to see that \(C_1 \preceq \rho \ C_2\) implies \(C_1 \preceq \rho \ C_2\), the reverse direction is more involved. To prove the reverse direction, we construct for a given \(s \in \mathcal{L}^*[\rho, \chi]\), an observer \(O\) such that for \(C : [\rho, \chi]\), if \(C\) may \(O\) then \(\langle C \rangle^\rho \overset{r}{\Rightarrow} \) for some \(r \triangleq s\).

Definition 26 (canonical observer) For \(s \in \mathcal{L}^*[\rho, \chi]\), we define an observer

\[O([\rho, \chi], s) = (\nu \xi, z)(\{x_i \in \text{ext}([\rho, \chi], s)\} \text{Proxy}(s, x_i, z) \mid O'([\rho, \chi], s, z))\]

where for \(u, v\) fresh \(\{\xi\} = \text{bn}(s) - \text{rcp}([\rho, \chi], s)\)

\[O'([\rho, \chi], \xi, z) \triangleq \overline{\rho}\]

\[O'([\rho, \chi], (\hat{y})xy.s, z) \triangleq \overline{\rho}y|O'([\rho, \chi \cup \{y\} - \rho], s, z)\]

\[O'([\rho, \chi], \overline{x}y.s, z) \triangleq z(u, v)[u = x \land v = y](O'([\rho, \chi], s, z), 0)\]

\[O'([\rho, \chi], \overline{x}(y).s, z) \triangleq z(u, y)[u = x \land y \notin (\rho \cup \chi)](O'([\rho \cup \{y\}, \chi], s, z), 0)\]

\[\text{Proxy}(\epsilon, x, z) = 0\]

\[\text{Proxy}((\hat{y})xy.s, x, z) \triangleq \text{Proxy}(s, x, z)\]

\[\text{Proxy}((\hat{y})\overline{x}y.s, x, z) \triangleq x(v).(\overline{x}(x, v) \mid \text{Proxy}(s, x, z))\]

In the above, \(\triangleq\) is used for macro definitions.

The observer \(O([\rho, \chi], s)\) consists of a collection of proxies and a central matcher. There is one forwarding proxy for each external name a configuration \(C\) knows while doing \(s\). This forwarding mechanism, which is absent in Boreale’s construction, is essential in our case because of uniqueness of actor names. The matcher which analyzes the forwarded messages, keeps track of names in the “current” interface of \(C\) and uses them to distinguish bound names from free names in outputs. This technique works because if \((\hat{y})\overline{x}y.s \in \mathcal{L}^*[\rho, \chi]\) and \(y \notin \rho \cup \chi\) then \(\hat{y} = \{y\}\). The abbreviations \(\notin\) and \(\land\) used in the definition can be encoded using the conditional construct. The encoding of \(\notin\) requires the ability to mismatch names. Note that the definition also uses polyadic communication between proxies and matcher, whose encoding was shown in Section 5.5.

We not only require a different construction for the canonical observer than Boreale’s, but also an essentially different argument for establishing Lemma 36 (see Appendix for proof). For
s ∈ \mathcal{L}^*[\rho, \chi]$, let $\mathcal{ffl}(\rho, \chi, s)$ be the set of all names $y$ such that $s$ can be written as $s_1.xy.s_2$ and $y \notin \text{recp}(\rho, \chi, s_1) \cup \text{ext}(\rho, \chi, s_1)$. It is easy to show that if $s \in \mathcal{L}^*[\rho, \chi]$ and $\chi' = \chi \cup \mathcal{ffl}(\rho, \chi, s)$, then $O([\rho, \chi], s) : [\chi', \rho]$.

**Lemma 36** Let $r, s \in \mathcal{L}^*[\rho, \chi]$ and $\chi' = \chi \cup \mathcal{ffl}(\rho, \chi, s)$. Then

$\langle \langle O([\rho, \chi], s) \rangle \rangle^*_\rho \xRightarrow{r, \mu} \text{ implies } r \leq s$.

**Proof:** The proof is by induction on length of $s$. For the base case, we have $s = \epsilon$. It follows from the definition of $O([\rho, \chi], \epsilon)$ that $\tau$ contains only inputs, and $r$ only outputs. Lemma follows from repeated application of (L1). For the induction step, there are three cases:

1. $s = \mathcal{R}(y).s'$: Then $x \in \chi$, and $y \notin \rho \cup \chi$. Since $s \in \mathcal{L}^*[\rho, \chi]$, we have $s' \in \mathcal{L}^*[\rho \cup \{y\}, \chi]$. Note that $O([\rho, \chi], \mathcal{R}(y).s')$ first waits for a message $\tau w$ for some $w \notin \rho \cup \chi$ before generating an event or sending any messages. From this observation, it follows that $r$ is of the form $(v_1)\mathcal{R}v_1 \ldots (v_n)\mathcal{R}v_n.(\hat{w})\tau w.r_0$. Since $r \in \mathcal{L}^*[\rho, \chi]$ and $w \notin \rho \cup \chi$, it follows that $\hat{w} = \{w\} - \cup_1 \leq i \hat{v}_i$, i.e. $r$ has a bound output with argument $w$. Then $\tau$ has a bound input with argument $w$. Then, since $\tau \in \mathcal{L}^*[\chi', \rho]$, we have $w \notin \chi' \cup \rho$. Then we have

$\langle \langle O([\rho, \chi], s) \rangle \rangle^*_\rho \xRightarrow{r(w)} \langle \langle O([\rho \cup \{w\}, \chi], s'\{w/y\}) \rangle \rangle^*_\rho \xRightarrow{r'} \tau$

where $\tau' = (v'_1)u_1v_1 \ldots (v'_n)u_nv_n.\tau_0$, $v'_i = \hat{v}_i - \{w\}$. Clearly, $r' \in \mathcal{L}^*[\rho \cup \{w\}, \chi]$. It is easy to show since $w \notin \rho \cup \chi'$, $s'\{w/y\} \in \mathcal{L}^*[\rho \cup \{w\}, \chi]$. Further, $\chi' = \chi \cup \mathcal{ffl}(\rho \cup \{w\}, \chi, s'\{w/y\})$. By induction hypothesis, $r' \leq s'\{w/y\}$. Then $\tau(w).r' \leq \tau(w).\langle \langle O([\rho, \chi], s'\{w/y\}) \rangle \rangle^*_\rho \xRightarrow{r'} \tau$. The result follows from transitivity of $\leq$ and that $s$ is alpha equivalent to $\tau(w).\langle \langle O([\rho, \chi], s'\{w/y\}) \rangle \rangle^*_\rho \xRightarrow{r'} \tau$.

2. $s = \mathcal{X}y.s'$: Then $x \in \chi$, and $y \notin \rho \cup \chi$. Since $s \in \mathcal{L}^*[\rho, \chi]$, we have $s' \in \mathcal{L}^*[\rho, \chi]$. Note that $O([\rho, \chi], \mathcal{X}y.s')$ first waits for a message $\tau y$ before generating an event or sending any messages. From this observation, it follows that $\tau$ is of form $(v_1)u_1v_1 \ldots (v_n)u_nv_n.\mathcal{X}y.\tau_0$, where $y \notin \cup_i \hat{v}_i$. Then we also have

$\langle \langle O([\rho, \chi], s) \rangle \rangle^*_\rho \xRightarrow{\mathcal{X}y} \langle \langle O([\rho, \chi], s') \rangle \rangle^*_\rho \xRightarrow{r'} \tau$

where $\tau' = (v_1)u_1v_1 \ldots (v_n)u_nv_n.\tau_0$. Further, since $r \in \mathcal{L}^*[\rho, \chi]$ it is clear that $r' \in \mathcal{L}^*[\rho, \chi]$. 72
Moreover, since $\text{ffi}([\rho, \chi], s) = \text{ffi}([\rho, \chi], s')$, we have $\chi' = \chi \cup \text{ffi}([\rho, \chi], s')$. By induction hypothesis, $r' \preceq s'$. Then $\text{xy}.r' \preceq \text{xy}.s'$. By repeated application of $\text{L4}$ we deduce $r \preceq \text{xy}.r'$. The result follows from transitivity of $\preceq$.

3. $s = \text{xy}.s'$: Then $x \in \rho$, and since $s \in L^*[\rho, \chi]$, we have $s' \in L^*[\rho, \chi \cup \{y\} - \rho]$. Now, we can show

$$O([\rho, \chi], s) \equiv \text{xy} \mid O([\rho, \chi \cup \{y\} - \rho], s')$$

There are two possible cases depending on whether $\text{xy}$ fires or not. We consider only the case where it fires, the other is similar. Since $\text{xy}$ fires, it follows that $\tau = \tau_1 \cdot \text{xy} \cdot \tau_2$, where $y \notin bn(r_1)$, because $y \in \chi' \cup \rho$. Then it is the case that

$$\langle \langle O([\rho, \chi \cup \{y\} - \rho], s') \rangle \rangle^{\chi'}_{\rho} \Rightarrow$$

Further, since $r \in L^*[\rho, \chi]$ and $y \notin bn(r_1)$, we have $r_1, r_2 \in L^*[\rho, \chi \cup \{y\} - \rho]$. Since $\chi' = \chi \cup \text{ffi}([\rho, \chi], s)$, we have $\chi' = (\chi \cup \{y\} - \rho) \cup \text{ffi}([\rho, \chi \cup \{y\} - \rho], s')$. By induction hypothesis, $r_1, r_2 \preceq s'$. Then $xy. r_1, r_2 \preceq xy. s'$ by repeated application of $\text{L3}, \text{L5}$, we have $r \preceq xy. r_1, r_2$. The result follows by transitivity of $\preceq$.

