A THEORY OF TESTING FOR ASYNCHRONOUS CONCURRENT SYSTEMS

BY

PRASANNAA THATI

B.Tech., Indian Institute of Technology at Kanpur, 1997
M.S., University of Illinois at Urbana-Champaign, 2000
M.S., University of Illinois at Urbana-Champaign, 2002

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, 2003

Urbana, Illinois

ABSTRACT

Testing equivalence is a notion of process equivalence that is widely used for establishing semantic correspondence between concurrent systems. Testing equivalence can be used to formally establish the fact that a given system is a safe or live implementation of an abstract specification. However, since the definition of testing equivalence involves a universal quantification over all possible contexts, proving equivalences between processes is a difficult task. In this dissertation, we present a collection of proof techniques and decision procedures for establishing testing equivalences between asynchronous concurrent systems.

The process model that will be our primary focus is the π-calculus, which has been one of the most popular models of concurrency since its introduction a decade-and-a-half ago. We study a collection of typed variants of π-calculus that enforce several ontological commitments in addition to those in the basic π-calculus. These commitments capture ubiquitous computational phenomena such as asynchrony, locality, object paradigm, and restrictions on pointer comparisons. We investigate testing equivalence on several variants of π-calculus with combinations of these features. The result is a collection of proof techniques that can be applied to reason about a rich class of systems that exhibit these computational features.

The central idea behind our proof techniques for testing equivalence is to obtain semantic characterizations of the equivalence that do away with universal quantification over contexts. Using these characterizations one can thus establish an equivalence by simply comparing the semantic mappings of the given processes in an abstract domain instead of accounting for their interactions with all possible contexts of use. We use these semantic characterizations to obtain both complete axiomatizations of the equivalence and decision procedures for it over restricted classes of processes. We have also implemented some of the variants of π-calculus and the proof techniques we have developed for establishing equivalences over them.

To nanna, amma, and cheens

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Gul Agha for his intellectual guidance and constant encouragement, and for fostering a research environment with great intellectual freedom. He has been an excellent role model and has had a profound impact on not only my outlook towards research but also towards life in general.

I would also like to thank my committee members, Prof. Jose Meseguer, Prof. Mahesh Viswanathan, and Dr. Carolyn Talcott for their invaluable guidance and feedback on my thesis research. Major portions of this thesis were motivated by frequent discussions with them, and some were completely worked out in collaboration with them. I thank Dr. Carolyn Talcott for making it possible for me to visit her at Stanford University and SRI International for a few months in 2000 and 2002 respectively. Many of the results in Chapters 3 and 4 were originally worked out during these visits. I thank Prof. Jose Meseguer for encouraging me to work on the problems described in Chapter 6. Chapter 5 was entirely worked out in collaboration with Prof. Mahesh Viswanathan; I thank him for his interest and technical input in my research.

Special thanks to my long time friend Reza Ziaei. The origin of my thesis research can be traced back to initial collaborations with him on algebraic formulation of the Actor model (Chapter 3). The Open Systems Lab members have provided a very stimulating intellectual environment throughout my graduate life. The countless discussions I have had with them have been a source of great fun and inspiration. I would like to especially thank Mark Astley, Po-Hao Chang, Nadeem Jamali, Nirman Kumar, Youngmin Kwon, Soham Mazumdar, Koushik Sen, Sudarshan Srinivasan, Predrag Tosic, Sandeep Uttamchandani, Abhay Vardhan, Carlos Varela, Nalini Venkatasubramanian, and Reza Ziaei.

Last but not the least, I am very grateful to my parents and brother who have always given me unconditional love and emotional support. No words can express my gratitude for their love and confidence in me.

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

We present a theory of testing equivalence for asynchronous concurrent systems that are open to interactions with their environment. The problem we address is an important instance of a range of problems concerned with verification of concurrent systems.

There are two major approaches to system verification: *model checking* and *equivalence checking*. In the model checking approach, one specifies a property as a formula in a modal or temporal logic [74], and then uses a decision procedure to check whether a given system is a model for the formula, i.e. whether the given system satisfies the property specified by the formula. This approach was introduced two decades ago by Clarke, Emerson, and Sistla [21], and has since received wide attention from both researchers and practitioners [22] of system verification.

Equivalence checking on the other hand, aims at establishing some semantic correspondence between two systems, one of which is typically considered to be an abstract specification and the other a more concrete implementation of the specification. The problem of equivalence checking is an old question in theoretical computer science with a large body of literature. Ever since the seminal paper by Moore [66] on the language equivalence of finite state machines, a number of other researchers have considered the equivalence problem over a number of computational models of varying expressivity [17, 20, 34, 50, 60, 61, 68].

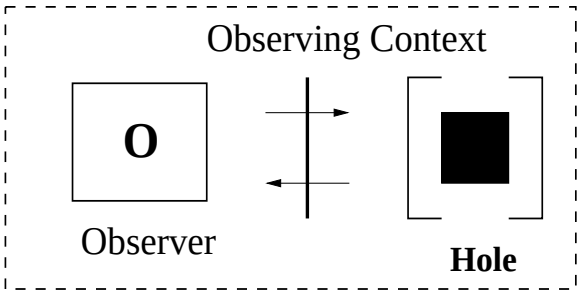


Figure 1.1: A context in testing equivalence consists of an observing process that executes concurrently and interacts with the process being tested.

The problem of testing equivalence belongs to the category of equivalence checking; testing equivalence is one way of formalizing the semantic correspondence between two systems. In fact, testing equivalence is an instance of the general notion of behavioral equivalence where, roughly, two processes are said to be behaviorally equivalent if they are indistinguishable in all contexts of use. Depending on the chosen notion of context and the criteria for indistinguishability, one gets a variety of equivalences [27, 56, 68, 79]. Testing equivalence [36, 68] is one such instance where the context consists of an observing process that runs in parallel and interacts with the process being tested (see Figure 1.1). The observer can in addition signal a success while interacting with the process being tested, in which case the process is said to pass the test proposed by the observer. Two process are said to be indistinguishable if they pass exactly the same set of tests (see Figure 1.2).

The testing scenario is inherently concurrent; the observer and the process being tested execute concurrently. Such concurrent systems are inherently non-deterministic; there is no single order in which different computation steps in the system can occur, and different orders lead the system along different computation paths. This non-determinism in execution gives rise to at least two notions of equivalences. In *may* testing, a process is said to pass a test proposed by an observer, if there *exists* a run in which the observer signals a success. On the other hand, in *must* testing a process is said to pass a test if the observer signals a success in *every* possible run (see Figure 1.3).

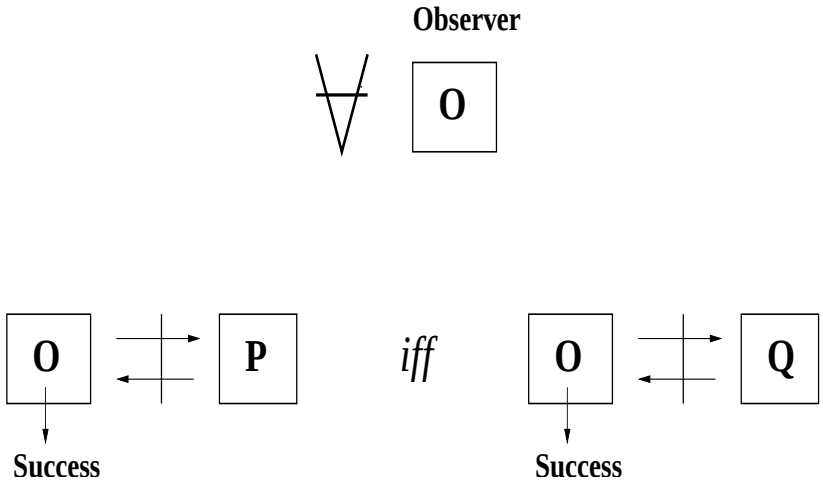


Figure 1.2: Processes P and Q are said to be testing equivalent if they lead exactly the same set of observers to a success.

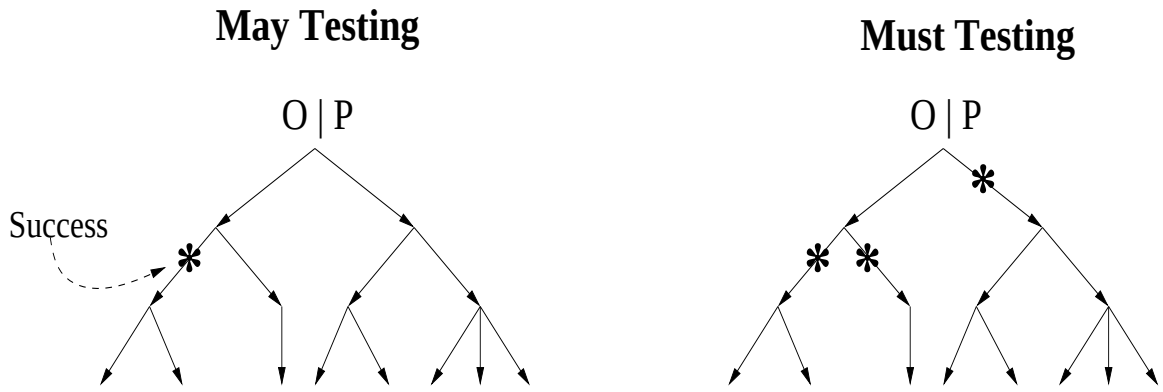


Figure 1.3: In may testing even the possibility of a success is taken to be a success, while in must testing only the guarantee of a success is taken to be a success.

May and must testing are known to be useful for reasoning about safety and liveness properties of concurrent systems. Specifically, define the may preroder $P \stackrel{\sqsubset}{\sim} Q$ if for every observer O whenever P may lead O to a success then so can Q, and define the must preorder $P \stackrel{\sqsubset}{\sim}^{M} Q$ if for every O whenever P always leads O to success then so does Q. Then, by viewing the observer's success as something bad happening, may testing can be used for reasoning about *safety* properties; $P \stackrel{\sqsubset}{\sim} Q$ can be interpreted as P is a safe implementation of the specification Q, because if the specification Q is guaranteed to not cause anything bad to happen in a given context, then the implementation P would also not cause anything bad to happen in the same context. Similarly , by viewing a success as something good happening, must testing can be used for reasoning about *liveness* properties; $P \stackrel{\sqsubset}{\sim}^{M} Q$ can be interpreted as Q is a live implementation of the specification P, because if the specification P is guaranteed to always cause something good to happen in a given context, then the implementation Q would also always cause something good to happen in the same context.

Although testing equivalences have several elegant applications [1], they pose serious difficulties in verification. The universal quantification over contexts in the definition of the equivalences makes it very hard to prove equalities. Specifically, to prove an equivalence, one has to consider all possible interactions between the given processes and all possible observers. As a solution, we develop a theory of testing equivalence that provides a collection of proof techniques and automatic verification algorithms for establishing equivalences over a large class of concurrent systems.

In our discussion so far, we have not committed to any particular process model. Clearly, testing equivalence is a generic notion that can be instantiated over a variety of process models. The model that will be our primary focus from now on, is the π-calculus which was proposed by Milner, Parrow and Walker [65]. Since its introduction a decade-and-a-half ago the π-calculus has become one of the most studied models of concurrency. Just as the λ-calculus assumes only the notions of function abstraction and application as the essential aspects of deterministic sequential computing, the π-calculus assumes only *naming* and *communication* at names as the primitive notions to

describe concurrency. Names in the π-calculus denote communication links, and can by themselves be communicated across the links. One can thus express a collection of interacting process with a continually changing interconnection topology.

The primitive notions of names and communication of names are sufficient to model a variety of concurrent systems. In fact, the pervasiveness of these ontological commitments makes the π-calculus a highly expressive formalism that can model a number of systems with minimal representational distance [1, 72, 78, 81]. Further, the π-calculus facilitates modelling of systems at various levels of abstraction; one can abstractly specify the observable behavior of a system and then refine it down to a more concrete system describing the internal implementation that realizes the specification. The theory of process equivalence which is useful for relating such multiple descriptions in π-calculus, has been an intense of area of research over the last decade [9, 10, 26, 65, 69, 71, 79].

In practice the π-calculus is often too general, and it is useful to restrict the basic calculus to fragments that are tailored to specific problems at hand. For instance, it is useful to impose type systems on the basic calculus, that enforce additional ontological commitments such as the object-paradigm. These commitments can typically be expressed by the type systems as certain disciplines in the use of names. The resulting variants of π-calculus have a richer theory of process equivalence, and a powerful collection of proof techniques that are more effective in practice. This trend is apparent in the large number of variants of π-calculus proposed in the recent years [16, 41, 47, 58, 71, 82, 83].

We focus our research on variants of π-calculus that embody the following additional features.

- **Asynchrony**: Communication in the π-calculus is synchronous, it involves a (directed) handshake between the sender and the receiver. However, asynchronous message passing is closer to what is offered by most distributed systems, where the sender has no knowledge of the receipt of a message, other than by an explicit acknowledgment.

- **Locality**: This is a discipline where the recipient of a name communicated in a message, can use the name to only send messages. Specifically, the recipient cannot start listening at the name and receive messages targeted to the name. This phenomenon is common in real systems. For instance, in a client-server model, a client that knows the access name to the server cannot steal messages targeted to the server. However, this discipline is absent in the π-calculus.

- **Restricted Name matching**: Comparing names is rarely useful in programming. The behavior observed while communicating at a name is all that matters; the specific name used for communication is not important. Accordingly, it is useful to restrict the use of names to only communication, and disallow the association of additional meaning to names through matching. This discipline is enforced in concurrent languages such as Join [30] and Pict [72]. The advantages include, additional program transformations that would otherwise be unsound. We are also interested in scenarios where this restriction can be relaxed a little to allow comparison on specific names, so that the advantages of completely disallowing name matching do not disappear all together.

- **An object paradigm**: In an object world, names uniquely denote persistent computational objects rather than communication channels. This can be seen as a natural extension to the locality discipline mentioned above. There are several object paradigms that embody this general view. The specific paradigm of our interest is the Actor Model [5], that has had a profound influence on the design of concurrent languages [6, 52, 99] and open distributed systems [18, 31, 87, 95] over the last three decades. Both asynchrony and locality are inherent features of the Actor Model.

We will investigate testing equivalences on variants of π-calculus with combinations of these features. The result will be a theory of equivalence and a collection of proof techniques that can be applied to reason about a fairly rich class of systems that arise

in practice. We will primarily focus only on the may testing equivalence, although we present a few results on must testing. Following is the outline of this dissertation.

In Chapter 2, we consider asynchronous variants of π-calculus with various combinations of locality and restricted name matching. For each of these variants, we develop an alternate characterization of may testing that does not involve a universal quantification over all observers, thereby simplifying the process of establishing equivalences. Specifically, we construct semantic models in which processes can be interpreted so that, to decide an equivalence between two processes we only need to compare their semantic mappings rather than consider their interactions with all possible observers. Our characterizations build upon the one developed by Boreale, de Nicola, and Pugliese for an asynchronous variant of π-calculus [11]. We not only extend their characterization to variants with locality and restricted name matching, but also consider the more general problem of *parameterized* may testing where the parameter determines the set of observers used to decide the equivalence. The usual notion of (unparameterized) may testing is just a special case where the (implicit) parameter denotes the set of all possible observers. Finally, we exploit our characterizations of parameterized may testing in each of the variants we consider, to develop an axiomatization of may testing that is complete for the finitary fragment of the variants (without recursion).

In Chapter 3, we impose a type system on asynchronous π-calculus to obtain a variant called Aπ that is a faithful representation of the Actor Model. The type system captures the object paradigm proposed by the Actor model by enforcing certain discipline in the use of names. We also consider variants of Aπ that differ in their name matching capabilities. We extend the results in Chapter 2 to obtain an alternate characterization of may testing for Aπ and its variants. The formulation of Aπ also settles the following old problem quoted from Milner's Turing award lecture [62]: "*An important task is to compare π-calculus with Hewitt's Actors; there are definite points of agreement where we have followed the intuition of Actors and also some subtle differences, such as the treatment of names.*" A noteworthy feature of our embedding of actors in π-calculus is that it has almost no representational distance. This shows that the Actor Model and the π-calculus

only differ in their treatment of names, except for some fairness conditions on message deliveries in the Actor model. The formal connection thus established between the two models, can be exploited to apply concepts and techniques developed for one to the other. Our alternate characterization of may testing for actors, is one such demonstration.

The results in Chapters 2 and 3 can be exploited to reason about concurrent programming languages with computational features such as asynchrony, restricted name matching, locality and the object paradigm. This can be done by either adapting our results to these languages, or by encoding the languages into one of our calculi. As an illustration, in Chapter 4, we give a formal semantics to a simple asynchronous concurrent object-based language inspired by the Actor model, by translating its programs into Aπ. The translation also has the property that two programs are may equivalent if and only if their translations are, thereby giving us a technique to reason about programs in the language.

In Chapter 5, we investigate the decidability of may and must testing equivalences over a fragment of asynchronous π-calculus that can express a rich set of infinite state systems. We present an algorithm for unparameterized may equivalence over this fragment, and show that the problem is EXPSPACE-hard. In comparison, algorithms for may testing were previously known for only the simple class of finite state machines [45]. We also show that the parameterized may and must testing equivalences are undecidable even over the fragment we consider. The decidability of unparameterized must testing over the fragment is still open.