4. $s = \text{x}(y).s'$: Then $x \in \rho$, and $y \notin \rho \cup \chi$. Since $s \in L^*[\rho, \chi]$, we have $s' \in L^*[\rho, \chi \cup \{y\}]$. Now, we can show

$$O([\rho, \chi], s) \equiv (\nu y)(\text{xy} \mid O([\rho, \chi \cup \{y\}], s'))$$

There are two possible cases depending on whether $\text{xy}$ fires or not. We consider only the case where it does not fire, the other is similar. Since $\text{xy}$ never fires, all the interactions are performed by $(\nu y)(O([\rho, \chi \cup \{y\}], s'))$, that is

$$\langle \langle (\nu y)(O([\rho, \chi \cup \{y\}], s')) \rangle \rangle_{\rho}^{\nu y}$$

During the computation above, $y$ may be alpha renamed to other names. Furthermore, either the name is exported through some other message, or never exported at all. The second case is simpler; so we only consider the case where the name is exported at some point as a fresh
name \( w \), i.e. \( \tau = \tau_T \cdot \pi(w) \cdot \tau_T \) where \( w \notin \text{rcp}([\chi', \rho], \tau_T) \cup \text{ext}([\chi', \rho], \tau_T) \). Then we can deduce

\[
\langle O([\rho, \chi \cup \{ w \}]), s'\{w/y\}\rangle^\chi_{\rho} \triangleleft_{\mu\mu} \tau_T^w \tau_T.
\]

where \( \tau' = \tau_T \cdot \pi(w) \cdot \tau_T \). Further, since \( r \in L^\ast[\rho, \chi] \), we have \( r' \in L^\ast[\rho, \chi \cup \{ w \}] \). It is easy to show that since \( w \notin \rho \cup \chi' \), we have \( s'\{w/y\} \in L^\ast[\rho, \chi \cup \{ w \}] \). Since \( \chi' = \chi \cup \text{ff}([\rho, \chi], s) \), we have \( \chi' \cup \{ w \} = (\chi \cup \{ w \}) \cup \text{ff}([\rho, \chi \cup \{ w \}, s'\{w/y\}] \). By induction hypothesis, \( r' \preceq s'\{w/y\} \).

Then \( x(w).r' \preceq x(w).s'\{w/y\} \). By repeated application of L3, L5, we have \( r.xw \preceq x(w).r' \), and by L2, we have \( r \preceq r.xw \). The result follows by transitivity of \( \preceq \), and that \( s \) is alpha equivalent to \( x(w).s'\{w/y\} \).

\[ \square \]

Following is the alternate characterization of may preorder.

**Theorem 13** \( C_1 \triangleleft_{\rho} C_2 \) if and only if \( C_1 \preceq_{\rho} C_2 \).

**Proof:** Let \( [\rho_1, \chi_1] \) be the minimal interface of \( C_1 \) and \( [\rho_2, \chi_2] \) that of \( C_2 \).

**(if)** Let \( C_1 \preceq_{\rho} C_2 \), and \( C_1 \text{ may } O \). We have \( \rho_1, \rho_2 \subset \rho \). Let \( \chi = (\chi_1 \cup \chi_2) - \rho \). Let \( [\rho', \chi'] \) be the minimal interface of \( O \), \( \rho' = \rho'' \cup \chi \) and \( \chi' = \chi'' - \chi \). Then \( O : [\rho', \chi'] \), and since \( \rho'' \cap \rho = \emptyset \) we have \( \rho' \cap \rho = \emptyset \). From Lemma 30, it follows that the computation \( C_1|O \rightarrow C'_{\pi\mu} \) can be unzipped into \( \langle C_1 \rangle^\rho_{\chi'}^\chi \Rightarrow \pi \) and \( \langle O \rangle^\rho_{\chi'}^\rho \Rightarrow \langle O' \rangle^\rho_{\pi\mu}^\rho_{\chi''} \Rightarrow \pi^\mu \). Then \( \langle C_2 \rangle^\rho_{\chi'}^\chi \Rightarrow \pi \) for some \( r \preceq s \), and \( \langle O \rangle^\rho_{\chi'}^\rho \Rightarrow \pi^\mu \). Since \( \mu \notin \rho \cup \chi \), we have \( r.\mu \mu, s.\mu \mu \in L^\ast[\rho \cup \{ \mu \}, \chi] \), and we can show by induction on the length of a derivation of \( r \preceq s \) that \( r.\mu \mu \preceq s.\mu \mu \). By a similar induction we can show that, since \( \pi^\mu \in L^\ast[\rho', \chi'] \) and \( r \in L^\ast[\rho, \chi] \) we have \( \pi^\mu, \pi^\mu \in L^\ast[\rho', \chi'] \). Then by Lemma 35, \( \langle O \rangle^\rho_{\chi'}^\chi \Rightarrow \pi^\mu \Rightarrow \pi^\mu \) and we can zip up these computations to produce \( C_2|O \rightarrow C'_{\pi\mu} \). Hence \( C_2 \text{ may } O \).

**(only if)** Let \( C_1 \triangleleft_{\rho} C_2, \chi = (\chi_1 \cup \chi_2) - \rho \). Let \( \langle C_1 \rangle^\rho_{\chi'}^\chi \Rightarrow \pi \). Let \( \chi' = \chi \cup \text{ff}([\rho, \chi], s) \). It is clear from Definition 26 that \( \langle O([\rho, \chi], s) \rangle^\chi_{\rho} \Rightarrow \pi^\mu \). We can zip these up to get \( C_1 \text{ may } O([\rho, \chi], s) \), and therefore \( C_2 \text{ may } O([\rho, \chi], s) \). The computation \( C_2|O([\rho, \chi], s) \Rightarrow \pi^\mu \) can be unzipped into \( \langle C_2 \rangle^\rho_{\chi'}^\rho \Rightarrow \pi^\mu \Rightarrow \pi^\mu \), for some \( r \in L^\ast[\rho, \chi] \). Then by Lemma 36, \( r \preceq s \), and hence \( C_1 \preceq_{\rho} C_2 \). \( \square \)

This alternate characterization can be further strengthened to set inclusion in the case of \( A \pi \).

This is a consequence of Lemma 38 which is not true in Boreale’s setting. Note that Theorem 14 renders the interaction-path preorder to a proof tool rather than a part of the characterization.
Lemma 37 For $s \in \mathcal{L}^*[\rho, \chi]$

1. $rcp([\rho, \chi], s) \cup ext([\rho, \chi], s) = n(s) \cup \rho \cup \chi$, and

2. $s = s'.\alpha$ implies $bn(\alpha) \cap (n(s') \cup \rho \cup \chi) = \emptyset$.

Proof: By induction on $s$. \hfill \Box

Lemma 38 Let $r, s \in \mathcal{L}^*[\rho, \chi]$. Then $r \prec s$ implies $s \prec r$.

Proof: There are five cases for $r \prec s$ one for each law in Table 5.4. We consider only (L5) in detail. We have $r = s_1.(\hat{v})\overline{w}.(\hat{y})xy.s_2$, $s = s_1.(\hat{y})xy.(\hat{v})\overline{w}.s_2$, and $u, v \notin \hat{y}$. Applying Lemma 37 to $s$, we have $x, y \notin \hat{v}$. Then by (L5) $s \preceq r$. For the other cases, the reader may verify that (L1) and (L2) complement each other, and so do (L3) and (L4). \hfill \Box

Theorem 14 Let $[\rho_1, \chi_1]$ be the minimal interface of $C_1$ and $[\rho_2, \chi_2]$ that of $C_2$. For $\rho$ such that $\rho_1, \rho_2 \subset \rho$, if $C_1 \ll C_2$ and $\chi = (\chi_1 \cup \chi_2) - \rho$, then $\ll^{\rho} \Rightarrow$ implies $\ll^{\rho} \Rightarrow$.

Proof: Let $\ll^{\rho} \Rightarrow$. Then $\ll^{\rho} \Rightarrow$ for some $r \preceq s$. From Lemma 38, we have $s \preceq s$. From Lemma 35, we conclude $\ll^{\rho} \Rightarrow$. \hfill \Box

5.9 Variants of $A\pi$

We now present the two variants of $A\pi$ without mismatch. First, we consider the restricted version which allows comparison of a received name with only a local name. For this variant, the $COND$ rule is replaced by

\[
CASE: \quad \forall 1 \leq i \leq n \rho_i; f_i \vdash C_i \\
(\cup_i(\rho_i \cup y_i)); f \vdash case \: x \: of \: (y_1 : C_1, \ldots, y_n : C_n) \\
\quad \text{if} \quad y_i \neq y_j \text{ for } i \neq j, \text{ and} \\
\quad f_i \text{ are mutually compatible}
\]

where

\[
f(x) = \begin{cases} 
(f_1 \oplus f_2 \oplus \ldots \oplus f_n)(x) & \text{if } x \in \cup_i \rho_i \\
\bot & \text{otherwise.}
\end{cases}
\]

Note that unlike in $COND$, $y_i$'s are included in the set of actors of the resulting configuration, thus making them local. This ensures that $y_i$'s are constants because by the $ACT$ rule they cannot be
bound by input prefixes, and hence cannot be received names. The case construct can be seen as a macro for the following π-calculus term.

\[
\text{[case } x \text{ of } (y_1 : C_1, \ldots, y_n : C_n)] = [x = y_1][C_1] \ldots [x = y_n][C_n]
\]

We can not use this translation directly in Aπ instead of the case construct because it need not be well-typed. Although only one of the \(C_i\)'s is activated because \(y_i\)'s are distinct, the type system is not clever enough to accept the translation. Specifically, since more than one \(C_i\) may contain the same actor, the COMP rule could be violated. It is possible to enhance the type system to accept such terms; but that would make the type system significantly complex. We therefore adopt the simpler approach of using a multi-branching case statement.