In Chapter 6, we present our implementation of asynchronous π-calculus, the Actor model, and a procedure for may testing over finitary process in these models. Specifically, we represent asynchronous π-calculus and the Actor model as theories in rewriting logic [59], and use the Maude tool [23] that supports specifications in rewriting logic, to execute these specifications. We then exploit our alternate characterizations of may testing and some metalevel facilities in Maude to implement a procedure to decide parameterized may equivalence between finitary process in these models. Our implementation techniques

can be adapted to all the other variants of π-calculus and the Actor model that we have considered.

We conclude in Chapter 7, with a discussion on the various directions along which the work so far can be extended.

Chapter 2

Asynchronous π-Calculus with Locality and Restricted Name Matching

We present a series of asynchronous variants of π-calculus that progressively account for locality and restricted name matching, and provide alternate characterizations of may testing in them [94]. We consider a generalized version of may testing that is parameterized with the set of observers used for testing. Our characterizations directly build upon the known results for asynchronous π-calculus [11], which are summarized in Section 2.1. The reader is referred to [11] for the proofs of propositions in this section. In Section 2.2, we present the characterization for an asynchronous variant with locality, called L$\pi_{=}$. In Section 2.3, we consider Lπ which is an asynchronous variant with locality and no name matching. In Section 2.4, we consider an asynchronous variant with locality and a restricted version of name matching, where a process can compare a given name only against the names that it owns. In Section 2.5 we present an axiomatization of parameterized may testing over L$\pi_{=}$ and Lπ, that is complete for the finitary fragment of these calculi (without recursion). We end the chapter with Section 2.6, where we discuss related work. To simplify the presentation, we have moved the proofs of some of the propositions in this chapter to Appendix A.

2.1 Asynchronous π-calculus

An infinite set of names $\mathcal{N}$ is assumed, and $u, v, w, x, y, z, \ldots$ are assumed to range over $\mathcal{N}$. The set of processes, ranged over by P, Q, R, is defined by the following grammar.

$$P := \quad 0 \;\mid\; \overline{x}y \;\mid\; x(y).P \;\mid\; P|P \;\mid\; (\nu x)P \;\mid\; [x = y]P \;\mid\; !x(y).P$$

The nil process 0 has the trivial behavior that does nothing. The output term $\overline{x}y$ denotes an asynchronous message with target x and content y. The input term $x(y).P$ receives an arbitrary name z at port x and then behaves like $P\{z/y\}$ (substitution). The composition $P|Q$ consists of P and Q acting in parallel. The components can act independently, and also interact with each other. The restriction $(\nu x)P$ behaves like P except that it can not exchange messages targeted to x, with its environment. The match $[x = y]P$ behaves like P if x and y are identical, and like 0 otherwise. The replication $!x(y).P$ provides an infinite number of copies of $x(y).P$.

The name x is said to be the subject of the output $\overline{x}y$ and the input $x(y).P$. For a tuple $\tilde{x}$, the set of names occurring in $\tilde{x}$ is denoted by $\{\tilde{x}\}$. The result of appending $\tilde{y}$ to $\tilde{x}$ is denoted by $\tilde{x}, \tilde{y}$. The variable $\hat{z}$ is assumed to 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, $fn(.)$, $bn(.)$ and $n(.)$, of a process, are defined as expected. As in λ-calculus, alpha-equivalent processes, i.e. processes that differ only in the use of bound names are not distinguished. A name substitution is a function on names that is almost always the identity. The form $\{\tilde{y}/\tilde{x}\}$ denotes a substitution that maps x_i to y_i and is identity on all other names. The variable σ is assumed to range over substitutions. The result of simultaneous substitution of y_i for x_i in P is denoted by $P\{\tilde{y}/\tilde{x}\}$. As usual, substitution on processes is defined only modulo alpha equivalence, with the usual renaming of bound names to avoid captures.

A labeled transition system [64] is used to give an operational semantics for the calculus (Table 2.1). The transition system is defined modulo alpha-equivalence on processes in that alpha-equivalent processes have the same transitions. The rules *COM, CLOSE*,

$$\textit{INP:}\quad x(y).P \xrightarrow{xz} P\{z/y\} \qquad \textit{OUT:}\quad \overline{x}y \xrightarrow{\overline{x}y} 0$$

$$\textit{BINP:}\quad \frac{P \xrightarrow{xy} P'}{P \xrightarrow{x(y)} P'} \quad y \notin fn(P)$$

$$\textit{PAR:}\quad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset$$

$$\textit{COM:}\quad \frac{P_1 \xrightarrow{\overline{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$$

$$\textit{RES:}\quad \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \quad y \notin n(\alpha)$$

$$\textit{OPEN:}\quad \frac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(y)} P'} \quad x \neq y$$

$$\textit{CLOSE:}\quad \frac{P_1 \xrightarrow{\overline{x}(y)} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} (\nu y)(P_1'|P_2')} \quad y \notin fn(P_2)$$

$$\textit{REP:}\quad \frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \qquad \textit{MATCH:}\quad \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$$

Table 2.1: An early style labeled transition system for L$\pi_{=}$.

and *PAR* have symmetric versions that are not shown. Transition labels, which are also called actions, can be of five forms: τ (a silent action), $\overline{x}y$ (free output of a message with target x and content y), $\overline{x}(y)$ (bound output), xy (free input of a message) and $x(y)$ (bound input). The relation $\xrightarrow{x(y)}$ is defined by the additional rule $P \xrightarrow{x(y)} Q$ if $P \xrightarrow{xy} Q$ and $y \notin fn(P)$. The set of all visible actions (non-τ) actions is denoted by $\mathcal{L}$, and α is assumed to range over $\mathcal{L}$, and β over all the actions. The functions $fn(.)$, $bn(.)$ and $n(.)$ are defined on $\mathcal{L}$ as expected. As a uniform notation for free and bound actions the following notational convention is adopted: $(\emptyset)\overline{x}y = \overline{x}y$, $(\{y\})\overline{x}y = \overline{x}(y)$, and similarly for input actions. The complementation function on $\mathcal{L}$ is defined as $\overline{(\hat{y})xy} = (\hat{y})\overline{x}y$, $\overline{(\hat{y})\overline{x}y} = (\hat{y})xy$.

The structural congruence relation $\equiv$ on processes is the smallest congruence that is closed under the following rules.

1. $P|0 \equiv P$, $P|Q \equiv Q|P$, and $P|(Q|R) \equiv (P|Q)|R$
2. $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$, and $P|(\nu x)Q \equiv (\nu x)(P|Q)$ if $x \notin fn(P)$.
3. $!P \equiv P|!P$.
4. $[x = x]P \equiv P$.

It is the case that structurally congruent processes have the same transitions, i.e. if $P \xrightarrow{\beta} Q$, $P \equiv P'$ and $Q \equiv Q'$ then $P' \xrightarrow{\beta} Q'$.

The variables s, r, t are assumed to range over $\mathcal{L}^*$. The functions $fn(.)$, $bn(.)$ and $n(.)$ are extended to $\mathcal{L}^*$ the obvious way. Complementation on $\mathcal{L}$ is extended to $\mathcal{L}^*$ the obvious way. Alpha equivalence over traces is defined as expected, and alpha-equivalent traces are not distinguished. From now on, only *normal* traces $s \in \mathcal{L}^*$ that satisfy the following hygiene condition are considered: if $s = s_1.\alpha.s_2$, then $(n(s_1) \cup fn(\alpha)) \cap bn(\alpha.s_2) = \emptyset$.

The relation $\Longrightarrow$ denotes the reflexive transitive closure of $\xrightarrow{\tau}$, and $\stackrel{\beta}{\Longrightarrow}$ denotes $\Longrightarrow\xrightarrow{\beta}\Longrightarrow$. For $s = l.s'$, $C_1 \xrightarrow{l}\xrightarrow{s'} C_2$ is compactly written as $C_1 \xrightarrow{s} C_2$, and similarly $C_1 \stackrel{l}{\Longrightarrow}\stackrel{s'}{\Longrightarrow} C_2$ as $C_1 \stackrel{s}{\Longrightarrow} C_2$. The assertion, $C \stackrel{s}{\Longrightarrow} C'$ for some C', is written as $C \stackrel{s}{\Longrightarrow}$, and similarly $C \xrightarrow{s}$ and $C \xrightarrow{\tau}$.

The may testing framework [36] is adapted to asynchronous π-calculus in a straightforward manner.

Definition 2.1 (may testing) *Observers are processes that can emit a special message $\overline{\mu}\mu$; it is assumed that the name μ does not occur free in the processes being tested. We let O range over the set of observers. For P, O, we say P may O if $P|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. We say $P \stackrel{\sqsubset}{\sim} Q$ if for every O, P may O implies Q may O. We say $P \simeq Q$ if $P \stackrel{\sqsubset}{\sim} Q$ and $Q \stackrel{\sqsubset}{\sim} P$. Note that $\stackrel{\sqsubset}{\sim}$ is reflexive and transitive, and $\simeq$ is an equivalence relation.* □

In synchronous models such as CCS and π-calculus, trace equivalence is known to coincide with may-testing [36]. But this coincidence fails to hold in an asynchronous setting due to the non-blocking nature of output primitives. For instance, due to non-determinism in the arrival order of messages, the equation $x(u).y(v).P = y(v).x(u).P$, which is not true for trace equivalence, is true for may-testing. As another example, since the receipt of a message by a receiver is not observable to the sender, the equation $0 = x(u).\overline{x}u$ holds for may-testing, but not for trace equivalence.

To account for asynchrony, the definition of trace semantics is modified as follows. The preorder $\preceq$ is defined on the set of traces as the reflexive transitive closure of the

(L1)	$s_1.(\hat{y})s_2$	$\prec$	$s_1.(\hat{y})xy.s_2$	if $(\hat{y})s_2 \neq \perp$
(L2)	$s_1.(\hat{y})(\alpha.xy.s_2)$	$\prec$	$s_1.(\hat{y})xy.\alpha.s_2$	if $(\hat{y})(\alpha.xy.s_2) \neq \perp$
(L3)	$s_1.(\hat{y})s_2$	$\prec$	$s_1.(\hat{y})xy.\overline{x}y.s_2$	if $(\hat{y})s_2 \neq \perp$
(L4)	$s_1.\overline{x}w.(s_2\{w/y\})$	$\prec$	$s_1.\overline{x}(y).s_2$	

Table 2.2: A preorder relation on traces.

laws shown in Table 2.2, where the notation $(\hat{y})\cdot$ is extended to traces as follows.

$$(\hat{y})s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } b \notin \mathit{fn}(s) \\ s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\ & s = s_1.xy.s_2 \text{ and } y \notin \mathit{fn}(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

The intuition behind the preorder is that an environment that accepts a trace s can also accept any trace $r \preceq s$. The characterization of may preorder $\stackrel{\sqsubset}{\sim}$ then becomes $P \stackrel{\sqsubset}{\sim} Q$ if and only if for every trace s that P exhibits, Q can exhibit a trace r such that $r \preceq s$. While laws *L1-L3* capture asynchrony, law *L4* captures the inability to mismatch names and disappears in the presence of mismatch operator. 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 2.1 *If $P \stackrel{\overline{s}}{\Longrightarrow}$, then $r \preceq s$ implies $P \stackrel{\overline{r}}{\Longrightarrow}$.* □

The relation $\ll$ defined below is shown to be an alternate characterization of $\stackrel{\sqsubset}{\sim}$.

Definition 2.2 *We say $P \ll Q$ if for every trace s, $P \stackrel{s}{\Longrightarrow}$ implies there is $r \preceq s$ such that $Q \stackrel{r}{\Longrightarrow}$.* □

To prove the characterization, a special class of observers $O(s)$ is constructed such that P *may* $O(s)$ implies $P \stackrel{r}{\Longrightarrow}$ for some $r \preceq s$.

Definition 2.3 (canonical observer) *For a trace s, we define $O(s)$ as follows:*

$$\begin{array}{rcll} O(\epsilon) & \stackrel{def}{=} & \overline{\mu}\mu & \\ O((\hat{y})xy.t) & \stackrel{def}{=} & (\nu\hat{y})(\overline{x}y|O(t)) & \\ O(\overline{x}y.s) & \stackrel{def}{=} & x(u).[u=y]O(s) & u \textit{ fresh} \\ O(\overline{x}(y).t) & \stackrel{def}{=} & x(y).O(t) & \end{array}$$

□

Lemma 2.2 *For a trace s, $O(s) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$ implies $r \preceq s$.* □

The following theorem establishes the relation between $\stackrel{\sqsubset}{\sim}$ and $\ll$.

Theorem 2.1 *$P \stackrel{\sqsubset}{\sim} Q$ if and only if $P \ll Q$.*

We end the section with a note on a variant of asynchronous π-calculus with both match and mismatch capabilities on names. This variant is additionally equipped with a conditional construct $[x=y](P,Q)$ whose semantics is defined by the following rules.

$$IF \quad \frac{P \stackrel{\alpha}{\longrightarrow} P'}{[x=y](P,Q) \stackrel{\alpha}{\longrightarrow} P'} \quad x=y \qquad\qquad ELSE \quad \frac{Q \stackrel{\alpha}{\longrightarrow} Q'}{[x=y](P,Q) \stackrel{\alpha}{\longrightarrow} Q'} \quad x \neq y$$

The relation $\ll$ defined as in Definition 2.2, but using only laws *L1, L2* and *L3*, characterizes the may preorder over this variant [11]. From now on, unless mentioned otherwise, by asynchronous π-calculus we refer to the calculus with only the match capability on names.

2.2 The Calculus L$\pi_=$

We now impose the discipline of locality on asynchronous π-calculus, that disallows processes from receiving a name and listening to it. We call the resulting calculus L$\pi_=$. Locality is enforced by requiring that the bound name y in an input $x(y).P$ does not occur as the subject of an input in P. The transition rules for L$\pi_=$ are the same as for asynchronous π-calculus (Table 2.1).

The locality property weakens may testing equivalence as it reduces the number of observers that can be used to test a process. For example, the following two processes are distinguishable in asynchronous π-calculus but equivalent in L$\pi_=$.

$$\begin{aligned} P_1 &= (\nu x)(!x(z).0|\overline{x}x|\overline{y}x) \\ P_2 &= (\nu x)(!x(z).0|\overline{y}x) \end{aligned}$$

The observer $O = y(z).z(w).\overline{\mu}\mu$ can distinguish P_1 and P_2 in asynchronous π-calculus, but is not a valid L$\pi_=$ term as it violates locality. In fact, no L$\pi_=$ term can distinguish P_1 and P_2, because the message $\overline{x}x$ is not observable.

Extending the notion of locality, we consider a generalized version of may testing that supports encapsulation. We define a parameterized may preorder $\stackrel{\sqsubset}{\sim}_\rho$, where only observers that do not listen on names in ρ are used to decide the order. For instance, $\overline{x}x \stackrel{\sqsubset}{\sim}_{\{x\}} 0$, but $\overline{x}x \stackrel{\sqsubset}{\not\sim}_\emptyset 0$. The set of names ρ can be interpreted as being owned by (or private to) 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.

Definition 2.4 (may testing) *Let $rcp(P)$ (receptionist names in P) be the set of all free names in P that occur as the subject of an input in P. For any given ρ we say $P \stackrel{\sqsubset}{\sim}_\rho Q$ if for every O such that $rcp(O) \cap \rho = \emptyset$, P may O implies Q may O.* □

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 $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$ implies $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$. However, $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$ need not imply $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$. For instance, $0 \simeq_{\{x\}} \overline{x}x$, but only $0 \stackrel{\sqsubset}{\sim}_\emptyset \overline{x}x$ and

$\overline{x}x \stackrel{\sqsubset}{\not\sim}_{\emptyset} 0$. Similarly, $\overline{x}x \simeq_{\{x,y\}} \overline{y}y$, but $\overline{x}x \stackrel{\sqsubset}{\not\sim}_{\emptyset} \overline{y}y$ and $\overline{y}y \stackrel{\sqsubset}{\not\sim}_{\emptyset} \overline{x}x$. However, $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$ implies $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$ if $fn(P) \cup fn(Q) \subset \rho_1$.

Theorem 2.2 *Let $\rho_1 \subset \rho_2$. Then $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$ implies $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$. Further, if $fn(P) \cup fn(Q) \subset \rho_1$ then $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$ implies $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$.*

Proof: Let $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$. Suppose $P \underline{may}\ O$ and $rcp(O) \cap \rho_2 = \emptyset$. Since $\rho_1 \subset \rho_2$, we have $rcp(O) \cap \rho_1 = \emptyset$. Then since $P \stackrel{\sqsubset}{\sim}_{\rho_1} Q$, we have $Q \underline{may}\ O$. Hence $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$.