For the variant with general match that allows comparison of any two names, the COND rule is replaced with

\[
\text{MATCH: } \forall 1 \leq i \leq n \rho_i; f_i \vdash C_i \\
(\cup_i \rho_i);(f_1 \oplus f_2 \oplus \ldots \oplus f_n) \vdash \text{if x is } (y_1 : C_1, \ldots, y_n : C_n) \\
\text{if } f_i \text{ are mutually compatible}
\]

Unlike in case, \(y_i\)'s can be received names and therefore, it is possible that more than one branch is true during the match. In such cases, one of the branches is non-deterministically chosen. Thus, we deviate slightly from the Actor model by allowing actor behaviors to be non-deterministic. But this is only internal non-determinism because the choice can not be influenced by external interactions. In fact, the if construct can be seen as a macro for the following π-calculus term that does not involve the choice operator.

\[
\text{[if x is } (y_1 : C_1, \ldots, y_n : C_n)] = \\
(\nu u, v_1, \ldots, v_n)([x = y_1]\bar{u}v_1]\ldots[x = y_n]\bar{u}v_n] \\
u(w).([\text{case } w \text{ of } (v_1 : C_1, \ldots, v_n : C_n)]) \quad \text{u, v, w fresh}
\]

Note that this encoding can not be used directly in Aπ instead of if, because it will not type check when \(\cup \rho_i\) contains more than one element (violates the ACT rule). For the same reason given for case we need a new construct to simplify the type system. Lemmas 24 and 25 are true for both variants of Aπ.
For reduction semantics of the variants, the structural congruence rules of \( \mathsf{A}\pi \) are left unchanged, but the reduction rules are changed by replacing \( \mathsf{IF} \) and \( \mathsf{ELSE} \) rules with one of the following:

\[
\begin{align*}
\text{BCASE:} & \quad \text{case } x \text{ of } (y_1 : C_1, \ldots, y_n : C_n) \to C_i \quad \text{if } x = y_i \\
\text{BMATCH:} & \quad \text{if } x \text{ of } (y_1 : C_1, \ldots, y_n : C_n) \to C_i \quad \text{if } x = y_i
\end{align*}
\]

Both Lemma 26 and Theorem 10 hold for the variants. The definition of may testing for variants is the same as that for \( \mathsf{A}\pi \), and Theorem 11 holds for both. The labeled transition system for variants is also the same, and all lemmas and theorems in Section 5.7 hold in both.

For the alternate characterization of may testing in the variant with general match, we need to weaken the trace preorder (relate more paths) by adding some laws to Table 5.4. Since without mismatch capability an observer cannot fully discriminate between free and bound outputs (of the configuration it observes), we need the following law.

\[\text{(L6) } s_1.\mathcal{F}w.(s_2\{w/y\}) \prec s_1.\mathcal{F}(y).s_2\]

Note that substitutions such as the above may lead to internalization of messages. Specifically, if an observer receives the name of one of its own actors instead of a fresh name, the messages that the observer sends to the argument will now be internalized and cannot be consumed by the environment. Furthermore, these internalized messages can themselves be consumed by the observer in a successful computation. We account for these possibilities by a new law.

\[\text{(L7) } s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.\mathcal{F}y.s_2 \quad \text{if } (\hat{y})s_2 \text{ is defined}\]

where \((\hat{y})s_2\) is \(s_2\) if \(\hat{y}\) is empty, and otherwise is the path obtained from \(s_2\) by binding the first free occurrence of \(y\) as the argument of an input action. If the first free occurrence of \(y\) is not the argument of an input action then \((\hat{y})s_2\) is undefined. With these new laws, Theorem 13 holds for the variant with \(\ll\rho\) defined as in Definition 25. However, Lemma 38 and Theorem 14 do not hold. We note that laws \textbf{L6} and \textbf{L7} also appear in the alternate characterization of may testing for asynchronous \(\pi\)-calculus with match operator [5]. In fact, the characterizations for the two calculi are essentially the same.
For the variant with restricted match, the usual approach for characterization does not work for several reasons. First, since an observer can only match a name against its local names, it cannot fully discriminate between outputs containing local names and outputs containing non-local names. Thus, in contrast to $L_6$, any output argument that is not a local name (not just bound names) can be substituted with arbitrary names. Second, different such outputs with the same argument can be substituted with different names, because the observer can not compare two received names with each other. Furthermore, since different observers can use names they receive in different ways to send messages, the result of such general substitutions is one of many possible paths depending on the data flow in the observer’s computation.

We demonstrate the need for a different approach to characterization through an example. Consider $C_1 = (\nu u)(\nu u, y(u)|u(w)|w).$ That can exhibit $s = \varphi(u, y(u, w, w).$ The following observers can be satisfied by $s$:

\[
O_1 = (\nu w)(x(t)\cdot \varphi t|y(t').0|w(v), \mu = w|\varphi t\mu)
\]
\[
O_2 = (\nu w)(x(t).0|y(t').0|w(v), \mu = w|\varphi t\mu)
\]

Let $C_2 = (\nu u_1, u_2)(v(t).v(t').(\nu t|y(t)|\nu u_1|\nu u_2|u_1(w)|w).$ Now, $C_2$ can satisfy $O_1$ with $r_1 = \varphi(u_1, y(u_2).u_1(w).w).$ and $O_2$ with $r_2 = \varphi(u_2, y(u_1).u_1(w).w).$ But, it cannot exhibit a single path that can satisfy both $O_1$ and $O_2$. In fact, it is the case that $C_1 \preceq \varnothing C_2$. This example shows that the alternate characterization of $C_1 \preceq \varnothing C_2$ should only require that, for a given path $s$ that $C_1$ exhibits, $C_2$ can exhibit a set of paths $P$ such that if $s$ satisfies an observer $O$ then there is a path $r \in P$ that can satisfy $O$. The choice of $r \in P$ depends on the dataflow in a successful computation of $O$. In comparison, the characterizations for the other two variants imposed the additional constraint that $P$ is a singleton, which is too strong for the variant with restricted match.

We can formalize the concepts above through the notions of templates and matches relation. A template $t^\circ$ of path $s$ represents the dataflow of non-local names in an observer that exhibits $\varphi$. The template explicit represents the dependencies from outputs in $s$ with non-local names as arguments (inputs received by the observer) to names (targets and arguments) in input actions of $s$ (outputs emitted by the observer). Clearly, there can be several possible templates of a path. We
say a path $s$ matches template $t^\circ$ if it can satisfy an observer that exhibits the dataflow $t^\circ$ during a successful computation. For the alternate characterization, we first define for a path $s$ and set of names $\rho$, the set $T(s, \rho)$ as the set of all possible templates that can be obtained from $s$, assuming names in $\rho$ as non-local. This covers all possible dataflows in any observer’s computation while exhibiting $\overline{s} \mu \mu$. We say $C_1 \preccurlyeq_\rho C_2$, if for each path $s$ that $C_1$ can exhibit, $C_2$ can exhibit a set of paths $P$ such that for any template $t^\circ \in T(s, \rho)$ there is a path $r$ in $P$ such that $r$ matches $t^\circ$. Formal definition of these concepts and proof that this characterization is correct is omitted here due to space constraints.

### 5.10 Related Work

A possible direction of future work is to give a complete axiomatization for finite configurations, i.e. configurations that do not use recursive behavior definitions. Preliminary results for characterization of may testing for variants of $A\pi$ with different name matching capabilities have been given in [57]. These results have also been extended in [56] to get a characterization for $L\pi$ [39].

We have not considered fairness property of the Actor model in this paper as it does not affect the notion of may testing. May testing is concerned only with the occurrence of an event after a finite computation, while fairness requires eventual delivery of messages, thereby affecting only potentially infinite computations. An interesting consequence of fairness is that must equivalence implies may equivalence, which was shown for a specific Actor based language in [2]. It can be shown by a similar argument that this result holds in $A\pi$ also.

Several calculi [17, 21, 31, 51] and programming languages [2, 17] have been proposed for actors. Since these works were motivated by different reasons, such as design of high-level languages or type systems for certain generic problems in object-oriented languages, their systems are not faithful to the pure Actor model [1]. For instance, they either are equipped with high level programming constructs that are not intrinsic to actors, or ignore actor properties such as uniqueness and persistence. Furthermore, these systems are not directly comparable to $\pi$-calculus. In contrast, our aim was to investigate a theory for the pure Actor model and compare it with that of Asynchronous $\pi$-calculus.

Notions of equivalence and semantic models have been studied for actors, such as asynchronous
bisimulation [21], testing equivalences [2], event diagrams [11], and interaction paths [55]. We have not only related may testing [2] to the interaction paths model [55], but also related our characterizations to that of asynchronous $\pi$-calculus given in [5].
Chapter 6

SynchNet: Meta-level Coordination of Objects

6.1 Introduction

In this chapter we take a programming language design approach to specifying the coordination of a group of distributed objects. The language we propose, SynchNet, is a meta-level language that can be used to extend any programming language that supports object-based programming. SynchNet follows the same design principles as Frølund’s Synchronizers [19]: It separates the specification of coordination constraints and policies from the functional behavior of objects. Unlike Synchronizers which have the expressive power of high-level languages, SynchNet is based on the Petri Net [47] model of concurrent computation and synchronization. Using SynchNet, one can specify the policies for coordinating a group of objects in an abstract entity called a synchronizer. The specification of a synchronizer is translated into a Petri Net, with a slightly different semantics, called synchronizing net or synchnet. A two-level semantics relates the execution of the synchnet to method invocations in coordinated objects and thus allows enforcement of coordination requirements as specified.

The formal language of Petri Nets allows us to give formal definitions of interesting properties for synchnets. In particular, we define a preorder relation on synchnets that states when it is safe to replace a deadlock-free synchnet with an alternative implementation while preserving the coordination properties of the first synchnet. Using this relation, one can verify the correctness of a synchnet implementation with respect to a more abstractly defined synchnet.

All the communication and coordination mechanisms mentioned thus far suffer from a software engineering deficiency, namely the inter-twining of coordination behavior with the computational
aspects of a system. There have been a number of proposals for modular specification, which separate the specification of the coordination from the computation aspects, there have been many proposals for modular specification [19, 12, 23, 44, 60]. The primary focus of these proposals has mainly been on the software engineering benefits obtained from a separation of concerns, such as reuse and customizability. Our proposal, while fitting in this category of work, further attempts to limit the expressivity of the language to the extent that available formal tools and theories for analysis and verification become applicable.

A useful aspect of our proposed framework is that the compiler for SynchNet automatically generates distributed code from the specification of a synchronizing net. The generated code, which is interwoven with the coordinated objects’ code, uses the communication primitives available in the base-language. In this sense our framework can also be placed in the more general scheme of aspect oriented programming [32], in which a separately specified aspect of a program’s behavior is automatically “woven” into the code that implements the basic functionality of the program.