Let $fn(P) \cup fn(Q) \subset \rho_1$ and $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$. Suppose $P \underline{may}\ O$ and $rcp(O) \cap \rho_1 = \emptyset$. We have to show $Q \underline{may}\ O$. Let $\{\tilde{x}\} = rcp(O)$. Then, we have $rcp((\nu\tilde{x})O) = \emptyset$. Since $fn(P) \cap \{\tilde{x}\} = \emptyset$, we have $P|(\nu\tilde{x})O \equiv (\nu\tilde{x})(P|O)$. From this, *RES* rule, and $P|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, we deduce $P|(\nu\tilde{x})O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, i.e. $P \underline{may}\ (\nu\tilde{x})O$. Now, since $P \stackrel{\sqsubset}{\sim}_{\rho_2} Q$ we have $Q \underline{may}\ (\nu\tilde{x})O$. From $fn(Q) \cap \{\tilde{x}\} = \emptyset$ we have $(\nu\tilde{x})(Q|O) \equiv Q|(\nu\tilde{x})O$. It follows that $(\nu\tilde{x})(Q|O) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, and hence $Q|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. □

We now give a trace based characterization of $\stackrel{\sqsubset}{\sim}_{\rho}$, by making appropriate modifications to the characterization for asynchronous π-calculus. As motivated by the example at the beginning of this section, we need to weaken the characterization to consider only traces that correspond to interaction between L$\pi_=$ processes. Note that the transition system for L$\pi_=$ does not by itself account for locality. For instance, in case of the example, we have $P_1 \stackrel{\overline{y}x}{\longrightarrow}\stackrel{\overline{x}x}{\longrightarrow}$ although the message $\overline{x}x$ is not observable. To counter this deficiency, we define the notion of *well-formed* traces.

Definition 2.5 *For a set of names ρ and trace s we define $rcp(\rho, s)$ inductively as*

$$\begin{aligned} rcp(\rho, \epsilon) &= \rho \\ rcp(\rho, s.(\hat{y})xy) &= rcp(\rho, s) \\ rcp(\rho, s.(\hat{y})\overline{x}y) &= rcp(\rho, s) \cup \hat{y} \end{aligned}$$

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

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

Definition 2.6 *We say $P \ll_\rho Q$, if for every ρ-well-formed trace s, $P \overset{s}{\Longrightarrow}$ implies there is $r \preceq s$ such that $Q \overset{r}{\Longrightarrow}$.* □

For a well-formed trace s the canonical observer $O(s)$ is the same as in Definition 2.3. 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 ρ-well-formed, then $rcp(O(s)) \cap \rho = \emptyset$. Since L$\pi_=$ is a subcalculus of asynchronous π-calculus, Lemma 2.1 holds in L$\pi_=$. Further, since the canonical observer construction is unchanged, the following lemma (which is a weaker form of Lemma 2.2) holds for L$\pi_=$.

Lemma 2.3 *For a well-formed trace s, $O(s) \overset{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$ implies $r \preceq s$.* □

Theorem 2.3 proves the equivalence of $\overset{\sqsubset}{\sim}_\rho$ and $\ll_\rho$ using Lemma 2.5.

Lemma 2.4 *Let $P \overset{s}{\Longrightarrow}$. Then $s = s_1.(\hat{y})xy.s_2$ implies $x \in rcp(rcp(P), s_1)$.*

Proof: Suppose $P \overset{s_1}{\Longrightarrow} P_1 \overset{(\hat{y})xy}{\longrightarrow} P_2 \overset{s_2}{\Longrightarrow}$. We observe that $P_1 \overset{(\hat{y})xy}{\longrightarrow}$ if and only if $x \in rcp(P_1)$. So we are done if we show that $rcp(P_1) \subset rcp(rcp(P), s_1)$. This can shown by a simple induction on s using the locality property. □

Lemma 2.5 *Let ρ be a set of names where $rcp(O) \cap \rho = \emptyset$. Then $P|O \overset{\overline{\mu}\mu}{\Longrightarrow}$ can be unzipped into $P \overset{s}{\Longrightarrow}$ and $O \overset{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ for some s that is ρ-well-formed.*

Proof: The sequence $P|O \overset{\overline{\mu}\mu}{\Longrightarrow}$ can be unzipped into $P \overset{s}{\Longrightarrow}$ and $O \overset{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ for some trace s. While unzipping, we can choose bound names such that $bn(\overline{s}) \cap rcp(O) = \emptyset$. To show that s is ρ-well-formed, we show that if $s = s_1.(\hat{y})\overline{x}y.s_2$, then $x \notin rcp(\rho, s_1)$. By applying Lemma 2.4 to $O \overset{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ we deduce that $x \in rcp(rcp(O), \overline{s_1})$. If $x \in rcp(O)$, then $x \notin rcp(\rho, s_1)$ because $\rho \cap rcp(O) = bn(s_1) \cap rcp(O) = \emptyset$. If $x \notin rcp(O)$, then x must be the bound argument of an output in $\overline{s_1}$, which again implies $x \notin rcp(\rho, s_1)$. Therefore, s is ρ-well-formed. □

Theorem 2.3 *$P \stackrel{\sqsubset}{\sim}_\rho Q$ if and only if $P \ll_\rho Q$.*

Proof: **(if)** Let $P \ll_\rho Q$ and P *may* O for an observer O such that $rcp(O) \cap \rho = \emptyset$. From P *may* O we have $P|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. By Lemma 2.5, this computation can be unzipped into $P \stackrel{s}{\Longrightarrow}$ and $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ for some ρ-well-formed trace s. From $P \ll_\rho Q$, there is a trace $r \preceq s$ such that $Q \stackrel{r}{\Longrightarrow}$. Moreover, $r \preceq s$ implies $r.\mu\mu \preceq s.\mu\mu$. Therefore, by Lemma 2.1, $O \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$. We can zip this with $Q \stackrel{r}{\Longrightarrow}$ to obtain $Q|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, which means Q *may* O.

(only if): Let $P \stackrel{\sqsubset}{\sim}_\rho Q$ and $P \stackrel{s}{\Longrightarrow}$ where s is ρ-well-formed. We have to show that there is a trace $r \preceq s$ such that $Q \stackrel{r}{\Longrightarrow}$. Now, it is easy to show that $O(s) \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$. This can be zipped with $P \stackrel{s}{\Longrightarrow}$ to get $P|O(s) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, that is P *may* $O(s)$. From $P \stackrel{\sqsubset}{\sim}_\rho Q$, we have Q *may* $O(s)$ and therefore $Q|O(s) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. This can be unzipped into $Q \stackrel{r}{\Longrightarrow}$ and $O(s) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$. From Lemma 2.3, it follows that $r \preceq s$. □

For the variant of L$\pi_=$ with mismatch capability on names, the relation $\ll$ defined as in Definition 2.6, but using only laws *L1, L2* and *L3*, characterizes the may preorder.

2.3 The Calculus Lπ

We now investigate the effect of lack of name matching capability. We remove the match operator from L$\pi_=$, to obtain the calculus Lπ. The rules in Table 2.1 except the *MATCH* rule, constitute the operational semantics of Lπ.

The lack of name matching capability further weakens may testing equivalence. For example, the processes $(\nu u)(\overline{x}u|\overline{x}u)$ and $(\nu u, v)(\overline{x}u|\overline{x}v)$ are equivalent in Lπ, but not in L$\pi_=$. For the alternate characterization of $P \stackrel{\sqsubset}{\sim}_\rho Q$, 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 Lπ processes P and Q such that $P \stackrel{\sqsubset}{\sim}_\rho Q$, and if P exhibits s, then Q exhibits *different* traces to satisfy different observers that accept s. For instance, let $P = \overline{x}u_1|\overline{y}u_1|u_1(w).\overline{w}w$ which can exhibit $s = \overline{x}u_1.\overline{y}u_1.u_1(w).\overline{w}w$. The

following Lπ observers accept s.

$$\begin{aligned}
O_1 &= (\nu w)(x(u).y(v).\overline{u}w|w(v).\overline{\mu}\mu) \\
O_2 &= (\nu w)(x(u).y(v).\overline{v}w|w(v).\overline{\mu}\mu) \\
O_3 &= (\nu w)(x(u).y(v).\overline{u_1}w|w(v).\overline{\mu}\mu) \\
O_4 &= (\nu w)(x(u).y(v).(\overline{v}v|\overline{u}u) \mid u_1(z).u_1(z).\overline{u_1}w \mid w(v).\overline{\mu}\mu)
\end{aligned}$$

Now consider

$$Q = (\nu v)(v(z).v(z').(\overline{x}z|\overline{y}z')|\overline{v}u_1|\overline{v}u_2| \; !u_2(z).\overline{u_1}z \mid u_1(w).\overline{w}w)$$

which can satisfy

$$\begin{aligned}
O_1 &\text{ with } r_1 = \overline{x}u_1.\overline{y}u_2.u_1(w).\overline{w}w \\
O_2 &\text{ with } r_2 = \overline{x}u_2.\overline{y}u_1.u_1(w).\overline{w}w \\
O_3 &\text{ with } r_1 \text{ or } r_2, \text{ and} \\
O_4 &\text{ with } r_4 = \overline{x}u_1.\overline{y}u_2.u_2u_2.\overline{u_1}u_2.u_1(w).\overline{w}w
\end{aligned}$$

but cannot exhibit a single trace that can satisfy all four observers. In fact, it is the case that $P \stackrel{\sqsubset}{\sim}_{\emptyset} Q$. Intuitively, although unlike P, Q always exports two different names to 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 ρ, we define a set $T(s, \rho)$ that has a template for each possible dataflow in a computation $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ with $rcp(O) \cap \rho = \emptyset$. Further, if t represents the dataflow in a computation $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$, then it will be the case that $O \stackrel{\overline{t}.\overline{\mu}\mu}{\Longrightarrow}$. 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 $P \stackrel{\sqsubset}{\sim}_\rho Q$ if and only if for every ρ-well-formed trace s that P exhibits, for each $t \in T(s, \rho)$, Q exhibits some $r \preceq t$.

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 $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$, by replacing free input arguments in $\overline{s}$ 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 $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ abbreviates the following computation:

$$O \stackrel{\overline{s_1}}{\Longrightarrow} O_0 \stackrel{xy}{\longrightarrow} O_1 \stackrel{\beta_1}{\longrightarrow} O_2 \stackrel{\beta_2}{\longrightarrow} \cdots O_n \stackrel{\beta_n}{\longrightarrow} \stackrel{\overline{\mu}\mu}{\longrightarrow}$$

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:

$$O \stackrel{\overline{s_1}}{\Longrightarrow} O_0 \stackrel{(\hat{w})xw}{\longrightarrow} O_1 \stackrel{\gamma_1}{\longrightarrow} O_2 \stackrel{\gamma_2}{\longrightarrow} \cdots O_n \stackrel{\gamma_n}{\longrightarrow} \stackrel{\overline{\mu}\mu}{\longrightarrow}$$

where γ_i is obtained from β_i as follows. If β_i is an output action, then γ_i is obtained from β_i by substituting dependent occurrences of y with w. If β_i is an internal delivery of a message $\overline{y}z$ 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 β_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 $O \stackrel{\overline{t}.\overline{\mu}\mu}{\Longrightarrow}$ abbreviate a computation thus obtained. The trace t is a template that explicitly represents all dependencies between received names and subsequent computation steps. The set $T(s,\rho)$ consists of all the templates that can be obtained by this construction starting from arbitrary computations of the form $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ with $rcp(O) \cap \rho = \emptyset$. Note that $T(s,\rho)$ can contain more than one template since different computations of form $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ may have different dependency relations between input arguments in $\overline{s}$ and their subsequent use.

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

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \stackrel{xy}{\longrightarrow} O_1 \stackrel{\overline{s_2}}{\Longrightarrow} \stackrel{\overline{\mu}\mu}{\longrightarrow}$$

We first consider the simple case where $y \notin rcp(O_1)$. 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.

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \stackrel{(\hat{w})xw}{\longrightarrow} O_1' \stackrel{\overline{s_2'}}{\Longrightarrow} \stackrel{\overline{\mu}\mu}{\longrightarrow}$$

where $\overline{s_2'}$ is obtained from $\overline{s_2}$ 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 $\overline{s_2'}$ is that it is obtained from $\overline{s_2}$ 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 2.7 and Lemma 2.7.

Definition 2.7 (random output substitution) *For $\sigma = \{\tilde{u}/\tilde{v}\}$ 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 $bn(P) \cap \{\tilde{v}\} = fn(P)\sigma \cap bn(P) = \emptyset$.*

For a name x we define $x[\sigma] = \{x, x\sigma\}$.

$$\begin{aligned}
0[\sigma] &= \{0\} \\
(x(y).P)[\sigma] &= \{x(y).P' \mid P' \in P[\sigma]\} \\
(\overline{x}y)[\sigma] &= \{\overline{x'}y' \mid x' \in x[\sigma], y' \in y[\sigma]\} \\
(P|Q)[\sigma] &= \{P'|Q' \mid P' \in P[\sigma], Q' \in Q[\sigma]\} \\
((\nu x)P)[\sigma] &= \{(\nu x)P' \mid P' \in P[\sigma]\} \\
(!x(y).P)[\sigma] &= \{!x(y).P' \mid P' \in P[\sigma]\}
\end{aligned}$$

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

$$\begin{aligned}
\epsilon[\sigma] &= \{\epsilon\} \\
((\hat{y})\overline{x}y.s)[\sigma] &= \{(\hat{y})\overline{x}y.s' | s' \in s[\sigma]\} \\
(x(y).s)[\sigma] &= \{x'(y).s' | x' \in x[\sigma], s' \in s[\sigma]\} \\
(xy.s)[\sigma] &= \{x'y'.s' | x' \in x[\sigma], y' \in y[\sigma], s' \in s[\sigma]\}
\end{aligned}$$

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

Lemma 2.6 *If $P_0' \in P_0[w/y]$, then*

1. *$P_0 \xrightarrow{(\hat{v})uv} P_1$ and $w \notin \hat{v}$ implies there is $P_1' \in P_1[w/y]$ such that $P_0' \xrightarrow{(\hat{v})uv} P_1'$.*
2. *$P_0 \xrightarrow{(\hat{v})\overline{u}v} P_1$ and $w \notin \hat{v}$ implies there is $P_1' \in P_1[w/y]$ such that $P_0' \xrightarrow{\alpha} P_1'$ where for some $u' \in u[w/y]$ and $v' \in v[w/y]$*

$$\alpha = \begin{cases} \overline{u'}v' & \text{if } \hat{v} = \emptyset \\ \overline{u'}(v) & \text{otherwise} \end{cases}$$

3. *$P_0 \xrightarrow{\tau} P_1$ implies one of the following;*
   (a) *There is $P_1' \in P_1[w/y]$ such that $P_0' \xrightarrow{\tau} P_1'$.*

(b) $P_1 \equiv (\nu \hat{z})Q$, $w, y \notin \hat{z}$, *and there is* $P_1' \in Q[w/y]$ *such that* $P_0' \xrightarrow{(\hat{z})\overline{w}z.yz} P_1'$.

Proof: See Appendix A. □

Lemma 2.7 *If* $P \stackrel{\overline{s}}{\Longrightarrow}$, $P' \in P[w/y]$, *and* $y \notin rcp(P)$, *then* $P' \stackrel{\overline{s'}}{\Longrightarrow}$ *for some* $s' \in s[w/y]$.

Proof: Since we work modulo alpha equivalence on traces we can assume the hygiene condition $bn(s) \cap \{w, y\} = \emptyset$. Proof is by induction on the number of steps in the transition sequence abbreviated by $P \stackrel{\overline{s}}{\Longrightarrow}$. The base case is obvious by letting $s' = \epsilon$ and the fact that $\epsilon \in \epsilon[w/y]$. For the induction step, suppose $P \stackrel{\overline{s}}{\Longrightarrow}$ can be written as $P \xrightarrow{\overline{\beta}} P_1 \stackrel{\overline{r}}{\Longrightarrow}$. There are three cases depending on β:

1. $P \xrightarrow{(\hat{v})uv} P_1$: Due to locality, $rcp(P_1) \subset rcp(P)$. By Lemma 2.6.1, $P' \xrightarrow{(\hat{v})uv} P_1'$ for some $P_1' \in P_1[w/y]$. Let $\overline{\beta'} = (\hat{v})uv$.

2. $P \xrightarrow{(\hat{v})\overline{u}v} P_1$: We have $rcp(P_1) \subset rcp(P) \cup \hat{v}$. Since by hygiene condition $y \notin \hat{v}$, we have $y \notin rcp(P_1)$. By Lemma 2.6.2, $P' \xrightarrow{\overline{\beta'}} P_1'$ for some $P_1' \in P_1[w/y]$ and $\overline{\beta'}$ is α as stated in Lemma 2.6.2.

3. $P \xrightarrow{\tau} P_1$: By locality $y \notin rcp(P_1)$. Now, we apply Lemma 2.6.3. From $y \notin rcp(P)$ and $P' \in P[w/y]$ we deduce $y \notin rcp(P')$. It follows that only the first case of Lemma 2.6.3 applies, because in the second case y is used as the subject of an input action which implies $y \in rcp(P')$. Therefore, $P' \xrightarrow{\tau} P_1'$ for some $P_1' \in P_1[w/y]$.

In all cases, $y \notin rcp(P_1)$. Then by induction hypothesis, $P_1' \stackrel{\overline{r'}}{\Longrightarrow}$ where $r' \in r[w/y]$. The result follows from the observation that in cases 1 and 2, $\beta'.r' \in (\beta.r)[w/y]$, and in case 3, $r' \in r[w/y] = s[w/y]$. □

Now, suppose $y \in rcp(O_1)$. Then, in the computation

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \xrightarrow{xy} O_1 \stackrel{\overline{s_2}}{\Longrightarrow} \xrightarrow{\overline{\mu}\mu}$$

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

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \stackrel{(\hat{w})xw}{\longrightarrow} O_1' \stackrel{\overline{s_2'}}{\Longrightarrow} \stackrel{\overline{\mu}\mu}{\longrightarrow}$$

where $\overline{s_2'}$ is obtained from $\overline{s_2}$ 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 $\overline{s_2'}$ is obtained from some $r \in s_2[w/y]$ by exposing arbitrary number of internal transitions at any point in $\overline{r}$. The relation between s_2 and s_2' is formalized in Definition 2.8 and Lemma 2.8. To account for the situation where an exposed pair of actions $(\hat{z})\overline{w}z.yz$ export a private name z, we need the following function on traces.

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

Definition 2.8 *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. $(\hat{v})\overline{u}v.s' \in F((\hat{v})\overline{u}v.s, w, y)$ *if* $s' \in F(s, w, y)$
4. $(\hat{z})wz.\overline{y}z.[\hat{z}]s' \in F(s, w, y)$ *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 $F(S, w, y) = \cup_{s \in S} F(s, w, y)$. $\square$

Lemma 2.8 *If $P \stackrel{\overline{s}}{\Longrightarrow}$ and $P' \in P[w/y]$, then $P' \stackrel{\overline{s'}}{\Longrightarrow}$ for some $s' \in F(s[w/y], w, y)$.*

Proof: Since we work modulo alpha equivalence on traces we can assume $bn(s) \cap \{w, y\} = \emptyset$. Proof is by induction on the number of steps in transition sequence $P \stackrel{\overline{s}}{\Longrightarrow}$. The base

case is obvious with $s' = \epsilon$ because $\epsilon \in F(\epsilon[w/y], w, y)$. For the induction step, suppose $P \stackrel{\overline{s}}{\Longrightarrow}$ can be written as $P \stackrel{\overline{\beta}}{\longrightarrow} P_1 \stackrel{\overline{r}}{\Longrightarrow}$. We have three cases:

1. $P \stackrel{(\hat{v})uv}{\longrightarrow} P_1$: Since $w \notin \hat{v}$, by Lemma 2.6.1, there is $P_1' \in P_1[w/y]$ such that $P' \stackrel{(\hat{v})uv}{\longrightarrow} P_1'$. By induction hypothesis $P_1' \stackrel{\overline{r'}}{\Longrightarrow}$ for some $r' \in F(r[w/y], w, y)$. By letting $s' = (\hat{v})\overline{u}v.r'$, we have $s' \in F((\hat{v})\overline{u}v.r[w/y], w, y) = F(s[w/y], w, y)$ and the lemma follows.

2. $P \stackrel{(\hat{v})\overline{u}v}{\longrightarrow} P_1$: Since $w \notin \hat{v}$, by Lemma 2.6.2, there is $P_1' \in P_1[w/y]$ such that $P' \stackrel{\alpha}{\longrightarrow} P_1'$ where for some $u' \in u[w/y]$ and $v' \in v[w/y]$

$$\alpha = \begin{cases} \overline{u'}v' & \text{if } \hat{v} = \emptyset \\ \overline{u'}(v) & \text{otherwise} \end{cases}$$

   By induction hypothesis $P_1' \stackrel{\overline{r'}}{\Longrightarrow}$ for some $r' \in F(r[w/y], w, y)$. By letting $s' = \overline{\alpha}.r'$, we have $s' \in F(\overline{\alpha}.r[w/y], w, y) = F(s[w/y], w, y)$ and the lemma follows.

3. $P \stackrel{\tau}{\longrightarrow} P_1$: Then $r = s$. By Lemma 2.6.3 we have two cases:

   (a) There is $P_1' \in P_1[w/y]$ such that $P' \stackrel{\tau}{\longrightarrow} P_1'$. By induction hypothesis, $P_1' \stackrel{\overline{s'}}{\Longrightarrow}$ for some $s' \in F(s[w/y], w, y)$. The lemma follows from $P' \stackrel{\overline{s'}}{\Longrightarrow}$.

   (b) $P_1 \equiv (\nu\hat{z})Q_1$, $w, y \notin \hat{z}$, and there is $P_1' \in Q_1[w/y]$ such that $P' \stackrel{(\hat{z})\overline{w}z}{\longrightarrow} \stackrel{yz}{\longrightarrow} P_1'$. From $w, y \notin \hat{z}$ we have $(\nu\hat{z})P_1' \in ((\nu\hat{z})Q_1)[w/y]$. By induction hypothesis, $(\nu\hat{z})P_1' \stackrel{\overline{s''}}{\Longrightarrow}$ for some $s'' \in F(s[w/y], w, y)$. It is easy to show that $P_1' \stackrel{\overline{[\hat{z}]s''}}{\Longrightarrow}$. The lemma follows from $(\hat{z})wz.\overline{y}z.[\hat{z}]s'' \in F(s[w/y], w, y)$ and $P' \stackrel{(\hat{z})\overline{w}z}{\longrightarrow} \stackrel{yz}{\longrightarrow} \stackrel{\overline{[\hat{z}]s''}}{\Longrightarrow}$. $\square$

For a trace s and a set of names ρ, we say s is ρ-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

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \stackrel{xy}{\longrightarrow} O_1 \stackrel{\overline{s_2}}{\Longrightarrow} \stackrel{\overline{\mu}\mu}{\longrightarrow}$$

where $\overline{s_1}.xy.\overline{s_2}$ is ρ-normal. If $y \in \rho$ or y is the argument of a bound input in $\overline{s_1}$, 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 ρ-normal trace $s_1.\overline{x}y.s_2$, then O also accepts $s_1.(\hat{w})\overline{x}w.s_2'$ 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 2.9.

Definition 2.9 *For a trace s and a set of names ρ, the set of templates $T(s, \rho)$ is defined modulo alpha equivalence as follows.*

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. $\overline{x}(y).s' \in T(\overline{x}(y).s, \rho)$ *if* $s' \in T(s, \rho \cup \{y\})$
4. $\overline{x}(w).s' \in T(\overline{x}y.s, \rho)$ *if w fresh,* $s' \in T(s'', \rho \cup \{w\})$, *and*
$$s'' \in \begin{cases} s[w/y] & \text{if } y \in \rho \\ F(s[w/y], w, y) & \text{if } y \notin \rho \end{cases}$$

□

Lemma 2.9 *If $P \xrightarrow{xy} P_1$ and $w \notin fn(P)$, then there is $P_1' \in P_1[w/y]$ such that $P \xrightarrow{x(w)} P_1'$.*

Proof: See Appendix A. □

Lemma 2.10 *If $P \stackrel{\overline{s}}{\Longrightarrow}$ and $\rho \cap rcp(P) = \emptyset$, then there is $t \in T(s, \rho)$ such that $P \stackrel{\overline{t}}{\Longrightarrow}$.*

Proof: Since we work modulo alpha equivalence on traces, we assume s is ρ-normal. The proof is by induction on the number of τ transitions and the number of steps in the transition sequence $P \stackrel{\overline{s}}{\Longrightarrow}$ ordered lexicographically. Base case is easy with $t = \epsilon$. For the induction step, we can write $P \xrightarrow{\overline{\beta}} P_1 \stackrel{\overline{r}}{\Longrightarrow}$. Now, there are four cases based on β.

1. $\overline{\beta} = \tau$: By locality, $rcp(P_1) \subset rcp(P)$, and hence $\rho \cap rcp(P_1) = \emptyset$. Further, $r = s$ and the lemma follows from induction hypothesis.

2. $\overline{\beta} = (\hat{y})\overline{x}y$: From ρ-normality of s, we have $\hat{y} \cap \rho = \emptyset$. Since $rcp(P_1) \subset rcp(P) \cup \hat{y}$, we have $\rho \cap rcp(P_1) = \emptyset$. Now, r is ρ-normal and by induction hypothesis, there exists $r' \in T(r, \rho)$ such that $P_1 \stackrel{\overline{r'}}{\Longrightarrow}$. The lemma follows from $(\hat{y})xy.r' \in T(s, \rho)$.

3. $\overline{\beta} = x(y)$: By locality, $(\rho \cup \{y\}) \cap rcp(P_1) = \emptyset$. Furthermore, r is $(\rho \cup \{y\})$-normal. By induction hypothesis, there exists $r' \in T(r, \rho \cup \{y\})$ such that $P_1 \stackrel{\overline{r'}}{\Longrightarrow}$. The lemma follows from $\overline{x}(y).r' \in T(s, \rho)$.

4. $\overline{\beta} = xy$: Let w be fresh, that is $w \notin fn(P) \cup n(s) \cup \rho$. By Lemma 2.9, there is $P_1' \in P_1[w/y]$ such that $P \stackrel{x(w)}{\longrightarrow} P_1'$. Because of locality, $rcp(P_1') \subset rcp(P)$ and therefore $(\rho \cup \{w\}) \cap rcp(P_1') = \emptyset$. We have two subcases:

   - $y \in \rho$: Then $y \notin rcp(P)$ and by locality $y \notin rcp(P_1)$. Then by Lemma 2.7, $P_1' \stackrel{\overline{r''}}{\Longrightarrow}$ for some $r'' \in r[w/y]$. From the proof of Lemma 2.7, it is clear that the computation $P_1' \stackrel{\overline{r''}}{\Longrightarrow}$ has the same number of τ transitions and computation steps as $P_1 \stackrel{\overline{r}}{\Longrightarrow}$.
   - $y \notin \rho$: Then by Lemma 2.8, $P_1' \stackrel{\overline{r''}}{\Longrightarrow}$ for some $r'' \in F(r[w/y], w, y)$. From the proof of Lemma 2.8, it is clear that if the number of τ transitions in $P_1' \stackrel{\overline{r''}}{\Longrightarrow}$ is not less than that in $P_1 \stackrel{\overline{r}}{\Longrightarrow}$, then both computations have exactly the same number of steps.

   In either case, without loss of generality we may assume r'' is $(\rho \cup \{w\})$-normal. Then by induction hypothesis, $P_1' \stackrel{\overline{r'}}{\Longrightarrow}$ for some $r' \in T(r'', \rho \cup \{w\})$. The lemma follows from $\overline{x}(w).r' \in T(s, \rho)$. □

Lemma 2.12 states that template construction in Definition 2.9 preserves ρ-well-formedness.

Lemma 2.11 *Let s be ρ-well-formed. Then for $y \notin \rho$ all traces in $F(s, x, y)$ are ρ-well-formed.*

Proof: See Appendix A. □

Lemma 2.12 *If s is ρ-well-formed then every $t \in T(s,\rho)$ is ρ-well-formed.*

Proof: We prove by induction on derivation of $r \in T(s,\rho)$ that r is ρ-well-formed. The base case $\epsilon \in T(\epsilon,\rho)$ is obvious. For the induction step there are four cases.

1. $s = (\hat{y})xy.s'$, $r = (\hat{y})xy.r'$ and $r' \in T(s',\rho)$. Suppose $r = (\hat{y})xy.r_1.(\hat{v})\overline{u}v.r_2$. Now s' is ρ-well-formed, and by induction hypothesis r' is ρ-well-formed. Then we have $u \notin rcp(r_1,\rho) = rcp((\hat{y})xy.r_1,\rho)$. Hence r is ρ-well-formed.

2. $s = \overline{x}(y).s'$, $r = \overline{x}(y).r'$ and $r' \in T(s',\rho\cup\{y\})$. Suppose $r = \overline{x}(y).r_1.(\hat{v})\overline{u}v.r_2$. Now, s' is $\rho\cup\{y\}$-well-formed, and by induction hypothesis r' is ρ-well-formed. Then we have $u \notin rcp(r_1,\rho\cup\{y\}) = rcp(\overline{x}(y).r_1)$. Further, since s is ρ-well-formed $x \notin \rho$. Hence r is ρ-well-formed.

3. $s = \overline{x}y.s'$, $y \in \rho$, $r = \overline{x}(w).r'$ for some w fresh and $r' \in T(r'',\rho\cup\{w\})$ for some $r'' \in s'[w/y]$. Now s' is ρ-well-formed. Since s is normal $y \notin bn(s')$. From this, and the facts that w is fresh and random substitution on traces does not change output actions, we have r'' is ρ-well-formed. Moreover, since w is fresh we also have r'' is $\rho\cup\{w\}$-well-formed. By induction hypothesis r' is $\rho\cup\{w\}$-well-formed. Further, since s is ρ-well-formed we have $x \notin \rho$. We conclude r is ρ-well-formed.

4. $s = \overline{x}y.s'$, $y \notin \rho$, $r = \overline{x}(w).r'$ for some w fresh and $r' \in T(r'',\rho)$ for some $r'' \in F(s'[w/y],w,y)$. Now, s' is ρ-well-formed, and by the argument in case 3 we have r'' is $\rho\cup\{w\}$-well-formed. By Lemma 2.11, r' is $\rho\cup\{w\}$-well-formed. Further, since s is ρ-well-formed we have $x \notin \rho$. We conclude r is ρ-well-formed. □

The relation $\ll_\rho$ on Lπ processes is defined as follows.

Definition 2.10 *We say $P \ll_\rho Q$, if for every ρ-well-formed trace s, $P \stackrel{s}{\Longrightarrow}$ implies for each $t \in T(s,\rho)$ there is $r \preceq t$ such that $Q \stackrel{r}{\Longrightarrow}$.* □

For $t \in T(s,\rho)$, where s is a ρ-well-formed trace, let $O(t)$ be the canonical observer as defined in Definition 2.3. By Lemma 2.12, since s is ρ-well-formed t is also ρ-well-formed.

Hence $O(t)$ satisfies the locality constraint, and $rcp(O(t)) \cap \rho = \emptyset$. Further, since t is a template, the case $t = \overline{x}y.t'$ does not arise in the construction of the observer. Hence $O(t)$ is an Lπ term. Since Lπ is a subcalculus of asynchronous π-calculus, Lemma 2.1 holds for Lπ. Further, since the canonical observer construction is unchanged, the following lemma (which is a weaker form of Lemma 2.2) holds for Lπ.

Lemma 2.13 *For $t \in T(s, \rho)$, where s is a ρ-well-formed trace, $O(t) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$ implies $r \preceq t$.*

□

Lemma 2.5 holds for Lπ with formally the same proof. Now, we are ready to prove that $\ll_\rho$ is an alternate characterization of $\stackrel{\sqsubset}{\sim}_\rho$.

Theorem 2.4 *$P \stackrel{\sqsubset}{\sim}_\rho Q$ if and only if $P \ll_\rho Q$.*

Proof: **(if)** Let $P \ll_\rho Q$ and P *may* O for an observer O such that $rcp(O) \cap \rho = \emptyset$. From P *may* O we have $P|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. By Lemma 2.5, this computation can be unzipped into $P \stackrel{s}{\Longrightarrow}$ and $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$ for some ρ-well-formed trace s. From Lemmas 2.1 and 2.10 we deduce there is a $t' \in T(s.\mu\mu, \rho)$ such that $r' \preceq t'$ implies $O \stackrel{\overline{r'}}{\Longrightarrow}$. 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 $P \ll_\rho Q$, there is a trace $r \preceq t$ such that $Q \stackrel{r}{\Longrightarrow}$. Moreover, $r \preceq t$ implies $r.\mu\mu \preceq t.\mu\mu = t'$. Therefore, $O \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$. We can zip this with $Q \stackrel{r}{\Longrightarrow}$ to obtain $Q|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, which means Q *may* O.

(only if): Let $P \stackrel{\sqsubset}{\sim}_\rho Q$ and $P \stackrel{s}{\Longrightarrow}$ where s is ρ-well-formed. We have to show for every $t \in T(s, \rho)$ there is a trace $r \preceq t$ such that $Q \stackrel{r}{\Longrightarrow}$. It is easy to show that if $t \in T(s, \rho)$, then $O(t) \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$. This can be zipped with $P \stackrel{s}{\Longrightarrow}$ to get $P|O(t) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, that is P *may* $O(t)$. From $P \stackrel{\sqsubset}{\sim}_\rho Q$, we have Q *may* $O(t)$ and therefore $Q|O(t) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. This can be unzipped into $Q \stackrel{r}{\Longrightarrow}$ and $O(t) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$. From Lemma 2.3, it follows that $r \preceq t$. □

For finitary processes we can obtain a simpler characterization based on a modified version of Definition 2.9 as given below.

Definition 2.11 *For a trace s and a set of names ρ, the set $T_f(s, \rho)$ is defined inductively using the first three rules of Definition 2.9 and the following two.*

4 $\overline{x}(w).s' \in T_f(\overline{x}y.s, \rho)$ *if $y \in \rho, w$ fresh, $s' \in T_f(s'', \rho \cup \{w\})$, and $s'' \in s[w/y]$*

5 $\overline{x}y.s' \in T_f(\overline{x}y.s, \rho)$ *if* $y \notin \rho$, *and,* $s' \in T_f(s, \rho)$ □

The main difference from Definition 2.9 is that output arguments y that are not in ρ are not converted to bound arguments. According to rule 4 of Definition 2.9, 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. Lemma 2.15 helps formalize this observation.