SynchNet can also be used in specifying synchronization constraints on the order of method invocation for a single object. It is known, however, that synchronization constraints often conflict with inheritance in concurrent object-oriented languages. This phenomenon is generally known as inheritance anomaly. Many linguistic solutions have been proposed to counter the inheritance anomaly. Matsuoka and Yonezawa provide a rather comprehensive analysis of the problem and compare various proposed solutions in [37]. They distinguish three causes for the inheritance anomaly in this paper and show that most proposals fail to consider all there causes, and then present a satisfactory solution. We believe that SynchNets, too, successfully avoids the three causes of inheritance anomaly, despite the simplicity of their syntax and semantics.

In Section 2 we present basic definitions of the Petri Net theory. In Section 3, we motivate our approach in designing a new coordination language and present some examples. In Section 4 we present the syntax and semantics of SynchNet. We also present several examples to illustrate the expressive power of the language. Section 5 defines a refinement relation that states when it is safe to replace a synchronizing net with another one.
6.2 Languages for Coordination of Distributed Objects

Designing linguistic primitives and styles for distributed coordination, that is coordination of systems in which the primary mode of communication is asynchronous message passing, has a long history. We discuss highlights of this evolution by first considering low-level mechanisms, and then move towards more abstract and modular constructs.

Most languages for programming communicating processes contain two operations send and receive that communicate data over channels connecting communicating processes. Usually a process executing a receive operation on a channel blocks until a message is available on the channel. Sending processes, however, may either block until a receiver is available to receive the message (synchronous mode), or proceed with their execution, leaving the message in channel’s buffer for the receiving process to pick it up later (asynchronous mode).

To protect blocked processes from remaining blocked whenever a channel remains empty indefinitely, the input-guarded command was introduced. The input-guarded command is an extension of Dijkstra’s guarded command [15] with added conditions to check availability of messages on a channel. This construct was introduced in the language Communicating Sequential Processes by Hoare [28]. CSP uses synchronous communication, but it is not difficult to conceive of input-guarded commands in a language with asynchronous communication primitives.

A more structured and higher-level construct, which is also based on input-guarded command is Ada’s rendez-vous mechanism [14]. Rendez-vous hides a pair of message-based communications behind an abstraction similar to a procedure call. Ada combines this procedure-like abstraction with input-guarded commands into an elegant and powerful coordination mechanism.

A practical communication abstraction, which is similar to rendez-vous, but can be virtually used with any procedural language is Remote Procedure Call (RPC). RPC was first introduced in the programming language Distributed Processes (DP) by Brinch Hansen [25]. RPC implementations slightly modify the procedure call semantics by translating a call/return into a pair of message communications, somewhat similar to rendez-vous. RPC is less flexible than rendez-vous as it does not allow receiving processes to choose the channel from which to receive messages. The RPC framework would require the programmer to write extensive code to avoid deadlock situations. Yet, the simplicity and efficiency of RPC has turned it into a widely used mechanism in practice.
Inspired by this success, some object-oriented languages, such as Java, extended their method invocation semantics in a similar fashion to a distributed version called Remote Method Invocation (RMI).

6.3 Petri Nets

We begin this subsection by an informal introduction to the Petri Nets model and its graphical notation. A formal definition of the model will be given later. A well-written exposition on Petri Nets can be found in [46]. We also argue why Petri Nets by itself is not a suitable language for the development of distributed systems.

The graphical presentation of a Petri net is a graph with nodes and arcs connecting the nodes. There are two kinds of nodes: places, which usually model resources or partial state of the system, and transitions, which model state transition and synchronization. Arcs are directed and always connect nodes of different types. Multiple arcs between two nodes are allowed. Figure 6.1 is an example net.

![Figure 6.1: A Petri Net Example](image)

In a Petri Net, the state of the system is modeled by marking the places of the net with tokens. A place can be marked with a finite number (possibly zero) of tokens. For instance, in Figure 6.1 the
token in place $m$ represents the availability of a semaphore. The semantics of transitions determine how the state of the system changes. A transition $t$ is said to be enabled in a certain marking if for every arc from a place $p$ to $t$, there exists a distinct token in the marking of $p$. For instance, in Figure 6.1, the two transitions $t1$ and $t2$ are enabled. An enabled transition can fire and result in a new marking. Firing of a transition $t$ in a marking $\mu$ is an atomic operation that subtracts one token from the marking of any place $p$ for every arc connecting $p$ to $t$, and adds one token to the marking of any place $p$ for every arc connecting $t$ to $p$. For instance in Figure 6.1 the transition $t1$ can fire and as a result change the marking of the net by removing one token from $m$ and one token from $p1$ and putting one token in $p2$. It is also possible for transition $t2$ to fire. The choice is made non-deterministically.

Petri Nets is not a suitable model for distributed object-based programming. In Petri Nets asynchronous computation is represented naturally, but only synchronous communication can be modeled directly. Modeling asynchronous communication requires explicit representation of communication channels. This renders Petri Nets unfit for distributed programming. Almost every distributed programming language hides channel and buffering representations and only provides high-level primitives for communication and synchronization. Another disadvantage is that Petri Nets are not capable of directly expressing creation of new processes or objects (More expressible extensions are available but they lack the nice decidability properties of classical Petri Nets.)

6.4 SynchNet: Two Motivating Examples

Our object-based model of distributed computation is inspired by the Actor model [1]. We assume each object is identified by a unique reference. Objects communicate by an asynchronous communication mechanism called ARMI (Asynchronous Remote Method Invocation). Physical locations of objects are not modeled explicitly and hence all communications are uniformly assumed to be remote. In ARMI, the source object asynchronously sends a message specifying the method of the target object to be invoked accompanied by the arguments to be passed. Messages are guaranteed to reach the target object free from error and are buffered in the target object’s mailbox. No assumption is made on the order of message arrival. A local scheduler selects a message from the mailbox and invokes the specified method using the message content as arguments. Objects are
single threaded and at most one method invocation can be in progress at any moment. According
to this model, synchronizers are specifications that dictate the behavior of schedulers.

ARMI is similar to the remote method invocation model used in many distributed object-based
languages and platforms such as CORBA [24], DCOM [53], and Java RMI [34]. The difference is
that our model of invocation is asynchronous. The usual remote method invocation (RMI) is a
rendez-vous like communication mechanism, in which the source object blocks until the method
execution is complete and returns with a message containing the result. In ARMI, the source does
not wait for the invocation to begin. When an invoked method reaches the end of its execution, it
may choose to send back the result using a separate ARMI. Hence, it is possible to model RMI as a
pair of ARMI communications. Therefore, our results can be incorporated into practical platforms
that use RMI like communication mechanisms.

We propose a two-level language for coordination of distributed objects communicating via
ARMI. The base language can be any conventional sequential class-based language such as Java or
C++, with the method invocation semantics modified to be ARMI. The meta language is SynchNet.
In SynchNet, coordination patterns are specified as modules. Each module is translated into a so
called synchronizing net or synchnet, which is a Petri net with a slightly modified semantics that
relates the transitions of the net to method invocations in the base objects. After a brief overview
of Petri Nets, we motivate our work via two classical coordination problems.

6.4.1 Distributed Mutual Exclusion

We state a coordination problem and write a SynchNet module to solve it. A group of transmitters
are scattered in a field to transmit sensed data. Transmitters communicate with one another via
asynchronous sending of messages. The delivery of messages triggers invocation of methods in
the objects that control the transmitters. Each transmitter is controlled by an object with two
methods: An on method takes an argument that determines transmission power and turns on the
transmitter, and an off method that turns it off. A global requirement is that no two transmitters
may be transmitting at the same time. It is therefore necessary that off messages be sent to turn
off the transmitters before the next transmission begins. We abstract away the distributed logic
that decides on when and to which transmitter on and off messages must be sent, and try to

86
coordinate the global order of message delivery so that two conditions are guaranteed: (I) on and off messages are delivered to each object in alternation, (II) no two transmitters are transmitting simultaneously.

Suppose controller objects are instances of the class TransmitterC and that Transmitters is a list containing references to the identifiers of a collection of controller objects. The following module specifies the two requirements stated above:

```
synchnet TransmitterME(Transmitters: list of TransmitterC)
  init = { ob'.off | ob' in Transmitters}

  foreach ob in Transmitters [with fairness]
    method ob.on
      requires {ob'.off | ob' in Transmitters}
      consumes {ob.off}
    method ob.off
      requires {ob.on}
      consumes {ob.on}
  end TransmitterME
```

To generate and install a synchnet according to the specification of TransmitterME on a collection of objects G by issuing the statement TransmitterME(G) in the base-language. G is a list of object references on which the generated synchnet must be installed.

TransmitterME states that an on method can be invoked on object ob if every transmitter in the group is off. In Petri net terms, it states that ob.on may be invoked only when in the state of TransmitterME there is one ob’.off token available for each object ob’ in the group. Once the invocation of an ob.on is decided, the state of the generated synchnet is modified by adding one token corresponding to the invoked method (ob.on here), and consuming the tokens specified in the consumes multilist. Note that consuming ob.off here guarantees that no other on method is invoked unless the object ob is turned off again. The only requirement on invocation of an ob.off method is that ob is turned on. After consuming the token ob.on which indicates ob is on, other transmitters may get a chance to be turned on.
6.4.2 Dining Philosophers

Philosophers sitting on the left and share one fork with the philosopher on the right. Each philosopher needs two forks to eat but every philosopher has to share one fork with the philosopher around a round table and spend their time between thinking and eating.

Now we use SynchNet to solve a distributed version of the dining philosophers problem: a group of

Figure 6.2: Diagram of TransmitterME instantiated on t1 and t2 in its initial state
Suppose philosophers and forks are modeled as objects and communicate via asynchronous remote method invocation. Since we are concerned only with access to forks, we won't worry about methods in philosopher objects. Each fork may be accessed using one of four methods: pickX and putX (where X stands for either L or R) are to back invocations of the form pickX or of the form putX. A local requirement is that no two back invocations of the form pickX or of the form putX are allowed. There are two global requirements: fairness and deadlock-freedom. The following syntnet only depicts the part of the net corresponding to two adjacent forks. The rest of the diagram consists of similar boxes for the other forks and the whole diagram forms a cycle.