Lemma 2.14

1. $(\hat{y})\overline{x}y.r \preceq \overline{x}(w).s$ *implies* $r \preceq s\{y/w\}$.

2. $r \preceq s_1.s_2$ *implies* $r = r_1.r_2$ *for some* $r_1 \preceq s_1$.

Proof: (1) By induction on the derivation of $(\hat{y})\overline{x}y.r \preceq \overline{x}(w).s$. (2) By induction on the derivation of $r \preceq s_1.s_2$. □

Lemma 2.15 *For a trace* s, *a set of names* ρ, *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* $r \preceq t_f$. □

Proof of Lemma 2.15: For sets of traces R and S, define $R \preceq S$, if for every $s \in S$ there is $r \in R$ such that $r \preceq s$. Then the statement of the lemma can be stated as: for a prefix closed set R of traces with bounded length, $R \preceq T(s, \rho)$ implies $R \preceq T_f(s, \rho)$.

The proof is by induction on the length of s. The base case is easy because $T(\epsilon, \rho) = T_f(\epsilon, \rho) = \{\epsilon\}$. For the induction step we have four cases, of which we only consider the one which is central to the proof, namely where $s = \overline{x}y.s'$ and $y \notin \rho$. The others are routine. We are done if we construct a prefixed closed set R' of traces with bounded length such that $R' \preceq T(s', \rho)$ and $\overline{x}y.R' \subset R$. For, by induction hypothesis, $R' \preceq T(s', \rho)$ implies $R' \preceq T_f(s', \rho)$. Then, $\overline{x}y.R' \preceq \overline{x}y.T_f(s', \rho) = T_f(s, \rho)$, which together with $\overline{x}y.R' \subset R$ implies $R \preceq T_f(s, \rho)$.

Suppose $t' \in T(s', \rho)$ and l is the bound on the length of traces in R. Now, let $t'' = w(z_1).\overline{y}(z'_1) \ldots w(z_n).\overline{y}(z'_n)$ for some $n > len(t') + l$. Now, for w fresh, $\overline{x}(w).t'.t'' \in T(\overline{x}y.s', \rho)$, because $s' \in s'[w/y]$, $s'.t'' \in F(s', w, y)$ and $t'.t'' \in T(s'.t'', \rho \cup \{w\})$. Then, since $R \preceq T(s, \rho)$, there exists $r \in R$ such that $r \preceq \overline{x}(w).t'.t''$. It is easy to see by inspecting *L1-L4* that r can only start with an output action, that is $r = (\hat{z})\overline{x}z.r_1$ for some r_1. By Lemma 2.14.1, $r_1 \preceq (t'.t'')\{z/w\}$. Furthermore, $(t'.t'')\{z/w\} = t'.(t''\{z/w\})$ because w does not occur in t'.

Since the number of outputs in t'' is greater than $len(r_1)$, some of them have to be dropped, which is only possible by an application of *L3*. Further, since the number of these outputs is also greater than $len(t') + len(r_1)$, we conclude that some of these applications of *L3* must involve the inputs in t''. But, such annihilation are possible only if $z = y$, which implies $r = \overline{x}y.r_1$. Furthermore, since $r_1 \preceq t'.t''\{y/w\}$, by Lemma 2.14.2 we have $r_1 = r'.r''$ for some $r' \preceq t'$. Let R' be the prefix closure of the set of all traces r' thus obtained for each $t' \in T(s', \rho)$. By construction, $R' \preceq T(s', \rho)$. Furthermore, from $r = \overline{x}y.r'.r''$ and prefix closure of R, we have $\overline{x}y.R' \subseteq R$. Finally, since the length of traces in R is bounded, so is the length of traces in R'. □

Using this lemma, we can show that for finitary processes we can use $T_f(s, \rho)$ in Definition 2.10 instead of $T(s, \rho)$. The resulting characterization is equivalent to the earlier one for the following reason. Suppose $P \stackrel{s}{\Longrightarrow}$ implies, for every $t \in T(s, \rho)$, there exists $r \preceq t$ such that $Q \stackrel{r}{\Longrightarrow}$. 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 2.15, for every $t_f \in T_f(s, \rho)$, there exists $r \preceq t_f$ such that $Q \stackrel{r}{\Longrightarrow}$. Conversely, suppose $P \stackrel{s}{\Longrightarrow}$ implies that for every $t \in T_f(s, \rho)$ there exists $r \preceq t$ such that $Q \stackrel{r}{\Longrightarrow}$. 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 $P \stackrel{s}{\Longrightarrow}$ implies for every $t \in T(s, \rho)$ there exists $r \preceq t$ such that $Q \stackrel{r}{\Longrightarrow}$.

2.4 The Calculus L$\pi^{-}_{=}$

We now consider a variant of L$\pi_{=}$, called L$\pi^{-}_{=}$, with a restricted name matching ability. An L$\pi^{-}_{=}$ process can compare a given name only against a name that it owns (names are owned by a process in the sense explained in Section 2.2). Thus, a process can use the matching capability to associate additional meaning to only its own names. An environment the process executes in, can neither listen to, nor compare against the names owned by the process. As a result, an environment can not distinguish between names owned by the process. This reduction in observing power, gives rise to a coarser equivalence that is useful in practice. For instance, program transformations such as forward splicing, that transform a program into a version that has the same behavior except in the use of its internal names, can now be shown to be sound.

For convenience, we replace the match construct of L$\pi_{=}$ with the following `case` construct

$$\texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n)$$

which is a macro for the following L$\pi_{=}$ term.

$$\begin{aligned}[\![\texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n)]\!] = \\ (\nu u, v_1, \ldots, v_n)([x = y_1]\overline{u}v_1| \ldots |[x = y_n]\overline{u}v_n| \\ u(w).([w = v_1][\![P_1]\!]| \ldots |[w = v_n][\![P_n]\!]) \qquad u, v_i, w \text{ fresh}\end{aligned}$$

Thus, the process `case` x `of` $(y_1 : P_1, \ldots, y_n : P_n)$ behaves like P_i for some $1 \leq i \leq n$, if $x = y_i$, and like 0 otherwise. If more than one branch is true, then one of them is non-deterministically chosen.

To restrict the matching ability, it is required that in a `case` construct such as the above, the names y_i are owned by the process under consideration. To enforce this, we impose the typing constraint that the y_i's are not bound by an input prefix. Thus, only x can be a received name, and the y_i's are either free or bound by a restriction. We call the y_i's above as *tested* names, and denote the set of all free names used as tested names in a process P by $tn(P)$. Since names in $tn(P)$ are private to P, it is assumed

that P is executed only in an environment that neither listens to, nor matches against these names. Accordingly, the definition of $\stackrel{\sqsubset}{\sim}_\rho$ is modified as follows.

Definition 2.12 *For ρ such that $tn(P), tn(Q) \subset \rho$, we say $P \stackrel{\sqsubset}{\sim}_\rho Q$ if for every O such that $rcp(O) \cap \rho = tn(O) \cap \rho = \emptyset$, P may O implies Q may O.* □

For the alternate characterization of $\stackrel{\sqsubset}{\sim}_\rho$, following the approach used for Lπ, we identify the set of traces that an observer accepting a trace s will also accept due to its restricted matching ability. Let

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \xrightarrow{xy} O_1 \stackrel{\overline{s_2}}{\Longrightarrow} \xrightarrow{\overline{\mu}\mu}$$

Suppose $y \notin rcp(O_1) \cup tn(O_1)$. Then, adapting the discussion in Section 2.3, it follows that for w fresh and some $s_2' \in s_2[w/y]$

$$O \stackrel{\overline{s_1}}{\Longrightarrow} \xrightarrow{x(w)} \stackrel{\overline{s_2'}}{\Longrightarrow} \xrightarrow{\overline{\mu}\mu}$$

This motivates the following definition which is analogous to the definition of templates in Section 2.3.

Definition 2.13 *For a trace s and a set of names ρ, define the set $T(s, \rho)$ as*

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. $\overline{x}(y).s' \in T(\overline{x}(y).s, \rho)$ *if* $s' \in T(s, \rho \cup \{y\})$
4. $\overline{x}(w).s' \in T(\overline{x}y.s, \rho)$ *if* $y \in \rho, w$ *fresh,* $s' \in T(s'', \rho \cup \{w\})$, *and* $s'' \in s[w/y]$
5. $\overline{x}y.s' \in T(\overline{x}y.s, \rho)$ *if* $y \notin \rho$, *and,* $s' \in T(s, \rho)$ □

The idea behind this definition is that if an observer O such that $\rho \cap (rcp(O) \cup tn(O)) = \emptyset$ accepts s, then it also accepts some trace in $T(s, \rho)$. To prove this, we adapt Definition 2.7 for random substitution on processes by adding the following rule.

$\texttt{case}\ x\ of\ (y_1 : P_1, \ldots, y_n : P_n)[\sigma] =$

$$\{\texttt{case}\ x'\ of\ (y_1 : P'_1, \ldots, y_n : P'_n) \mid x' \in x[\sigma], P'_i \in P_i[\sigma]\}$$

Now, with minor modifications to the proofs, Lemma 2.6 holds with the additional condition that $y \notin tn(P_0)$, Lemma 2.7 holds with the condition that $y \notin tn(P)$, Lemma 2.10 holds with the condition that $\rho \cap tn(P) = \emptyset$, and Lemma 2.12 holds as it is.

The relation $\ll_\rho$ on L$\pi^-_=$ processes is the same as in Definition 2.10, except that $P \ll_\rho Q$ is defined only for ρ such that $tn(P), tn(Q) \subset \rho$. The canonical observer $O(t)$ for $t \in T(s, \rho)$ and s a well-formed trace, is defined as in Definition 2.3, except that the `case` construct is used instead of match in the obvious way. Note that if $t = t_1.\overline{x}y.t_2$ then $y \notin rcp(\rho, t_1)$. This implies that $O(t)$ does not use received names as tested names ($O(t)$ is well-typed), and $tn(O) \cap \rho = \emptyset$. Since L$\pi^-_=$ is subcalculus of asynchronous π-calculus, Lemma 2.1 holds for L$\pi^-_=$. Further, since the canonical observer construction is unchanged, Lemma 2.13 (which is a weaker form of Lemma 2.2) holds for L$\pi^-_=$. Now, a simple adaptation of the proof of Theorem 2.4 establishes the alternate characterization of may testing for L$\pi^-_=$, namely, that the relations $\ll_\rho$ and $\stackrel{\sqsubset}{\sim}_\rho$ are identical.

2.5 An Axiomatization of Finitary L$\pi_=$ and Lπ

We first give a sound and complete proof system for $\stackrel{\sqsubset}{\sim}_\rho$ for the finitary fragment of Lπ, i.e. for Lπ 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 2.3 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, $(\nu u)((|_{i\in I} u(u).P_i)|\overline{u}u)$ 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 $\sum_{i\in I} P$. 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 $\sum_{k\in I\uplus J} P_k$. We write $\sqsubseteq$ as a shorthand for $\sqsubseteq_\emptyset$, and $=$ for $=_\emptyset$. Random input substitution on processes $P[w/y]_i$ is defined similar to random output substitution (Definition 2.7), except that only the occurrences of y at the subject of input prefixes in P are randomly substituted with w.

Inference Rules

I1 if $P \sqsubseteq_\rho Q$ and $rcp(R) \cap \rho = \emptyset$, then $(\nu x)P \sqsubseteq_{\rho-\{x\}} (\nu x)Q$, $P|R \sqsubseteq_\rho Q|R$.

I2 if for each $z \in fn(P,Q)$ $P\{z/y\} \sqsubseteq_\rho Q\{z/y\}$ then $x(y).P \sqsubseteq_\rho x(y).Q$

I3 if for each $i \in I$ $P_i \sqsubseteq_\rho \sum_{j\in J} Q_{ij}$ then $\sum_{i\in I} P_i \sqsubseteq_\rho \sum_{i\in I, j\in J} Q_{ij}$

I4 if $\rho_1 \subset \rho_2$ and $P \sqsubseteq_{\rho_1} Q$ then $P \sqsubseteq_{\rho_2} Q$.

Axioms

A1 $G + G = G$

A2 $G \sqsubseteq G + G'$

A3 $P|0 = P$

A4 $P|Q = Q|P$

A5 $(P|Q)|R = P|(Q|R)$

A6 Let $G = \sum_{i\in I} \alpha_i.P_i$ and $G' = \sum_{j\in J} \alpha'_j.P'_j$ where each α_i (resp. α'_j) does not bind free names of G' (resp. G). Then $G|G' = \sum_{i\in I} \alpha_i.(P_i|G') + \sum_{j\in J} \alpha'_j.(G|P'_j)$

A7 $(\nu x)(\sum_{i\in I} P_i) = \sum_{i\in I}(\nu x)P_i$

A8 $(\nu x)(P|Q) = P|(\nu x)Q$ $\quad x \notin n(P)$

A9 $(\nu x)(\overline{x}y|\alpha.P) = \alpha.(\nu x)(\overline{x}y|P)$ $\quad x \notin n(\alpha)$

A10 $(\nu x)(\overline{x}y|x(z).P) = (\nu x)(P\{y/z\})$

A11 $(\nu x)(y(z).P) = \begin{cases} y(z).(\nu x)P & \text{if } x \neq y, x \neq z \\ 0 & \text{if } x = y \end{cases}$

A12 $\overline{x}y|\sum_{i\in I} P_i = \sum_{i\in I}(\overline{x}y|P_i)$ $\quad I \neq \emptyset$

A13 $\alpha.\sum_{i\in I} P_i = \sum_{i\in I} \alpha.P_i$ $\quad I \neq \emptyset$

A14 $P = \sum P$

A15 $x(y).(\overline{u}v|P) \sqsubseteq \overline{u}v|x(y).P$ $\quad y \neq u, y \neq v$

A16 $P\{y/z\} \sqsubseteq \overline{x}y|x(z).P$

A17 $x(u).y(v).P \sqsubseteq y(v).x(u).P$ $\quad u \neq y, u \neq v$

A18 $x(y).(\overline{x}y|P) \sqsubseteq P$ $\quad y \notin n(P)$

A19 $(\nu x)P \sqsubseteq P\{y/x\}$

A20 If $x \in \rho$, $w \neq x$ and $w \neq y$, then $\overline{x}y|z(w).P \sqsubseteq_\rho \sum z(w).(\overline{x}y|P) + \sum z(w).P + \sum Q$, where $Q = \begin{cases} P\{y/w\} & \text{if } x = z \\ 0 & \text{otherwise} \end{cases}$

A21 $\overline{x}y|P \sqsubseteq_\rho (\nu w)(\overline{x}w|\sum_{P'\in P[w/y]_i} P')$ $\quad w$ fresh, $y \in \rho$.

Table 2.3: Laws for Lπ.

While axioms *A1* to *A19* all hold in asynchronous π-calculus [11], axioms *A20* and *A21* are unique to Lπ. *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 $\overline{x}y \sqsubseteq_\rho 0$ as follows. For w fresh,

$$\begin{array}{lll}
\overline{x}y & \sqsubseteq_\rho & \overline{x}y|(\nu w)(w(w).0) \qquad\qquad (A3, A11, I1) \\
& \sqsubseteq_\rho & (\nu w)(\overline{x}y|w(w).0) \qquad\qquad (A8) \\
& \sqsubseteq_\rho & (\nu w)(\sum w(w).0 + \sum w(w).\overline{x}y + \sum 0) \qquad\qquad (A20, I1) \\
& \sqsubseteq_\rho & \sum(\nu w)(w(w).0) + \sum(\nu w)w(w).\overline{x}y + \sum(\nu w)0 \qquad\qquad (A7) \\
& \sqsubseteq_\rho & 0 \qquad\qquad (A1, A11, A14, I3)
\end{array}$$

Axiom *A21* captures the effect of lack of match operator. It is directly motivated from rule 4 of Definition 2.11 for template construction.

The inference rules extend the rules for asynchronous π-calculus to handle parameterization of the may preorder. In fact, the rules for asynchronous π-calculus presented in [11] can be obtained by setting $\rho = \emptyset$ in *I1, I2* and *I3*. *I4* is a new rule that is motivated by Theorem 2.2. We make a few remarks about *I1* which is significantly different from its analogue for asynchronous π-calculus. First, using $\overline{x}y \sqsubseteq_{\{x\}} 0$ (proved above) and *I1*, we get $(\nu x)\overline{x}y \sqsubseteq (\nu x)0$, and by axiom *A19* we have $(\nu x)0 \sqsubseteq 0$. Therefore, $(\nu x)\overline{x}y \sqsubseteq 0$. Note the use of the ability to contract the parameter ρ 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: $\overline{x}y \stackrel{\sqsubseteq}{\sim}_{\{x\}} 0$ but not $\overline{x}y|x(y).\overline{y}y \stackrel{\sqsubseteq}{\sim}_{\{x\}} x(y).\overline{y}y$, for the LHS can satisfy the observer $y(u).\overline{\mu}\mu$ and the RHS can not.

The soundness of rules *I1-I4* can be easily proved directly from Definition 2.1. We only show the argument for *I1*, which is given in Lemma 2.16. Soundness of axioms *A1-A21* is easy to check. For *A1-A19*, whenever $P \sqsubseteq Q$, we have $P \stackrel{s}{\Longrightarrow}$, implies $Q \stackrel{r}{\Longrightarrow}$ such that $r \preceq s$. For $A20$, both LHS and RHS exhibit the same ρ-well-formed traces.