Figure 6.3: Partial diagram of philosophers.
synchnet Philosophers(N: int, Forks : array[1..N] of Fork)

init = {fork.putL, fork.putR | fork in set(Forks)}

foreach fork in set(Forks) [require fairness]

    method fork.pickL
        requires {fork.putL, fork.putR}
        consumes {fork.putL}
    
    method fork.pickR
        requires {fork.putL, fork.putR}
        consumes {fork.putR}
    
    method fork.putL
        requires {fork.pickL}
        consumes {fork.pickL}
    
    method fork.putR
        requires {fork.pickR}
        consumes {fork.pickR}

foreach i in {1 .. N}

    atomic(Forks[i].pickL,Forks[(i mod N) + 1].pickR)

end Philosophers

6.5 Syntax and Semantics of Synchronizing Nets

We first present the concrete syntax, followed by a translational semantics of modules into synchnets which are in fact Petri nets. Then, we present the semantics of the two-level language by relating the firing semantics of synchnets to method invocation in base-level objects. Finally, we define an inheritance mechanism that allows extension of a module with new constraints.
6.5.1 Syntax of Synchnets

We first present the core syntax of SynchNet. Later, we will introduce more advanced constructs as syntactic sugar. A SynchNet module has the following general form:

```plaintext
synchnet ⟨id⟩ ( ⟨param-list⟩ ) is
  init = ⟨token-list⟩
  ⟨method-clauses⟩
  ⟨atomicity-clauses⟩
end ⟨id⟩
```

where ⟨id⟩ is a programmer supplied identifier used to refer to the defined synchnet. ⟨param-list⟩ is a list of formal parameters along with their types:

```plaintext
⟨param-list⟩ ::= ⟨var1⟩ : ⟨type1⟩, ..., ⟨varn⟩ : ⟨typen⟩
```

All ⟨vari⟩'s must be distinct. ⟨typei⟩s can be any type that is available in the base language, including both simple types and aggregate types such as lists or arrays. In particular, they can include class types if ⟨vari⟩ is supposed to be an object. ⟨param-list⟩ acts as a binder for the specified variables, with their static scope being the body of the synchnet.

The body of a module consists of a clause specifying the initial state of the corresponding synchnet and a collection of guard-transitions. Syntactically, the state of a synchnet is represented as a multilist of tokens. Tokens correspond to the methods of the objects whose references are passed to the synchnet and are supposed to be coordinated by the synchnet. The syntax of tokens and multilists of tokens is specified as

```plaintext
⟨token⟩ ::= ⟨var⟩.⟨method⟩
⟨token-list⟩ ::= {⟨token⟩, ..., ⟨token⟩}
              | {⟨token⟩ | ⟨predicate⟩}
              | ⟨token-list⟩ union ⟨token-list⟩
              | ⟨token-list⟩ intersect ⟨token-list⟩
```
where \( \langle \text{var} \rangle \) must be a variable whose type is a class and \( \langle \text{method} \rangle \) must be a method identifier belonging to that class. \( \langle \text{predicate} \rangle \) is a predicate over object references. Predicates consists of equality or inequality constraints over variables whose types are classes, composed with the usual boolean operators. The collection of tokens inside the brackets in the expression \( \{ \langle \text{token} \rangle, \ldots, \langle \text{token} \rangle \} \) must be treated as a multiset, that is, the order of tokens is irrelevant, and the same token may appear more than once.

The body of a module consists of two kinds of coordination behavior. The first kind, \( \langle \text{method-clauses} \rangle \) consists of a collection of method clauses. Each method clause has a header and a body. The header of the clause consists of a variable (with a class type) and a method identifier belonging to that class. The body of a method clause consists of a list of guard-transitions. A guard-transition has two parts: A guard, which is the condition required for the method to be invoked, and a transition that specifies how the state of the synchnet must change if the method is invoked. Each method clause is written as

\[
\text{method } \langle \text{var} \rangle. \langle \text{method} \rangle \\
\hspace{1cm} \text{requires } \langle \text{token-list} \rangle \\
\hspace{1cm} \text{consumes } \langle \text{token-list} \rangle \\
\text{or } \ldots \\
\text{or } \\
\hspace{1cm} \text{requires } \langle \text{token-list} \rangle \\
\hspace{1cm} \text{consumes } \langle \text{token-list} \rangle \\
\]

A syntactic requirement is that the consume list must be contained in the require list extended with an additional token corresponding to the method for which the require-consume pair is specified.

Atomicity is another kind of coordination requirement that can be specified in the body of a module. Atomicity requirements are represented as constraints called atomicity clauses. Each atomicity clause has the following format:

\[
\langle \text{atomicity-list} \rangle := \text{atomic}(\langle \text{var}_1 \rangle. \langle \text{method}_1 \rangle, \ldots, \langle \text{var}_n \rangle. \langle \text{method}_n \rangle)
\]

where all \( \langle \text{var}_i \rangle \)'s must refer to distinct objects.
To avoid repetitive declarations, we introduce a universal quantification operator as syntactic sugar.

\[
\text{foreach } (\text{var}) \text{ in } (\text{var-set}) \\
(\text{clauses})
\]

where all clauses in \((\text{clauses})\) use the variable \((\text{var})\) in their headers. \((\text{var-set})\) is a subset of variables specified as formal arguments to the synchnet. The variable must be of class type or of aggregate types such as arrays or lists with elements being of class type. This syntactic form is equivalent to a list of clauses obtained by making copies of clauses in \((\text{clauses})\) each having \((\text{var})\) replaced with some variable in the set \((\text{var-set})\).

### 6.5.2 Translating SynchNet modules to Synchnets

Let’s first formalize the Petri Net model introduced in Section 2.1.

**Definition 27** Formally, a Petri net \(N\) is a four-tuple \((P, T, I, O)\) where \(P\) is a finite set of places, \(T\) is a finite set of transitions. \(P\) and \(T\) are disjoint. \(I: T \rightarrow P^\infty\) is a mapping from transitions to multisets of places. Similarly, \(O: T \rightarrow P^\infty\) is a mapping from places to multisets of transitions. □

The following is the formal definition of the graph of a Petri Net:

**Definition 28** The graph of a Petri Net \(N = (P, T, I, O)\) is a bipartite directed multi-graph \(G = (P \cup T, A)\) where \(A = \{a_1, \ldots, a_n\}\) is a multiset of directed arcs of the form \((p, t)\) or \((t, p)\) for \(p \in P\) and \(t \in T\). □

Now we can formalize the notions corresponding to execution of a Petri Net.

**Definition 29** A marking \(\mu\) of a Petri Net \((P, T, I, O)\) is a multiset of places. That is \(\mu \in P^\infty\). A transition \(t \in T\) is enabled in a marking \(\mu\) if \(I(t) \subseteq \mu\). An enabled transition \(t\) fires by subtracting \(I(t)\) from \(\mu\) and adding \(O(t)\). That is, firing of \(t\) results in a new marking \(\mu' = (\mu - I(t)) \cup O(t)\), where \(-\) and \(\cup\) are taken to be multiset operations. □

A synchnet is a Petri net and is generated when an expression of the form \(S(\mathcal{O})\) is evaluated in the base language, where \(S\) is the name of a module specified in the SynchNet language and \(\mathcal{O}\) is
a collection of base-level object references. Now, we formally define the Petri net that constitutes
the synchnet generated by evaluating the expression \( S(O) \) given the specification of module \( S \) in
SynchNet. We need to make a few assumptions before we describe the construction of a synchnet.

To avoid aliasing problems, all object references \( O \) passed in \( S(O) \) must be distinct. This
includes all the references contained in aggregate data structures such as arrays and lists. This
condition can be checked at run-time when the synchnet instance is generated. Let \( O \) be the
collection of all object references used to create an instantiation of the module \( S \) specified in the
general form below.

\[
\text{synchnet } S \ (V_1 : T_1, \ldots, V_n : T_n) \text{ is}
\]
\[
\text{init} = I
\]
\[
\ldots
\]
\[
\text{method } V_i.M_j
\]
\[
\ldots
\]
\[
\text{or}
\]
\[
\text{requires } R_{ijk}
\]
\[
\text{consumes } C_{ijk}
\]
\[
\text{or}
\]
\[
\ldots
\]
\[
\ldots
\]
\[
AC
\]
\[
\text{end } S
\]

For simplicity of presentation, let’s assume that all \( T_1, \ldots, T_n \) are class types. In the general case,
we ignore parameters which have a non-class type, and we expand aggregate parameters into a
collection of class-type variables. The type system of the base language can be used to verify that
for every pair of the form \( V_i.M_j \) the method \( M_j \) actually belongs to the class \( T_i \). We further
assume that an environment \( \eta : \{V_1, \ldots, V_n\} \rightarrow O \) is given that maps variable names to actual
object references passed during the creation of the synchnet instance. For a multiset of tokens \( Tok \),
we let \( \eta(Tok) \) be the multiset obtained by renaming every occurrence of \( V_i.M \in Tok \) by \( \eta(V_i).M \).
With these assumptions, we construct a synchronet for $S(\mathcal{O})$ in two steps. First we ignore atomicity clauses and define a Petri Net $SN = (P,T,I,O)$. If $AC$ is non-empty, we modify $SN$ to obtain a net $SN' = (P,T',I',O')$ that incorporates atomicity constraints specified by the list of atomicity clauses $AC$.

For a module $S$ with the general form described above, let the net $SN = (P,T,I,O)$ be defined as follows. We assume that the environment $\eta$ binds formal parameters ($Vi$) of $S$ to object references given in $\mathcal{O}$.

- For every pair of variable $V_i$ and method $M$ that belongs to the class of $V_i$ we consider a place $o.M$ with $o = \eta(V_i)$. That is $P = \{\eta(V_i).M | 1 \leq i \leq n \text{ and } M \text{ belongs to } Ti\}$.

- $T$ is the smallest set such that for every pair of require-consume clause $(Rijk,Cijk)$ that belongs to a method $Vi.Mj$, there exists a transition $t \in T$ such that

$$I(t) = \eta(Rijk) \text{ and } O(t) = (\eta(Rijk) - \eta(Cijk)) \cup \{\eta(Vi).Mj\}$$

If no require-consume pairs are specified for a method $Vi.Mj$, then we assume that there is a transition $t \in T$ such that

$$I(t) = \{\} \text{ and } O(t) = \{\eta(Vi).Mj\}$$

we call these transitions simple and we say the simple transition $t$ corresponds to the singleton $\{\eta(Vi).Mj\}$.