Proof of soundness of axiom *A21* is more involved, and is established in Lemma 2.16. 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 2.16

1. *If* $P \overset{\sqsubset}{\sim}_\rho Q$ *and* $rcp(R) \cap \rho = \emptyset$, *then* $(\nu x)P \overset{\sqsubset}{\sim}_{\rho-\{x\}} (\nu x)Q$, $P|R \overset{\sqsubset}{\sim}_\rho Q|R$.

2. *For* $y \in \rho$ *and* w *fresh,* $\overline{x}y|P \overset{\sqsubset}{\sim}_\rho (\nu w)(\overline{x}w| \sum_{P' \in P[w/y]_i} P')$.

Proof: See Appendix A. □

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 $P \overset{\sqsubset}{\sim}_\rho Q$. Inspired by the alternate characterization, the proof relies on existence of canonical forms for processes.

Definition 2.14 *If* s *is a template, then we call* $\overline{s}$ *a cotemplate. Thus, a cotemplate is a trace with no free inputs. If* s *is well-formed, we say* $\overline{s}$ *is cowell-formed.*

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

$$e(\epsilon) \overset{def}{=} 0 \qquad e(\overline{x}y.s') \overset{def}{=} \overline{x}y|e(s')$$
$$e(\overline{x}(y).s') \overset{def}{=} (\nu y)(\overline{x}y|e(s')) \qquad e(x(y).s') \overset{def}{=} x(y).e(s')$$

   Note that cowell-formedness of s *implies that* $e(s)$ *is an L*π *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.* □

We now establish Lemmas 2.17, 2.18, 2.20 and 2.21, which will be useful in the proof of completeness. Lemma 2.17 states that every process has an equivalent canonical form.

Lemma 2.17 *For every process* P *there is a canonical form* C *such that* $P = C$. □

Lemma 2.18 *(1) If* $e(s) \overset{r}{\Longrightarrow}$, *then* $e(r) \sqsubseteq e(s)$. *(2) If* $s \preceq r$ *then* $e(r) \sqsubseteq e(s)$. □

The proofs of the two lemmas above are formally the same as the proofs of the corresponding lemmas for asynchronous π-calculus [11]. This is because, the proofs of $P = C$ and $e(r) \sqsubseteq e(s)$ constructed using the proof system of [11], can be transformed into proofs in our proof system. This claim is justified by the following observations. First, every Lπ term is also an asynchronous π-calculus term. Second, starting from Lπ terms, every term that appears in the proofs of [11] is also an Lπ 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.

The following lemma is a collection of technical facts that will be useful in establishing Lemmas 2.20 and 2.21.

Lemma 2.19

1. *Let $y \notin \rho$ and y does not occur free as the subject of an input in s. Then, for every ρ-well-formed r such that $e(s\{z/y\}) \stackrel{r}{\Longrightarrow}$ there is a ρ-well-formed cotemplate r' such that $e(s) \stackrel{r'}{\Longrightarrow}$ and $e(r'\{z/y\}) \stackrel{r}{\Longrightarrow}$.*

2. *If $P \stackrel{s}{\Longrightarrow}$ and $e(s) \stackrel{r}{\Longrightarrow}$ then $P \stackrel{r}{\Longrightarrow}$.*

3. *Let r be $\hat{y}$-well-formed and $\hat{y}$-normal. Then $(\nu\hat{y})e(r) = e(r')$, where*

$$r' = \begin{cases} r_1 & \text{if } \hat{y} = \{y\}, r = r_1.y(z).r_2, \text{ and } y \notin n(r_1) \\ r_1.\overline{x}(y).r_2 & \text{if } \hat{y} = \{y\}, r = r_1.\overline{x}y.r_2, \text{ and } y \notin n(r_1) \\ r & \text{otherwise} \end{cases}$$

   Note that the conditions on r imply that the three cases above are exhaustive. Further, $(\nu\hat{y})e(r) \stackrel{r'}{\Longrightarrow}$.

4. *Let $y \notin \rho$, and s be a trace such that y does not occur free in input actions of s. Then for every $t' \in T(s\{z/y\}, \rho)$ there is a $t \in T(s, \rho)$ such that $t\{z/y\} \preceq t'$ using only L4.*

Proof: See Appendix A. □

Lemma 2.20 *Let R contain all the cowell-formed cotemplates r such that $e(s) \stackrel{r}{\Longrightarrow}$ and r is ρ-well-formed. Then $e(s) \sqsubseteq_\rho \sum_{r\in R} e(r)$.*

Proof: For convenience, we write $R(s,\rho)$ to denote the set of all ρ-well-formed cowell-formed cotemplate traces r such that $e(s) \stackrel{r}{\Longrightarrow}$. The lemma can be stated as: for every set of names ρ, $e(s) \sqsubseteq_\rho \sum_{r\in R(s,\rho)} e(r)$. We will be using the following property in the proof, which the reader can verify easily. If $e(s) \stackrel{r}{\Longrightarrow}$ then $len(r) \leq len(s)$.

Without loss of generality, we can assume s is ρ-normal. The proof is by induction on the length of s. For the base case, $s = \epsilon$, we have $e(\epsilon) = 0$, $R(\epsilon,\rho) = \{\epsilon\}$, and the lemma follows using *A14* and *I4*. For the induction step we have three cases:

1. $s = x(y).s_1$: By induction hypothesis we have $e(s_1) \sqsubseteq_\rho \sum_{r'\in R(s_1,\rho)} e(r')$. Clearly, for every $r_1 \in R(s_1,\rho)$, $fn(e(r_1)) = fn(r_1) \subset fn(s_1) = fn(e(s_1))$. Therefore, we are done if we show that for all $z \in fn(s_1)$, $e(s_1)\{z/y\} \sqsubseteq_\rho \sum_{r'\in R(s_1,\rho)} e(r')\{z/y\}$, for then using the fact that $x(y).R(s_1,\rho) \subset R(s,\rho)$, and laws *I2, A2*, we conclude $e(s) \sqsubseteq_\rho \sum_{r\in R(s,\rho)} e(r)$.

   Now, $e(s_1)\{z/y\} = e(s_1\{z/y\})$. By induction hypothesis,

$$e(s_1\{z/y\}) \sqsubseteq_\rho \sum_{r'\in R(s_1\{z/y\},\rho)} e(r')$$

   Since s is cowell-formed, y does not occur free as the subject of an input in s_1, and since s is ρ-normal, $y \notin \rho$. Then, using Lemmas 2.19.1 and 2.18, and laws *I3, A1,* and *A2*, we conclude $\sum_{r'\in R(s_1\{z/y\},\rho)} e(r') \sqsubseteq_\rho \sum_{r'\in R(s_1,\rho)} e(r')\{z/y\}$. By transitivity of $\sqsubseteq_\rho$, we have $e(s_1\{z/y\}) \sqsubseteq_\rho \sum_{r'\in R(s_1,\rho)} e(r')\{z/y\}$.

2. $s = (\hat{y})\overline{x}y.s_1$: By induction hypothesis we have $e(s_1) \sqsubseteq_{\rho\cup\hat{y}} \sum_{r'\in R(s_1,\rho\cup\hat{y})} e(r')$. From $e(s) = (\nu\hat{y})(\overline{x}y|e(s_1))$ and using *I1, A12,* and *A7*, we have

$$e(s) \sqsubseteq_\rho \sum_{r'\in R(s_1,\rho\cup\hat{y})} (\nu\hat{y})(\overline{x}y|e(r')) = \sum_{r'\in R(s_1,\rho\cup\hat{y})} e((\hat{y})\overline{x}y.r')$$

If $x \notin \rho$, then $(\hat{y})\overline{x}y.R(s_1, \rho \cup \hat{y}) \subset R(s, \rho)$, and therefore using *A2* we have $e(s) \sqsubseteq_\rho \sum_{r \in R(s,\rho)} e(r)$ as required. For the case $x \in \rho$, we are done if we show for every $(\hat{y})\overline{x}y.r_1 \in (\hat{y})\overline{x}y.R(s_1, \rho \cup \hat{y})$ that $e((\hat{y})\overline{x}y.r_1) \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_1,\rho)} e(r')$. Following is the reason. If $e(s_1) \stackrel{r_1}{\Longrightarrow}$, we have $e(s) \stackrel{(\hat{y})\overline{x}y.r_1}{\Longrightarrow}$. Then by Lemma 2.19.2, we have $R((\hat{y})\overline{x}y.r_1, \rho) \subset R(s, \rho)$. Then using *I3, A1*, and *A2*, we conclude $e(s) \sqsubseteq_\rho \sum_{r \in R(s,\rho)} e(r)$. To show $e((\hat{y})\overline{x}y.r_1) \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_1,\rho)} e(r')$, we have two cases based on r_1. Without loss of generality, we can assume r_1 is $\rho \cup \hat{y}$-normal.

- $r_1 = \epsilon$. Then $(\hat{y})\overline{x}y.r_1 = (\hat{y})\overline{x}y$, $e((\hat{y})\overline{x}y) = (\nu\hat{y})\overline{x}y$. Since $x \in \rho$, $R((\hat{y})\overline{x}y, \rho) = \{\epsilon\}$, and $e(\epsilon) = 0$. The result follows because $(\nu\hat{y})(\overline{x}y) \sqsubseteq_\rho \sum 0$, which can be derived using the example in Section 2.5 and laws *A14, A19, I1, I4*.

- $r_1 = u(w).r_2$: We only consider the case $u = x$, the other is simpler. Using *A20, I1, A7, A11*, we deduce

$$\begin{aligned} e((\hat{y})\overline{x}y.r_1) = (\nu\hat{y})(\overline{x}y|x(w).e(r_2)) \quad \sqsubseteq_\rho \quad & \sum x(w).e((\hat{y})\overline{x}y.r_2) \\ + \quad & \sum x(w).(\nu\hat{y})e(r_2) \\ + \quad & \sum (\nu\hat{y})e(r_2\{y/w\}) \end{aligned} \tag{2.1}$$

  We are done if we show that for each summand Q in the RHS, $Q \sqsubseteq_\rho \sum_{r' \in T} e(r')$, for some set T of ρ-well-formed traces that Q exhibits. This is because, it is clear from (2.1) that if $Q \stackrel{r'}{\Longrightarrow}$ then $e((\hat{y})\overline{x}y.r_1) \stackrel{r'}{\Longrightarrow}$. Therefore $T \subset R((\hat{y})\overline{x}y.r_1, \rho)$, and by using *I3, A1*, and *A2*, we can conclude

$$e((\hat{y})\overline{x}y.r_1) \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_1,\rho)} e(r')$$

  Now, we consider each summand separately.

  (a) $x(w).e((\hat{y})\overline{x}y.r_2)$: Since $e(s_1) \stackrel{r_1}{\Longrightarrow}$, we have $len(r_1) \leq len(s_1)$. And since $len(r_2) < len(r_1) \leq len(s_1) < len(s)$, we have $len((\hat{y})\overline{x}y.r_2) < len(s)$.

Therefore, we can apply the induction hypothesis to conclude

$$e((\hat{y})\overline{x}y.r_2) \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_2,\rho)} e(r')$$

Now, for every $r' \in R((\hat{y})\overline{x}y.r_2, \rho)$, $fn(e(r')) = fn(r') \subseteq fn(e((\hat{y})\overline{x}y.r_2))$. Now, since r_1 is cowell-formed, w does not occur free as the subject of an input in $(\hat{y})\overline{x}y.r_2$. Further, since r_1 is $\rho \cup \hat{y}$-normal, and hence $w \notin \rho$. Then using arguments similar to that in case 1, we deduce that for each $z \in fn(e((\hat{y})\overline{x}y.r_2))$,

$$e((\hat{y})\overline{x}y.r_2)\{z/w\} \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_2,\rho)} e(r')\{z/w\}$$

Then using *I1* and *A13* we get

$$x(w).e((\hat{y})\overline{x}y.r_2) \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_2,\rho)} x(w).e(r')$$

Now, because $x(w).R((\hat{y})\overline{x}y.r_2, \rho) \subset R(x(w).(\hat{y})\overline{x}y.r_2, \rho)$, using *A2*, we have

$$x(w).e((\hat{y})\overline{x}y.r_2) \sqsubseteq_\rho \sum_{r' \in R(x(w).(\hat{y})\overline{x}y.r_2,\rho)} e(r')$$

(b) $x(w).(\nu\hat{y})e(r_2)$: Since r_1 is $\rho \cup \hat{y}$-well-formed, it also $\hat{y}$-well-formed. Then by Lemma 2.19.3, $(\nu\hat{y})e(r_2) = e(r')$, where r' is as defined in the lemma and $(\nu\hat{y})e(r_2) \stackrel{r'}{\Longrightarrow}$. The reader can check that r' is ρ-well-formed. Now, $fn(e(r')) \subset fn((\nu\hat{y})e(r_2))$. The reader can also check that, for $z \in fn((\nu\hat{y})e(r_2))$, by Lemma 2.19.3,
$((\nu\hat{y})e(r_2))\{z/w\} = e(r')\{z/w\}$. Then by *I2, I4* and *A14*,

$$x(w).(\nu\hat{y})e(r_2) =_\rho e(x(w).r') =_\rho \sum e(x(w).r')$$

Now, $x(w).r'$ is a ρ-well-formed trace, and since $(\nu\hat{y})e(r_2) \stackrel{r'}{\Longrightarrow}$, we have $x(w).(\nu\hat{y})e(r_2) \stackrel{x(w).r'}{\Longrightarrow}$

(c) $(\nu\hat{y})e(r_2\{y/w\})$: Since $len(r_2) < len(s)$, by induction hypothesis and axiom *I1* we have

$$(\nu\hat{y})e(r_2\{y/w\}) \sqsubseteq_\rho \sum_{r'' \in R(r_2\{y/w\}, \rho\cup\hat{y})} (\nu\hat{y})e(r'')$$

Let $r'' \in R(r_2\{y/w\}, \rho \cup \hat{y})$. We have, r'' is also $\hat{y}$-well-formed. Then by Lemma 2.19.3, $(\nu\hat{y})e(r'') = e(r')$, where r' is as defined in the lemma and $(\nu\hat{y})e(r'') \stackrel{r'}{\Longrightarrow}$. The reader can check r' is ρ-well-formed. Further, since $e(r_2\{y/w\}) \stackrel{r''}{\Longrightarrow}$, using Lemma 2.19.2 we can show $(\nu\hat{y})e(r_2\{y/w\}) \stackrel{r'}{\Longrightarrow}$. Let R be the set of all r' that are obtained for each $r'' \in R(r_2\{y/w\}, \rho\cup\hat{y})$. Then, using *I3, I4, A1, A14* and transitivity of $\sqsubseteq_\rho$, we conclude

$$(\nu\hat{y})e(r_2\{y/w\}) \sqsubseteq_\rho \sum_{r' \in R} e(r')$$

- $r_1 = (\hat{v})\overline{u}v.r_2$: Using axioms *A4, A5,* and *A8*, we deduce

$$e((\hat{y})\overline{x}y.r_1) = e((\hat{y})\overline{x}y.(\hat{v})\overline{u}v.r_2) = e((\hat{v'})\overline{u}v.(\hat{y'})\overline{x}y.r_2) \tag{2.2}$$

where (since r_1 is $\rho \cup \hat{y}$-normal)

$$\hat{v'} = \begin{cases} \hat{y} & \text{if } \hat{y} = \{v\} \\ \hat{v} & \text{otherwise} \end{cases} \quad \text{and} \quad \hat{y'} = \hat{y} - \hat{v'}$$

By induction hypothesis, we have

$$e((\hat{y'})\overline{x}y.r_2) \sqsubseteq_{\rho\cup\hat{v'}} \sum_{r' \in R((\hat{y'})\overline{x}y.r_2, \rho\cup\hat{v'})} e(r')$$

Then, using *I1, A7* and *A12* we deduce

$$e((\hat{v'})\overline{u}v.(\hat{y'})\overline{x}y.r_2) \sqsubseteq_\rho \sum_{r' \in R((\hat{y'})\overline{x}y.r_2, \rho \cup \hat{v'})} e((\hat{v'})\overline{u}v.r') \tag{2.3}$$

Now, $u \notin \rho \cup \hat{y}$, because $r_1 \in R(s_1, \rho \cup \hat{y})$. Then for every $r' \in R((\hat{y'})\overline{x}y.r_2, \rho \cup \hat{v'})$, $(\hat{v'})\overline{u}v.r'$ is ρ-well-formed. Further, $e((\hat{y'})\overline{x}y.r_2) \stackrel{r'}{\Longrightarrow}$ implies $e((\hat{v'})\overline{u}v.(\hat{y'})\overline{x}y.r_2) \stackrel{(\hat{v'})\overline{u}v.r'}{\Longrightarrow}$. Therefore,

$$(\hat{v'})\overline{u}v.R(\hat{y'}\overline{x}y.r_2, \rho \cup \hat{v'}) \subset R((\hat{v'})\overline{u}v.(\hat{y'})\overline{x}y.r_2, \rho) = R((\hat{y})\overline{x}y.(\hat{v})\overline{u}v.r_2, \rho) \tag{2.4}$$

Finally, from (2.2), (2.3) and (2.4), and using *I4, A2* we obtain

$$e((\hat{y})\overline{x}y.r_1) \sqsubseteq_\rho \sum_{r' \in R((\hat{y})\overline{x}y.r_1, \rho)} e(r')$$

□

Lemma 2.21 $e(s) \sqsubseteq_\rho \sum_{t \in T_f(s,\rho)} e(t)$.