Now, suppose the body of the module $S$ contains atomicity clauses $AC_1, \ldots, AC_n$. We obtain a sequence of nets by gradually merging simple transitions into tuple transitions. We also keep track of merged simple transitions as the set $MT \subseteq T$, to remove them from the set of transitions after all atomic clauses are processed. Let $SN_0 = SN$ and $MT_0 = \emptyset$. For every atomicity clause $AC_i$ ($1 \leq i \leq n$) of the form

$$\text{atomic}(V_1.M_1, \ldots, V_l.M_l)$$

we modify the net $SN_j = (P,T_j,I_j,O_j)$ to obtain $SN_{j+1} = (P,T_{j+1},I_{j+1},O_{j+1})$ in the following way. Let $T_{j+1}$ be the smallest set containing $T_j$ such that for every collection of transitions $t_i \in T$
where \(1 \leq i \leq l\) and \(t_i\) corresponds to \(V_i.\mathcal{M}_i\), we have a *tuple* transition \((t_1, \ldots, t_l) \in T_{j+1}\). We say that \((t_1, \ldots, t_l)\) corresponds to the set \(\{\eta(V_1).\mathcal{M}_1, \ldots, \eta(V_l).\mathcal{M}_l\}\). We further let \(I_{j+1}\) and \(O_{j+1}\) be identical to \(I_j\) and \(O_j\), respectively, on transitions in \(T_j\), and for a new transition \((t_1, \ldots, t_l) \in T_{j+1}\), let \(I_{j+1} = I_j(t_1) \cup \cdots \cup I_j(t_l)\) and \(O_{j+1} = O_j(t_1) \cup \cdots \cup O_j(t_l)\). Finally, let \(MT_{j+1} = MT_j \cup \{t_1, \ldots, t_l\}\).

By repeating the above process, we obtain \(SN_n = (P, T_n, I_n, O_n)\) and \(MT_n\). Now, let \(SN' = (P, T', I', O')\) where \(T' = T_n - MT_n\) and \(I'\) and \(O'\) are restrictions of \(I_n\) and \(O_n\) to \(T'\). This completes our translation, and we have \(SN'\) as the Petri net of the synchnet \(S\).

The operational semantics of a synchronizing net \(SN = (P, T, I, O)\) is a labeled transition system \((\mathcal{M}, \mathcal{L}, T)\) where \(\mathcal{M} = P^\infty\) is the set of possible markings of \(SN\), \(\mathcal{L} = 2^P\) the set of labels with each label being a finite set of places, and \(T \subseteq \mathcal{M} \times \mathcal{L} \times \mathcal{M}\) defined as the smallest ternary relation such that if \(t \in T\) is enabled in marking \(\mu\), \(t\) corresponds to the set of methods \(L\), and \(\mu'\) is the marking that results after \(t\) fires in marking \(\mu\), then \((\mu, L, \mu') \in T\). We write \(\mu \xrightarrow{L} \mu'\) for such a triple.

It is possible to extend SynchNet to support disjunction of atomicity constraints. Extending the synchnet construction to account for this extension is straightforward and is similar to the construction for disjunction of require-consume clauses. Due to space limitation we do not provide the construction in this chapter.

### 6.5.3 Semantics of Coordination with Synchnets

The specification of a synchnet \(S\) is akin to a class declaration in class-based languages. To coordinate a group of objects, a synchnet must be created by the base-level program. To allow this, we extend the base language to include expressions of the form \(S(Params)\). Such expressions can be added as statements if the base language is imperative or as function applications if the language is functional. The evaluation or execution of such an expression creates a new instance of \(S\) and uses \(Params\) to initialize and set up the Petri net corresponding to \(S\). In general, \(Params\) would include references to newly created objects. A sanity check guarantees that references included in \(Params\) are all distinct. We require this sanity check to generate a synchnet unambiguously.

We now present the operational semantics of the two-level language. We use \(SI\) to refer to a created synchnet and assume that \(O\) is the set of object references coordinated by \(SI\). Suppose
denotes the state of a SI (a net marking), and \( \sigma_o \) the state of an object \( o \in O \). As stated before, we expect the base language to follow the asynchronous remote method invocation scheme for communication. We don’t make any assumption about the representation of the state of objects in the base language, but we assume that its formal semantics is defined as a labeled transition system with labels being either \( o.\tau \) referring to some internal computation by the object \( o \), or \( o.l(v_1, \ldots, v_n) \) where \( o \) is an object reference, \( l \) is the label of some method that belongs to object references by \( o \) and \( v_1, \ldots, v_n \) are actual values. The transition corresponds to the invocation of method \( l \) of object \( o \) with \( v_1, \ldots, v_n \) passed as arguments. We will use the abbreviation \( \tilde{V} \) for the list of values \( v_1, \ldots, v_n \).

The semantics of object execution in the two-level language is defined as a labeled transition system. Suppose a synchnet \( SI \) coordinates a group of objects \( O = \{ o_1, \ldots, o_n \} \). We let \( S = \{ (\sigma_{o_1}, \ldots, \sigma_{o_n}, \mu) \} \), where \( \sigma_{o_i} \) are the local states of objects \( o_i \), \( 1 \leq i \leq n \), and \( \mu \) is a marking of \( SI \), be the set of global states of objects \( o_i \) coordinated by \( SI \) (we will also use the abbreviation \( (\tilde{\sigma}, \mu) \)). Let \( L_i \) be the set of transition labels that are either \( o_i.\tau \) (silent or internal transition by \( o_i \)) or correspond to invocations of \( o_i \)'s methods in the base language. We define a labeled transition system on global states as a triple \((S, L, T)\) where \( L = L_1 \times \cdots \times L_n \) and \( T \subseteq S \times L \times S \). We use the abbreviation \( \tilde{l} \) for \( (l_1, \ldots, l_n) \), where \( l_i \in L_i \). We also write \( s \xrightarrow{\tilde{l}} s' \) for \((s, \tilde{l}, s') \in T \). The transition relation \( T \) is defined as the smallest relation satisfying the following rules

\[
\begin{align*}
\sigma_{o_i} & \xrightarrow{o_i.l(\tilde{V})} \sigma'_{o_i} & \mu & \xrightarrow{o_i.l} \mu' & \quad \begin{cases} o_j.\tau & \text{if } j \neq i \\ o_j.l(\tilde{V}) & \text{if } j = i \end{cases} \\
(\ldots, \sigma_o, \ldots, \mu) & \xrightarrow{\tilde{l}} (\ldots, \sigma'_o, \ldots, \mu')
\end{align*}
\]

\[
\forall 1 \leq i \leq n \quad (\tilde{\sigma}, \mu) \xrightarrow{\tilde{l}} (\tilde{\sigma}', \mu')
\]

In words, a message \( l(\tilde{V}) \) sent to object \( o \) can result in invocation of \( o.l \), only if the synchnet is
in a state that permits the invocation. Furthermore, if the invocation takes place, then the state of the synchnet changes accordingly.

6.5.4 Composition of Synchronizers

We can extend a synchnet specification by relaxing or further constraining the constraints specified in it. We do so via an inheritance mechanism. Suppose $S_1$ is a synchnet specification, we write

```plaintext
synchnet $S_2$ ( Params ) extends $S_1$ is
  init = $I$
  ...
  method V.M
    ...
    or
      requires [ intersect | union ] $R$
      consumes [ intersect | union ] $C$
    or
    ...
    ...
    atomic(V1.M1,...,Vl.Ml)
  ...
end $S_2$
```

as the specification of a synchnet $S_2$ that extends the specification of $S_1$. Parameters of $S_1$ must be exactly the same as those of $S_2$. $S_2$ may refer to the initial state of its parent synchnet by the expression `Super.init`. Therefore, $I$ can be either a multiset of tokens or the union or intersection of `Super.init` with a new multiset of tokens. The optional operators `intersect` or `union` can be used in require-consume clauses to relax or further constrain the requirements of the parent synchnet. If neither `intersect` nor `union` are specified, the multisets replace those of the parent multisets.

An independent specification for $S_2$ can be obtained by a simple substitution: `Super.init` is replaced with the initial multiset of tokens defined in $S_1$, and for every pair of require-consume
clauses of the form

... or
requires \( X \) \( R \)
consumes \( Y \) \( C \)
...

that belongs to method \( V.M \), and where \( X, Y \in \{\text{intersect, union}\} \), we replace the pair of clauses with

... or
requires \( R_1 \) \( X \) \( R \)
consumes \( C_1 \) \( Y \) \( C \)
or
requires \( R_2 \) \( X \) \( R \)
consumes \( C_2 \) \( Y \) \( C \)
or
...
or
requires \( R_n \) \( X \) \( R \)
consumes \( C_n \) \( Y \) \( C \)
or
...

where

method \( V.M \)
requires \( R_1 \)
consumes \( C_1 \)
or
...

99
is the complete set of require-consume clauses of V.M in S1. The set of atomicity clauses of unwound S2 is the union of atomicity clauses in S1 and those specified in S2.

6.5.5 More Examples

Here we present some examples to illustrate how our language may be used to modify the interactive behavior of single objects and create more familiar coordination mechanisms such as semaphores.

Example 1 In this example, we show how synchnets may be used to implement semaphores, another coordination mechanism. Suppose ob is some object with two methods put and get. For instance, ob can simply be a variable, with its content accessed via get invocation, and updated with invocations of put. The following synchnet will turn this object into a semaphore, in the sense that the number of times the put method is invoked always exceeds the number of times get is invoked. In other words, put will behave like the V operation of a semaphore and get like the P operation.

synchnet Sem(of : Variable)

init = { }

method of.put

requires {}

consumes {}

method of.get

requires {of.put}

consumes {of.put}

end Sem

The guard-transition for method of.get indicates that every invocation of get requires a distinct invocation of put to occur in the past. As every invocation of either methods put or get adds a new token to the corresponding places, the consume clause of get guarantees that the number of
invocations of `get` would not exceed the number of invocations of `put`. However, because `put` does not require any tokens, `put` maybe invoked any number of times regardless of how many times `get` is invoked. This is the usual invariant requirement for semaphores, which can be intuitively verified through the simple semantics of the synchronizer language. Adding fairness to the semantics allow definitions of fair semaphores.

**Example 2** A *k*-bounded semaphore has the property that at most *k* processes can issue a *V* operation that is not matched by a *P* operation. A 1-bounded semaphore can be defined easily by further constraining the behavior of a general semaphore.

```
syncnet OneSem(ob : Variable) extends Sem
  init = {ob.get}
  method ob.put
    requires {ob.get}
    consumes {ob.get}
  end OneSem
```

This synchronizer states that the method `put` may be invoked only if `get` had been invoked once in the past. To allow for the first `put` to go through we have modified the initial state to include a token of type `ob.get`. When this synchnet is installed on a single element variable, it turns the variable into a single-element buffer.

A *2*-bounded semaphore can be defined similarly:

```
syncnet TwoSem(ob : Variable)
  init = {ob.get,ob.get}
  method ob.put
    requires { ob.get }
    consumes { ob.get }
  method ob.get
    requires { ob.put }
    consumes { ob.put }
  end TwoSem
```
Alternatively the same semaphore can be expressed using synchnet inheritance:

```plaintext
synchnet TwoSem(ob : Variable) extends OneSem
    init = \{ob.get, ob.get\}
end TwoSem
```

Only the initial state is modified. The rest of the synchnet specification is inherited from `OneSem`.

**Example 3** Synchronizers for inherited objects may be defined compositionally using synchnet inheritance. Suppose a new class of objects `InfBuf2` is defined that adds a new method `get2` to an infinite buffer such that two elements of the buffer may be fetched at one time. The required coordination for the new class can be specified modularly and compositionally:

```plaintext
synchnet TwoBuf(A : Buffer) extends OneSem
    init = Super.init
    method A.get2
        requires \{A.put, A.put\}
        consumes \{A.put, A.put\}
end TwoBuf
```

**A Preorder on Simple Synchronizers**

One important property that we usually want a group of objects to have is freedom from deadlock. We define deadlock as the situation in which the state of one or more synchnet disables certain methods forever. This definition, of course, also includes the extreme case of a synchronizer disabling the invocation of all methods of an object; regardless of the behavior of the environment, this is an obvious deadlock situation.

One can verify deadlock-freedom of a synchnet by performing a reachability analysis. However, since reachability of Petri nets has non-elementary complexity, we introduce an alternative formal method for development of deadlock-free synchnets. We define a preorder relation on synchnets that allows “safe” substitution of a synchnet with an alternative implementation. In other words, we introduce a preorder relation ≤ over synchnet instances that is deadlock-freedom preserving:
$S \preceq S'$ implies that whenever $S'$ does not deadlock in an environment $E$, using $S$ in environment $E$ would not result in deadlock either.

The formal framework that we develop here is along the lines of the theory of failure equivalence of CSP processes presented in [28].

**Definition 30** A trace of a synchnet $S$ with initial state $I$ is a path in the labeled transition systems of the net with the root $I$. □

Next we define the *failure* of a synchnet $S$. Intuitively, a failure describes an environment that allows $S$ to reach a state in which all the messages offered by the environment are blocked. A failure, therefore, consists of a pair $(t, L)$ meaning that $S$ can follow the trace $t$ and end up in state $\mu$ such that none of the methods in the set $L$ would be enabled in $\mu$.

**Definition 31** A failure of a synchnet $S$ with initial state $I$ is a pair $(t, L)$ where $t$ is a trace of $S$ starting with the marking $I$ and ending at state $\mu$, and $L$ is a set of method tags such that for all $o.M \in L$, $o.M$ is not enabled in $\mu$.

The failures of a synchnets $S$ with initial marking $I$ is written as $\text{failures}(S)$ and is the set of all failures of $S$ starting at $I$. □

We are now ready to define the preorder relation on synchnets.

**Definition 32** For two synchnets $S$ and $S'$ that are instantiated with the same set of objects, we say $S \preceq S'$ whenever $\text{failures}(S) \subseteq \text{failures}(S')$. We also write $S \equiv S'$ whenever $S' \preceq S$ and $S \preceq S'$. □

It is not difficult to see that substituting $S'$ with $S$ when $S \preceq S'$ would not cause further deadlock situations than $S$ would. Therefore, if synchnets are always substituted according to this preorder, a non-deadlocking synchnet would never be substituted with a one that deadlocks.

### 6.6 Discussion and Related Work

The design of SynchNet is motivated by considering inheritance anomaly, distributed coordination, and two-level architectures. We discuss these topics to provide a better picture of where SynchNet
stands among similar or related research. We will also discuss the expressive power of SynchNet and a prototypical implementation along with challenges towards a truly distributed implementation.

6.6.1 SynchNet and Inheritance Anomaly

A distinguishing feature of object-oriented languages is a powerful mechanism for code “reuse” called inheritance. To illustrate how inheritance helps with code reuse, consider a bounded buffer as a class Buffer with methods get and put, and suppose we need to define a new class Buffer2 with methods get and put, the behavior of which is identical to those of the class Buffer, and an additional method get2, which retrieves two values from the buffer instead of one. The code for Buffer2 can be written as follows (in a Java like language):

class Buffer2 inherits from Buffer {
    Data[] get2() {
        // body of get2()
    }
}

The programmer does not need to rewrite the code for put and get methods for Buffer2 because they are copied verbatim from the code for Buffer by the “inheritance” mechanism.

In the late 1980s, most attempts at designing languages that featured both object-oriented and concurrency mechanisms discovered a peculiar challenge, that inheritance does not mix properly with synchronization. In other words, one could not inherit synchronization constraints and/or code as cleanly as one would inherit procedural specification (bodies) of methods. In many cases, inheriting from a superclass requires rewriting most of the synchronization code (sometimes even the body) of inherited methods. The term inheritance anomaly was coined to refer to such irritating situations, and the topic received wide attention in early 1990s. A thorough and careful analysis of inheritance anomaly was published in [37], in which the authors presented three large categories in which the anomaly manifests itself. They also analyzed various language design in terms of how effectively they deal with the problem.

In the rest of this section, we illustrate these three classes of inheritance anomaly with the help of simple classical examples. We will also discuss briefly some of the language designs that
were proposed to deal with these anomalies, but we will limit ourselves to languages that are in some ways similar to SynchNet. For the examples, we will use the Buffer class and assume it is being used in a concurrent setting. When several threads of control need to concurrently access a bounded buffer we expect the buffer to block invocations of get when the buffer is empty and block invocations of put when the buffer is full. We can implement Buffer in a Java like language as follows:

```java
class Buffer {
    Data[] buf;
    // Constructors ...
    void put(Data d) {
        while (empty()) { wait(); }
        // add d to buf
        notifyAll();
    }

    Data get() {
        while (full()) { wait(); }
        // read d from buf and return it
        notifyAll();
    }
}
```

**Partitioning of State**

This situation arises when we need to add new methods, whose synchronization constraint depends on a finer distinction of buffer states as opposed to having just empty and full. This happens, for example, if we add a new method get2 that must block unless at least two elements are present in the buffer.

SynchNet does not have any problem with partitioning of state, as the constraints act as guards. In general, all synchronization solutions that are based on guards deal effectively with the state
partitioning situation. For language designs that suffer from this situation, refer to [37].

**History-sensitivity of acceptable states**

This situation arises when the synchronization constraint depends not only on the current state of the object, but also on the history of previous invocations of that object’s methods. As an example, suppose we need to add a new method `gget` to the `Buffer` example. `gget` has the same behavior of `get` but must block the calling thread if the previous successful invocation on `Buffer` was not a `get`.

Dealing with history-sensitivity in most conventional concurrent object-oriented languages involve writing a lot of code to keep track of the history of invocation. Most of the time, this involves adding code to almost every method in the superclass. Two solutions that require minimal method redefinition are offered in [40] and [19]. The solution provided in [40] benefits from the flexibilities of its underlying platform, Maude. Maude is a rewriting system that provides extensive support for implementing new languages, logics, and verification mechanisms.

Another approach is to provide reflective mechanisms to assist the programmer in writing history-dependent synchronization code without any need for method redefinition. An example of such a language is Jeeg [41]. Jeeg is a guard based solution, which also allows the synchronization constraints to be specified separately from the functional part of class methods. The language used to specify synchronization constraints is Past Linear Time Temporal Logic. In order to refer to past invocations, a propositional variable `event` is included in the language that refers to the method being invoked at every point in time.

SynchNet also deals very effectively with history-sensitivity. In fact, the only things that a synchronization guard in SynchNet can refer to are tokens that correspond to previous method invocations. Therefore, keeping track of the past method invocations is implicit in the language design and there is no need for method redefinition. An advantage of SynchNet over Jeeg is the ability to count the number of previous invocations. For instance, one can express whether a certain method was invoked at least three times in the past. Counting is impossible in Jeeg because of inherent limitations in Linear Time Temporal Logic. However, unlike Jeeg, SynchNet is not capable of specifying a certain sequence of method invocations in the past. This limitation can be improved
by allowing shadow places in the specification of a SynchNet. These places do not correspond to any method but serve to provide more fine grained partitioning of the SynchNet’s state.

**Modification of Acceptable States**

This kind of anomaly happens when the code for two classes need to be mixed. In absence of synchronization, such combination is possible via multiple inheritance, a mechanism that is available in languages Smalltalk and C++. With synchronization, however, things are a bit more complicated. As an example suppose we want to mix the class *Buffer* with the following class:

```java
class Lock {
    ...
    void lock() { ... }
    void unlock() { ... }
}
```

The intention is that after the method *lock* is invoked in the mixed class, all other methods are blocked until *unlock* is invoked. This normally requires rewriting the synchronization code for every method of *Buffer*. In [40], an ingenious solution is presented. The solution relies on a mechanism to tag and untag the class name of the *Buffer* class so that all synchronization guards automatically become inapplicable temporarily.