Proof: The proof is by induction on the length of s. Without loss of generality we may assume s is ρ-normal. The base case follows from $0 \sqsubseteq \sum_{t \in \{\epsilon\}} 0$ which holds by *A14*. For the induction step we have three cases:

1. $s = \overline{x}y.s'$: Then $e(s) = \overline{x}y|e(s')$. There are two subcases:

   (a) $y \notin \rho$: From induction hypothesis we have

   $$e(s') \sqsubseteq_\rho \sum_{t' \in T(s',\rho)} e(t')$$

   Using laws *I1* and *A12* in that order we get

   $$\overline{x}y|e(s') \sqsubseteq_\rho \overline{x}y| \sum_{t' \in T(s',\rho)} e(t') =_\rho \sum_{t' \in T(s',\rho)} \overline{x}y|e(t') = \sum_{t' \in T(s',\rho)} e(\overline{x}y.t') = \sum_{t \in T(s,\rho)} e(t)$$

(b) $y \in \rho$: It is easy to check that for w fresh, every $s'' \in s'[w/y]$ is a cowell-formed cotemplate. Further, $len(s'') = len(s')$. Hence by induction hypothesis, we have

$$e(s'') \sqsubseteq_\rho \sum_{t' \in T(s'',\rho)} e(t')$$

Then, by *I3*

$$\sum_{s'' \in s'[w/y]} e(s'') \sqsubseteq_\rho \sum_{t' \in T(s'[w/y],\rho)} e(t') \tag{2.5}$$

Now, since $y \in \rho$, by law *A21* we have

$$e(s) = \overline{x}y|e(s') \sqsubseteq_\rho (\nu w)(\overline{x}w| \sum_{P \in e(s')[w/y]_i} P)$$

The reader may check that for any $P \in e(s')[w/y]_i$, $P = e(s'')$ for some $s'' \in s'[w/y]$ and vice versa. Using this we get

$$e(s) \sqsubseteq_\rho (\nu w)(\overline{x}w| \sum_{s'' \in s'[w/y]} e(s''))$$

Now using 2.5, laws *I1* and *I3*, we have

$$e(s) \sqsubseteq_\rho (\nu w)(\overline{x}w| \sum_{t' \in T(s'[w/y],\rho)} e(t'))$$

Now, using laws *A7* and *A12*, we get

$$e(s) \sqsubseteq_\rho \sum_{t' \in T(s'[w/y],\rho)} e(\overline{x}(w).t') = \sum_{t \in T(s,\rho)} e(t)$$

2. $s = \overline{x}(y).s'$: Then $e(s) = (\nu y)(\overline{x}y|e(s'))$. By induction hypothesis, we have

$$e(s') \sqsubseteq_{\rho \cup \{y\}} \sum_{t' \in T(s',\rho \cup \{y\})} e(t')$$

Since s is ρ-normal, we have $y \notin \rho$. Using this and law *I1*, we get

$$e(s) \sqsubseteq_\rho (\nu y)(\overline{x}y| \sum_{t' \in T(s', \rho \cup \{y\})} e(t'))$$

Now, using laws *A7* and *A12*, we get

$$e(s) \sqsubseteq_\rho \sum_{t' \in T(s', \rho \cup \{y\})} e(\overline{x}(y).t') = \sum_{t \in T(s, \rho)} e(t)$$

3. $s = x(y).s'$: Then $e(s) = x(y).e(s')$. We are done if we show

$$e(s')\{z/y\} \sqsubseteq_\rho \left(\sum_{t' \in T(s', \rho)} e(t') \right) \{z/y\} \tag{2.6}$$

for every $z \in fn(e(s'))$, because then by laws *I2*, *A13* and *I4* we have

$$e(s) \sqsubseteq_\rho x(y). \sum_{t' \in T(s', \rho)} e(t') =_\rho \sum_{t' \in T(s', \rho)} x(y).e(t') = \sum_{t' \in T(s', \rho)} e(x(y).t') = \sum_{t \in T(s, \rho)} e(t)$$

Now we prove 2.6. By applying the induction hypothesis to $s'\{z/y\}$, we have

$$e(s')\{z/y\} = e(s'\{z/y\}) \sqsubseteq_\rho \sum_{t'' \in T(s'\{z/y\}, \rho)} e(t'')$$

Since

$$\left(\sum_{t' \in T(s', \rho)} e(t') \right) \{z/y\} = \sum_{t' \in T(s', \rho)} e(t')\{z/y\} = \sum_{t' \in T(s', \rho)} e(t'\{z/y\})$$

So, we are done if we show

$$\sum_{t'' \in T(s'\{z/y\}, \rho)} e(t'') \sqsubseteq_\rho \sum_{t' \in T(s', \rho)} e(t'\{z/y\})$$

Since s is ρ-normal, $y \notin \rho$. Further, since s is a cowell-formed cotemplate, y cannot occur free in the input actions of s'. Then by Lemma 2.19.4, for every $t'' \in T(s'\{z/y\}, \rho)$ there is a $t' \in T(s', \rho)$ such that $t'\{z/y\} \preceq t''$. Then by Lemma 2.18.2 we have $e(t'') \stackrel{\sqsubset}{\sim} e(t'\{z/y\})$, and 2.6 follows from laws *I3, I4, A1* and *A2*. □

Note that the summations in Lemmas 2.20 and 2.21 are finite because R and $T_f(s, \rho)$ 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 $fn(r) \subset fn(e(s))$, 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 2.5 *For finite Lπ processes P, Q and a set of names ρ, $P \sqsubseteq_\rho Q$ if and only if $P \stackrel{\sqsubset}{\sim}_\rho Q$.*

Proof: The only-if part follows from the soundness of laws in Table 2.3. We prove the if part. By lemma 2.17 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 the form $\sum_{s \in S_1} e(s)$ and Q is of the form $\sum_{s \in S_2} e(s)$. Using Lemma 2.20, and laws *I3, A1*, we get $P \sqsubseteq_\rho \sum_{r \in R} e(r)$, where R is the set of ρ-well-formed cowell-formed cotemplates that P exhibits. Using Lemma 2.21 and laws *I3, A1*, we have $\sum_{r \in R} e(r) \sqsubseteq_\rho \sum_{t \in T} e(t)$, 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 ρ-well-formed r that P exhibits. Using the characterization of may preorder based on $T_f(r, \rho)$, we have $P \stackrel{\sqsubset}{\sim}_\rho Q$ implies there is $s' \preceq t$ such that $Q \stackrel{s'}{\Longrightarrow}$. It follows that for some $s \in S_2$, $e(s) \stackrel{s'}{\Longrightarrow}$. Since $Q \stackrel{s'}{\Longrightarrow}$, 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 2.18.2 and law *I4*, $e(t) \sqsubseteq_\rho e(s')$. Further, by Lemma 2.18.1 and law *I4*, $e(s') \sqsubseteq_\rho e(s)$. Hence by transitivity of $\sqsubseteq_\rho$, we have $e(t) \sqsubseteq_\rho e(s)$. Since $t \in T$ is arbitrary, using laws *I3, A1*, and *A2*, we deduce $\sum_{t \in T} e(t) \sqsubseteq_\rho \sum_{s \in S_2} e(s)$. The result follows from transitivity of $\sqsubseteq_\rho$. □

We obtain a complete proof system for L$\pi_=$ by dropping axiom *A21* 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.

2.6 Discussion and Related Work

An asynchronous variant of π-calculus first appeared in [41]. The main idea was to replace the output prefix $\overline{x}y.P$ of π-calculus with the simpler output particle $\overline{x}y$, that has no continuation. It was shown that the resulting calculus was expressive enough to encode the π-calculus. However, the formulation was not quite satisfactory as it suffered from the problem of assigning infinitary descriptions for even simple non-recursive processes. The main reason for this was that in an asynchronous setting, the environment outputs are to be immediately executed, and hence processes should be receptive, i.e. ready to receive any message sent by the environment at any time. To model this, all possible process inputs were explicitly introduced resulting in a system where the same operational description applies to the process 0 and $!x(y).\overline{x}y$. Later formulations of asynchronous π-calculus [9, 12] avoided this problem by retaining the traditional finitary transition system of the π-calculus, and making up for the shortcoming by modifying the meta theory if necessary. We have adopted this strategy in the study of our variants, and have modified the trace semantics accordingly.

The importance of the locality property and disallowing name matching was first observed in [58], where the calculus Lπ is presented. There have been extensive investigations of bisimulation-based behavioral equivalences on Lπ and related variants of π-calculus. Merro and Sangiorgi [58] study a behavioral equivalence called barbed-congruence, and show that a variant of asynchronous early bisimulation [9] provides an alternate characterization for the equivalence.

Pierce and Sangiorgi [71] generalize Lπ into a typed π-calculus where names can be tagged with input/output capabilities. Three types of capabilities are defined: r for read, w for write, and b for both. The recipient of a name tagged with r can use the name to only receive messages targeted to the name. A name tagged with w can be

used only for sending messages, and a name tagged with b can be used for both sending and receiving messages. Thus, in this typed calculus one can express processes that selectively distribute different capabilities on names. The locality property is a special case where only the output capability on names can be passed. Boreale and Sangiorgi [13] investigate barbed congruence in a typed π-calculus with input/output capabilities and no name matching, and show that the equivalence is characterized by a typed variant of bisimulation. Merro [57] characterizes barbed congruence in the more restricted setting of asynchronous π-calculus with no name matching (no capability types, and no locality in particular). He defines the so called synonymous bisimulation and shows that it characterizes barbed congruence in this setting.

Hennessy and Rathke [37] study may and must testing equivalences in a typed π-calculus with input/output capabilities as described above. They define a novel labeled transition system over configurations which are process terms with two typed environments, one that constrains the process and the other the environment. They show that the standard definitions of trace and acceptance sets [36] 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 Lπ 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 a given an axiomatization of may testing, which is not pursued by Hennessy and Rathke.

Chapter 3

The Actor Model as a Typed Asynchronous π-Calculus

We impose on the π-calculus, the object paradigm proposed by the Actor Model. While asynchrony and locality are inherent features of the Actor Model, there are several additional ones to be accounted for. These are captured through a type system that enforces a certain discipline in the use of names [93]. The result is a typed π-calculus, called Aπ. The formal connection thus established between the two models is exploited to use the techniques developed in Chapter 2, to obtain an alternate characterization of may testing for Actors [8].

The Actor Model postulates fairness in message delivery; the delivery of a message can only be delayed for a finite but unbounded amount of time. But we do not account for fairness here as it has no effect on the theory of may testing. May testing is concerned only with the occurrence of an event after a finite computation, while fairness affects only potentially infinite computations. The reader is referred to [91], where we present a version of Aπ with fairness.

In Section 3.1, we give a brief and informal description of the Actor Model. In Section 3.2, we present the syntax, type system, and semantics of Aπ. In Section 3.3, we present an alternate characterization of may testing in Aπ. In Section 3.4, we consider variants

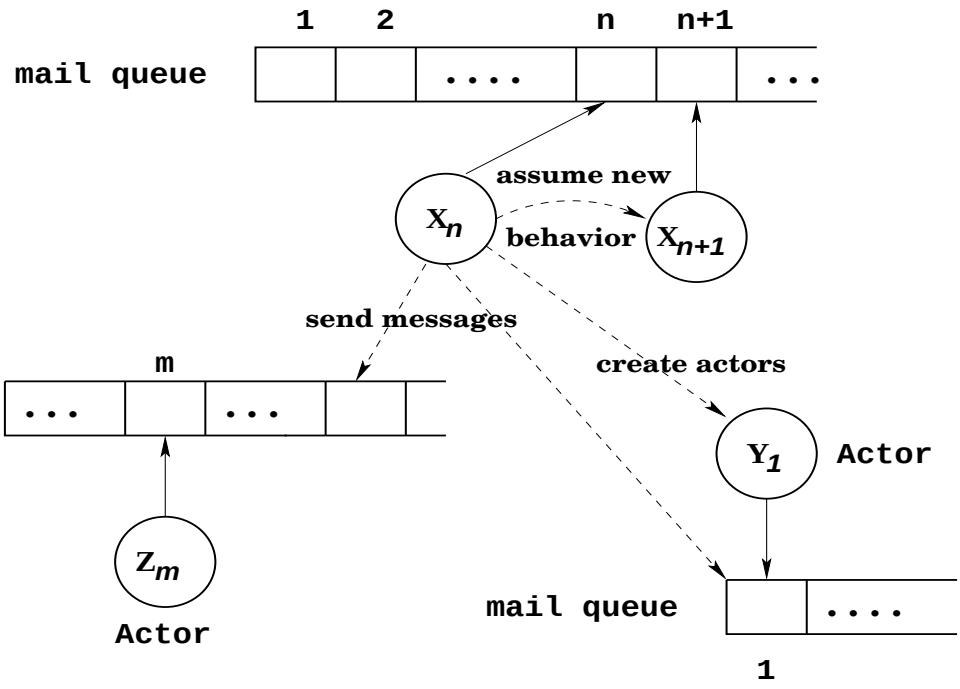


Figure 3.1: A diagram illustrating computation in an actor system.

of Aπ that differ in name matching capability. In Section 3.5, we discuss related work on actor semantics.

3.1 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 [5]. 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 [24]. 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 (see Figure 3.1: (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).

3.2 The Calculus Aπ

The syntax of Aπ is the same as that of L$\pi_{=}^{-}$ except that recursive definitions are used instead of replication. Specifically, the set of configurations is given by the following grammar.

$$P := \quad 0 \;\mid\; x(y).P \;\mid\; \overline{x}y \;\mid\; (\nu x)P \;\mid\; P_1|P_2 \;\mid\; \texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n) \;\mid\; B\langle \tilde{x}; \tilde{y} \rangle$$

We use recursive definitions instead of replication, as the former is more convenient in expressing actor systems. It is well known that replication and recursive definitions are equally expressive in that one can be encoded using the other [63].

Following is the intended interpretation of Aπ terms as actor configurations. The nil term 0, represents an empty configuration. The output term $\overline{x}y$, represents a configuration with a single message targeted to x and with contents y. The input term $x(y).P$ represents a configuration with an actor x whose behavior is $(\tilde{y})P$. The restriction $(\nu x)P$ is the same as P, except that x is now private to P. The composition $P_1|P_2$ is a configuration containing all the actors and messages in P_1 and P_2. The configuration $\texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n)$ behaves like P_i if $x = y_i$, and like 0 if $x \neq y_i$ for $1 \leq i \leq n$. If more than one branch is true, one of them is non-deterministically chosen. This construct allows actor behaviors to be non-deterministic, thus deviating slightly from the Actor Model. But this is not a severe deviation, as the non-determinism is strictly internal; the choice can not be influenced by external interactions, such as the

receipt of a message (see the transition semantics in Section 3.2.2). The term $B\langle\tilde{u};\tilde{v}\rangle$ is a behavior instantiation. The identifier B has a single defining equation of the form $B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$, where $\tilde{x}$ is a tuple of distinct names of length 1 or 2, x_1 denotes the first component of $\tilde{x}$, and $\tilde{x}$, $\tilde{y}$ together contain exactly the free names in $x_1(z).P$. The definition provides a template for an actor behavior. For an instantiation $B\langle\tilde{u};\tilde{v}\rangle$ we assume $len(\tilde{u}) = len(\tilde{x})$, and $len(\tilde{v}) = len(\tilde{y})$.

Before presenting the type system, a few notational conventions (in addition to those in Chapter 2) are in order. By $\tilde{x}, \hat{z}$ we mean $\tilde{x}, z$ if $\hat{z} = \{z\}$, and $\tilde{x}$ otherwise. Let $X \subset \mathcal{N}$. We write $\sigma(X)$ to denote the set obtained by applying the substitution σ to each element of X. We assume $\bot, * \notin \mathcal{N}$, and define $X^* = X \cup \{\bot, *\}$. For $f : X \to X^*$, we define $f^* : X^* \to X^*$ as $f^*(x) = f(x)$ for $x \in X$ and $f(\bot) = f(*) = \bot$. Further, if σ 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 $\sigma(*) = *$.