Unfortunately, the solution provided in [40] does not scale very naturally. For instance, suppose instead of the class *Lock* we intend to mix *Buffer* with a security class *Secure*. Suppose further that *Secure* provides four protection levels and several methods to switch among levels based on a complex logic that is encapsulated in *Secure*. Now if we want to modify *Buffer*’s synchronization constraints so that different methods are available in different security levels, we run into much difficulty. SynchNet fares slightly better in the sense that mixing with constraints similar to *Secure* is possible. However, SynchNet still does not provide a natural and general solution to this type of inheritance anomaly either. In fact, to the best of the author’s knowledge, none of the solutions for this kind of inheritance anomaly truly solve the problem (in the sense of eliminating the need for rewriting and redefinition of method’s codes) in all cases.
In short, even though SynchNet deals with certain kinds of inheritance anomaly relatively effectively, it still does not address all the possible types of the anomaly. This is not very surprising, as after more than twenty years since its discovery, and the abundance of solutions provided for the problem, research on inheritance anomaly is still alive [41].

6.6.2 Centralization of Distributed Coordination Policies

SynchNet does more than coordinating a single object’s interaction with its environment: one can also specify the coordination requirements among a group of concurrent or distributed objects. It should be noted, however, that SynchNet is not a distributed programming language and is not intended to write distributed algorithms. Its main purpose, in fact, is to specify how the activities of a group of distributed objects are coordinated, assuming that partial access to the local states of the object of the collective is available. Hence, SynchNet is a language for centrally specifying distributed coordination policies.

In this respect, SynchNet is very much like Synchronizers [19], from which it is inspired for the most part. There are, however, two main differences. Synchronizers contain high level language constructs. This enables them to express Turing computable functions. In contrast, SynchNet is in fact no more expressive than its underlying model, Petri Nets. The expected advantage is that with less expressive power, one can benefit from decidability of many property verification problems such as reachability and coverability.

The second difference lies in how guards are extended by the inheritance mechanisms of the two languages. In Synchronizers, when a synchronizer, let’s say B, extends the specification of another synchronizer, let’s say A, it can only add more constraints to the guards specified in A. The intuition is that by restricting the guards, fewer messages would be delivered to an object constrained by B. As a result, one can say that it is “safe” to replace A with B, where “safety” is interpreted as non-delivery of messages to an object that is not ready to receive the messages.

Even though increasing constraints on method guards provides a certain degree of safety for the object begin synchronized, there is a catch when a group of objects are being coordinated. By restricting delivery, we might increase the chance of creating a deadlock situation among a group of objects. This observation has led to the point of view expressed in [50] that requires a subtype
process admit more messages than its supertype.

SynchNet is more liberal in its inheritance mechanism: it is possible to both restrict and relax method guards. The reason is that the inheritance only allows access to the “super”-SynchNet’s guard. Forming new guard is possible by using any mix of conjunction or disjunction connectives that are provided for this purpose.

One might argue that SynchNet inheritance does not satisfy the “substitutability” principle as advocated by the “behavioral type” research [35, 45]. We argue that this is not a problem. The notion of substitutability as proposed in behavioral type literature, is a semantic one: Substitution of a component \( A \) of type \( t \) by a component \( B \) of type \( s \) should not change the “behavior” of the system if \( s \) is a subtype of \( t \).

However, the conventional notion of subtyping does not impose such a ‘heavy’ semantic requirements: Substitution of a component \( A \) of type \( t \) by a component \( B \) of type \( s \) should not violate the “syntactic” compositional structure of the system if \( s \) is a subtype of \( t \). This is the familiar notion of substitutability enforced by subtyping relation of typed object-oriented languages. An object of a subtype may have a completely unrelated behavior to an object of its supertype, but it must have the same syntactic interface. SynchNet follows this tradition of typing. Inheritance in SynchNet yields indeed a structural subtyping relation, although no behavior preservation is guaranteed by this subtyping relation.

Regarding the role of SynchNet as specification of distributed coordination policies, it is (probably) useful to consider its relationship with the coordination language/system Linda [22, 10]. Linda is a collection of language primitives providing a model of coordination based on a distributed data structure called tuple space. With primitives \texttt{out}, \texttt{in}, and \texttt{read}, distributed processes may add or retrieve tuples of values to and from a shared tuple space. Synchronization is achieved by the blocking nature of \texttt{in} and \texttt{read} primitives that block the calling process if no matching tuple is available in the tuple space. These primitives do not constitute and independent language, but they can be added to virtually any sequential language to create a distributed programming language.

As we noted before, SynchNet may not be used directly for distributed programming. It is merely a meta-level specification language that describes, in a high-level and centralized fashion, how the behavior of a collection of objects must coordinated. In order to run SynchNet specifica-
tions, one must translate the specification into distributed programming primitives of the underlying object-oriented language. These primitives could be shared memory base, or message passing based. They could also be more high-level mechanisms such as RPC or RMI, or these primitives could be those of Linda. In other words, Linda can be used as an ‘assembly language’ for implementation of SynchNets.

6.6.3 Aspect Oriented and Two-Level Programming

The key motivation behind SynchNet is the software engineering principle of “separation of concerns.” Separation of concerns requires that the basic functionality of a system be specified separately from special purpose concerns such as synchronization, real-time constraint, performance evaluation, and debugging.

Aspect Oriented Programming (AOP), which emerged during the past decade, is a novel programming paradigm that focuses on the principle of separation of concerns. The basis of AOP is to separate the specification of non-functional aspects of a programming system from the functional aspects. The aspect-oriented language implementation would generate code that mixes the functional part of the code with the specified aspects. Various types of aspects have received attention since the inception of AOP. Among these are logging, fault-tolerance, access control, authentication, and concurrency control.

Conceptually, SynchNet can be regarded as an AOP style extension to object-oriented languages whose job is to separate distributed coordination aspects from functional behavior of objects. Similar to AOP frameworks and languages, SynchNet automatically “weaves” coordination code into the source code that specifies object’s behavior. However, note that an implementation of SynchNet does more than AOP; in AOP, the so called “cross-cutting” aspects, the aspects that are specified separately and are woven into the functional part of the code, are specified verbatim. More precisely, the programmer must specify the exact point in the program’s control flow where the aspect’s code has to be inserted. SynchNet, however, needs a lot of processing on the coordination specification to figure out a distributed implementation of that specification and then to weave the generated code into the functional part of objects’ code. In other words, the SynchNet specification is much more abstract than a specification in a standard AOP framework.
Another approach to language design that is based on the principle of separation of concern is the so called \textit{reflective two-level languages}\cite{13, 54, 61}. Even though the ideas of reflective two-level (sometimes meta-level) designs are more commonly applied to middleware and software architectures, they can also be applied in the design of new programming language. Again, during the past decade, we have witnessed the emergence of many reflective two-level designs for the purpose of distributed coordination. SynchNet can be seen as two-level language, but it does not provide any reflective capabilities.

\subsection*{6.6.4 Expressive Power}

Examples given earlier in this chapter illustrate, to some extent, what can be expressed in SynchNet. In more precise terms, many conventional coordination mechanisms can be expressed in SynchNet: distributed mutual exclusion, critical sections, barrier synchronization, and multicast. Unfortunately, a precise characterization of the expressive power of SynchNet is not an easy task and requires further research.

However, it is useful to note things that SynchNet can definitely not express. For example, SynchNet can not express time-based coordination necessary for real-time distributed computing. Neither can SynchNet be used to program fault-tolerant systems. To achieve these requirements, SynchNet must be extended with timing and fault-detection primitives.

\subsection*{6.6.5 Implementation issues}

Currently, a prototype implementation of SynchNet is available for Java. The implementation weaves the specification into Java code. The operational semantics of SynchNet is achieved by assigning a central coordinating JVM that simulates the Petri Net specified by the SynchNet specification. The classes that are required to be coordinated with the SynchNet specification are augmented with code that control method invocations on corresponding objects of the class. The added code is responsible for communicating the pending method requests with the central coordinator and would wait for acknowledgment from the coordinator before it invokes any method.

However, the true benefit of SynchNet can be achieved only with a distributed implementation. Unfortunately, preliminary investigations show that naive distributed implementations do not offer
better performance than a centralized solution (if not worse). Even Synchronizer’s implementation [20] fails to provide a distributed solution. In order to have an efficient distributed implementation, one must be able to partition the SynchNet’s places into groups that would exhibit maximum coupling with the objects they control and little coupling with objects located on a remote node. This is a complex problem and an interesting topic for future research.

6.7 Summary and Future Work

We propose the use of Petri Nets as a simple meta-level language to specify coordination requirements of a group of distributed objects. When this meta-level language (SynchNet) is combined with an object-based language with asynchronous remote method invocation semantics, we obtain an expressive distributed object-based language. To keep things simple, our coordination language refers only to the labels of methods. As a result, coordination requirements that discriminate between messages containing distinct values cannot be expressed in our currently proposed language. We have observed that, despite this limitation, our language is still expressive enough to represent many interesting coordination patterns.

Since synchnets are in fact Petri nets, we can benefit from the rich and well studied theory of Petri Nets. The theory includes formal characterizations of many interesting properties along with decision algorithms to decide those properties. Automatic analysis tools have made these theories accessible to practitioners.

Our compiler for SynchNet automatically generates distributed code. The current implementation uses a naive distributed shared memory protocol and therefore suffers from low performance. Currently, we are working on a new algorithm that, by exploiting the structure of synchnets, would hopefully generate distributed code with efficiency comparable to the best distributed solutions available.

Considering that most modern distributed systems operate in open and dynamic environments and that coordination requirements usually evolve throughout the lifetime of a system, it is generally desirable to have development systems that allow dynamic customization of coordination aspects. Our proposed two-level model provides some support for dynamically evolving systems: It is possible to dynamically create new objects (and threads that execute their scripts), and instan-
tiate new synchnets to coordinate the newly created objects. More flexibility would be achieved if one could replace or modify a running synchnet on the fly. Even though our current model does not support this level of dynamic customizability, its simple and formal semantics should simplify the study of such issues.
References


Vita

Mahmood Reza Ziaei was born in Tehran, Iran, on April 16, 1970. He graduated from Shahid Beheshti University, Tehran, in 1993 with a B.S. degree in computer science, and from Sharif University of Technology, Tehran, in 1996 with a M.S. degree in computer science. He started the Ph.D. program at the University of Illinois in Urbana Champaign in January 1997. In 2000, Ziaei worked as a summer intern for Motorola’s Urbana Design Center. He completed his PhD in October 2004 and started working at Motorola as a Senior Software Engineer.