3.2.1 Type System

Not all terms represent actor configurations. The object paradigm embodied in the Actor Model is to be captured through a type system. Strictly enforcing all actor properties would make Aπ 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 transfered. 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 $\rho; f \vdash P$, where ρ is the set of free names in P that denote actors in P, and $f : \rho \to \rho^*$ is a function that relates actors in P to the temporary

names they have assumed currently. Specifically, $f(x) = \perp$ means that x is a regular actor name and not a temporary one, $f(x) = *$ means x is the temporary name of an actor with a private name (bound by a restriction), and $f(x) = y \notin \{\perp, *\}$ means that actor y has assumed the temporary name x. The function f has the following properties: for all $x, y \in \rho$, $f(x) \neq x$, $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$, and $f^*(f(x)) = \perp$. 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 3.1 *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*

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

   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 $x, y \in \rho_1 \cup \rho_2$, $f(x) \neq x$, $f^*(f(x)) = \perp$, and $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$.* □

Definition 3.2 *For a tuple $\tilde{x}$, we define $ch(\tilde{x}) : \{\tilde{x}\} \to \{\tilde{x}\}^*$ as $ch(\epsilon) = \{\}$, and if $len(x) = n$, $ch(\tilde{x})(x_i) = x_{i+1}$ for $1 \leq i < n$ and $ch(\tilde{x})(x_n) = \perp$.* □

$$\textit{NIL:}\quad \emptyset;\{\} \vdash 0 \qquad\qquad \textit{MSG:}\quad \emptyset;\{\} \vdash \overline{x}y$$

$$\textit{ACT:}\quad \frac{\rho; f \vdash P}{\{x\} \cup \hat{z}; ch(x,\hat{z}) \vdash x(y).P}\ \texttt{if}\ \begin{array}{l} \rho - \{x\} = \hat{z},\ y \notin \rho,\ \texttt{and} \\ f = \begin{cases} ch(x,\hat{z}) & \texttt{if}\ x \in \rho \\ ch(\epsilon,\hat{z}) & \text{otherwise} \end{cases} \end{array}$$

$$\textit{CASE:}\quad \frac{\forall 1 \le i \le n\quad \rho_i; f_i \vdash P_i}{(\cup_i \rho_i); (f_1 \oplus f_2 \oplus \ldots \oplus f_n) \vdash \texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n)}\ \texttt{if}\ f_i\ \texttt{are mutually compatible}$$

$$\textit{COMP:}\quad \frac{\rho_1; f_1 \vdash P_1 \quad \rho_2; f_2 \vdash P_2}{\rho_1 \cup \rho_2; f_1 \oplus f_2 \vdash P_1 | P_2}\ \texttt{if}\ \ \rho_1 \cap \rho_2 = \phi$$

$$\textit{RES:}\quad \frac{\rho; f \vdash P}{\rho - \{x\}; f|(\rho - \{x\}) \vdash (\nu x)P}$$

$$\textit{INST:}\quad \{\tilde{x}\}; ch(\tilde{x}) \vdash B\langle \tilde{x}; \tilde{y} \rangle \quad \texttt{if}\ len(\tilde{x}) = 2\ \texttt{implies}\ \ x_1 \neq x_2$$

Table 3.1: Type rules for Aπ.

The type rules are shown in Table 3.1. Rules *NIL* and *MSG* are obvious. In the *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. This is the locality property discussed in Chapter 2. 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 *sink* behavior that simply consumes all messages it receives. With this interpretation the persistence property is not violated.

The compatibility check in *CASE* 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 *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 P, then y's role as a temporary name is remembered but x is forgotten. The

INST rule assumes that if $len(\tilde{x}) = 2$ then $B\langle\tilde{x};\tilde{y}\rangle$ 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 $B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$ and substitution $\sigma = \{\tilde{u},\tilde{v}/\tilde{x},\tilde{y}\}$ that is one-to-one on $\{\tilde{x}\}$, the judgment $\{\tilde{u}\}; ch(\tilde{u}) \vdash (x_1(z).P)\sigma$ should be derivable. From Lemma 3.2, it follows that this constraint is satisfied if $\{\tilde{x}\}; ch(\tilde{x}) \vdash x_1(z).P$ is derivable. Thus, a preterm is well-typed only if for each accompanying behavior definition $B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$, the judgment $\{\tilde{x}\}; ch(\tilde{x}) \vdash x_1(z).P$ is derivable.

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

Lemma 3.1 *If $\rho; f \vdash P$ then $\rho \subset fn(P)$, and for all $x, y \in \rho$, $f(x) \neq x$, $f^*(f(x)) = \perp$, and $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$. Further, if $\rho'; f' \vdash P$ then $\rho = \rho'$ and $f = f'$.* □

Proof: By structural induction on P. □

It is easy to show that the type system respects alpha equivalence , i.e. if $P_1 \equiv_\alpha P_2$ then $\rho; f \vdash P_1$ if and only if $\rho; f \vdash P_2$. The proof is by induction on the length of a derivation of $P_1 \equiv_\alpha P_2$.

Not all substitutions on a term P yield terms. A substitution σ may identify distinct actor names in P and therefore violate the uniqueness property. But, if σ renames different actors in P to different names, then $P\sigma$ will be well typed.

Lemma 3.2 *If $\rho; f \vdash P$ and σ is one-to-one on ρ then $\sigma(\rho); f\sigma \vdash P\sigma$.* □

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

The proof is by induction on the length of a derivation of $\rho; f \vdash P$. It is straightforward to verify the base cases where the derivation is a direct application of *NIL, MSG* or *INST*. For the induction step, we consider only two cases; the others are simple.

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

$$ACT: \quad \frac{\rho'; f' \vdash P'}{\{x\} \cup \hat{z}; ch(x, \hat{z}) \vdash x(y).P'} \quad \texttt{if} \quad \begin{array}{l} \rho' - \{x\} = \hat{z},\ y \notin \rho',\ \texttt{and} \\ f' = \begin{cases} ch(x, \hat{z}) & \texttt{if}\ \ x \in \rho' \\ ch(\epsilon, \hat{z}) & \text{otherwise} \end{cases} \end{array}$$

Note that by hygiene condition $P\sigma = \sigma(x)(y).(P'\sigma)$. Since σ is an injection on $\{x\} \cup \hat{z}$ so it is on ρ', and thus by induction hypothesis $\sigma(\rho'); f'\sigma \vdash P'\sigma$. This, together with $\rho' - \{x\} = \hat{z}$, also implies $\sigma(\rho') - \sigma(\{x\}) = \sigma(\hat{z})$. By Lemma 3.1, we have $\rho' \subset fn(P')$, and hence by the hygiene condition $y \notin \sigma(\rho')$. Since σ is an injection on $\{x\} \cup \hat{z}$ and $\rho' \subset \{x\} \cup \hat{z}$, 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 *ACT* rule to get $\sigma(\{x\}) \cup \sigma(\hat{z}); ch(\sigma(x), \sigma(\hat{z})) \vdash \sigma(x)(y).(P'\sigma)$, i.e. $\sigma(\{x\} \cup \hat{z}); ch(x, \hat{z})\sigma \vdash \sigma(x)(y).(P'\sigma)$.

2. $P = \texttt{case}\ x\ of\ (y_1 : P_1, \ldots, y_n : P_n)$: The the last derivation step is

$$CASE: \quad \frac{\forall 1 \leq i \leq n \quad \rho_i; f_i \vdash P_i}{(\cup_i \rho_i); (f_1 \oplus f_2 \oplus \ldots \oplus f_n) \vdash \texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n)}$$

Since σ is an injection on $\cup_i \rho_i$, so it is on ρ_i. Then, by induction hypothesis $\sigma(\rho_i); f_i\sigma \vdash P_i\sigma$. The reader may verify that the facts f_i are mutually compatible, and σ is an injection on $\cup_i \rho_i$, together imply $f_i\sigma$ are mutually compatible. We can now apply the *CASE* rule to get $\cup_i \sigma(\rho_i); f_1\sigma \oplus \ldots \oplus f_n\sigma \vdash \texttt{case}\ x\sigma\ \texttt{of}\ (y_1\sigma : P_1\sigma, \ldots, y_n\sigma : P_n\sigma)$. The result follows from the fact that $\cup_i \sigma(\rho_i) = \sigma(\cup_i \rho_i)$ and $f_1\sigma \oplus \ldots \oplus f_n\sigma = (f_1 \oplus \ldots \oplus f_n)\sigma$, which can be verified easily. □

3.2.2 Operational Semantics

The transition rules for Aπ are the same as in Table 2.1 except that the *MATCH* and *RES* rules are replaced by the rules shown in Table 3.2.

$$\textit{BRNCH:}\quad \texttt{case}\ x\ \textbf{of}\ (y_1 : P_1, \ldots, y_n : P_n) \xrightarrow{\tau} P_i \quad \texttt{if}\ x = y_i$$

$$\textit{BEHV:}\quad \frac{(x_1(z).P)\{(\tilde{u},\tilde{v})/(\tilde{x},\tilde{y})\} \xrightarrow{\alpha} P'}{B\langle\tilde{u};\tilde{v}\rangle \xrightarrow{\alpha} P'} \quad B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$$

Table 3.2: New transition rules for Aπ.

Note that to retain internal non-determinism the *BRNCH* rule has been so defined instead of as

$$\textit{BRNCH:}\quad \frac{P_i \xrightarrow{\alpha} P_i'}{\texttt{case}\ x\ \textbf{of}\ (y_1 : P_1, \ldots, y_n : P_n) \xrightarrow{\alpha} P_i'} \quad x = y_i$$

in which case the choice is influenced by external interactions such as the receipt of a message.

The structural congruence laws on Aπ terms are, laws 1 and 2 of asynchronous π-calculus (Section 2.1), and the following

$$B\langle\tilde{u};\tilde{v}\rangle \equiv (x_1(z).P)\{(\tilde{u},\tilde{v})/(\tilde{x},\tilde{y})\} \quad \texttt{if} \begin{cases} B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P, \\ len(\tilde{u}) = len(\tilde{x}), len(\tilde{v}) = len(\tilde{y}) \end{cases}$$

As usual, structurally congruent configurations have the same transitions.

The following theorem states that the type system respects the transition rules.

Theorem 3.1 *If P is well-typed and $P \xrightarrow{\alpha} P'$ then P' is well-typed.*

Proof: We prove a stronger statement which is that, if $\rho; f \vdash P$ and $P \xrightarrow{\alpha} P'$, then $\rho'; f' \vdash P'$ for some $\rho' \subset \rho \cup bn(\alpha)$, and f'.

The proof is by induction on the derivation of $P \xrightarrow{\alpha} P'$. There are two base cases depending on whether the derivation is a direct application of *INP* or *OUT* rule. We only consider the *INP* case as the other is immediate. We have $P = x(y).P_1$, $\alpha = xz$ and $P' = P_1\{z/y\}$, for some x, y, z, P_1. Then by *ACT*, *MSG*, and *COMP* rules, for some ρ_1, f_1 we have $\rho_1; f_1 \vdash P_1$, $y \notin \rho_1$, $\rho = \rho_1 \cup \{x\}$, and $f_1 = f|\rho_1$. Then $\sigma = \{z/y\}$ is an

injection on ρ_1, and by Lemma 3.2 it follows that $\sigma(\rho_1); f_1\sigma \vdash P_1\sigma$. But since $y \notin \rho_1$, we have $\sigma(\rho_1) = \rho_1$, $f_1\sigma = f_1$. Thus, we have $\rho_1; f_1 \vdash P'$, and the theorem follows from the fact that $\rho_1 \subset \rho$.

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

1. *PAR*: Then $P = P_1|P_2$, $P' = P'_1|P_2$ and the last derivation step is

$$PAR: \quad \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1|P_2 \xrightarrow{\alpha} P'_1|P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset$$

   By *COMP* rule, for some ρ_1, ρ_2, f_1, f_2, we have $\rho_1; f_1 \vdash P_1$, $\rho_2; f_2 \vdash P_2$, $\rho_1 \cap \rho_2 = \emptyset$, $\rho = \rho_1 \cup \rho_2$, and $f = f_1 \oplus f_2$. By induction hypothesis, we have $\rho'_1; f'_1 \vdash P'_1$ for some $\rho'_1 \subset \rho_1 \cup bn(\alpha)$, and f'_1. By Lemma 3.1, $\rho_2 \subset fn(P_2)$. From this and the given fact that $bn(\alpha) \cap fn(P_2) = \emptyset$ we conclude $\rho'_1 \cap \rho_2 = \emptyset$. Then by *COMP* rule $\rho'_1 \cup \rho_2; f'_1 \oplus f_2 \vdash P'_1|P_2$. The theorem follows from the fact that $\rho' = \rho'_1 \cup \rho_2 \subset \rho_1 \cup bn(\alpha) \cup \rho_2 = \rho \cup bn(\alpha)$.

2. *COM*: Then $P = P_1|P_2$, $P' = P'_1|P'_2$, $\alpha = \tau$ and the last derivation step is

$$COM: \quad \frac{P_1 \xrightarrow{\overline{x}y} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1|P_2 \xrightarrow{\tau} P'_1|P'_2}$$

   By *COMP* rule, for some ρ_1, ρ_2, f_1, f_2, we have $\rho_1; f_1 \vdash P_1$, $\rho_2; f_2 \vdash P_2$, $\rho_1 \cap \rho_2 = \emptyset$, $\rho = \rho_1 \cup \rho_2$, and $f = f_1 \oplus f_2$. By induction hypothesis, we have $\rho'_1; f'_1 \vdash P'_1$ for some $\rho'_1 \subset \rho_1 and f'_1$, and $\rho'_2; f'_2 \vdash P'_2$ for some $\rho'_2, \subset \rho_2$ and f'_2. Therefore $\rho'_1 \cap \rho'_2 = \emptyset$, and by the *COMP* rule $\rho'_1 \cup \rho'_2; f'_1 \oplus f'_2 \vdash P'_1|P'_2$. The theorem follows from the fact that $\rho' = \rho'_1 \cup \rho'_2 \subset \rho_1 \cup \rho_2 = \rho$.

3. *RES*: Then $P = (\nu y)P_1$ and $P' = (\nu y)P_1'$ and the last derivation step is

$$\textit{RES:}\quad \frac{P_1 \xrightarrow{\alpha} P_1'}{(\nu y)P_1 \xrightarrow{\alpha} (\nu y)P_1'}\quad y \notin n(\alpha)$$

By *HIDE* rule, for some ρ_1, f_1, we have $\rho_1; f_1 \vdash P_1$, $\rho = \rho_1 - \{y\}$, and $f = f_1|\rho$. By induction hypothesis, $\rho_1'; f_1' \vdash P_1'$ for some $\rho_1' \subset \rho_1 \cup bn(\alpha)$, f_1'. Then by *HIDE* rule $\rho_1' - \{y\}; f_1'|(\rho_1' - \{y\}) \vdash (\nu y)P_1'$. Since $y \notin n(\alpha)$, we have $\rho_1' - \{y\} \subset (\rho_1 - \{y\}) \cup bn(\alpha) = \rho \cup bn(\alpha)$, and the theorem follows.

4. *OPEN*: Then $P = (\nu y)P_1$, $P' = P_1'$, $\alpha = \overline{x}(y)$ and the last derivation step is

$$\textit{OPEN:}\quad \frac{P_1 \xrightarrow{\overline{x}y} P_1'}{(\nu y)P_1 \xrightarrow{\overline{x}(y)} P_1'}\quad x \neq y$$

By *HIDE* rule, for some ρ_1, f_1, we have $\rho_1; f_1 \vdash P_1$, $\rho = \rho_1 - \{y\}$, and $f = f_1|\rho$. By induction hypothesis, $\rho_1'; f_1' \vdash P_1'$ for some $\rho_1' \subset \rho_1 \cup \{y\}$, f_1'. The theorem follows from the fact that $\rho_1 \cup \{y\} \subset \rho \cup \{y\}$.

5. *CLOSE*: Then $P = P_1|P_2$, $P' = (\nu y)(P_1'|P_2')$, $\alpha = \tau$ and the last derivation step is

$$\textit{CLOSE:}\quad \frac{P_1 \xrightarrow{\overline{x}(y)} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} (\nu y)(P_1'|P_2')}\quad y \notin fn(P_2)$$

By *COMP* rule, for some ρ_1, ρ_2, f_1, f_2, we have $\rho_1; f_1 \vdash P_1$, $\rho_2; f_2 \vdash P_2$, $\rho_1 \cap \rho_2 = \emptyset$, $\rho = \rho_1 \cup \rho_2$, and $f = f_1 \oplus f_2$. By induction hypothesis, we have $\rho_1'; f_1' \vdash P_1'$ for some $\rho_1' \subset \rho_1 \cup \{y\}$, and f_1', and $\rho_2'; f_2' \vdash P_2'$ for some $\rho_2' \subset \rho_2$, and f_2'. By Lemma 3.1, $\rho_2 \subset fn(P_2)$. From this and the given fact that $y \notin fn(P_2) = \emptyset$ we conclude $y \notin \rho_2$ and hence $\rho_1' \cap \rho_2 = \emptyset$. Then by *COMP* rule $\rho_1' \cup \rho_2; f_1' \oplus f_2 \vdash P_1'|P_2$. Then by *HIDE* rule $(\rho_1' \cup \rho_2) - \{y\}; (f_1' \oplus f_2)|((\rho_1' \cup \rho_2) - \{y\}) \vdash (\nu y)(P_1'|P_2)$. Since $y \notin \rho_2$, we have $(\rho_1' \cup \rho_2) - \{y\} = (\rho_1' - \{y\}) \cup \rho_2 \subset \rho_1 \cup \rho_2 = \rho$, and the theorem follows.

6. *BRNCH:* Then $P = \texttt{case}\ x\ \textbf{of}\ (y_1 : P_1, \ldots, y_n : P_n)$, and $P' = P_i$ which clearly is well-typed.

7. *BEHV:* Then $P = B\langle\tilde{u};\tilde{v}\rangle, P' = P_1'$ and the last derivation step is

$$\textit{BEHV:}\quad \frac{(x_1(z).P_1)\{(\tilde{u},\tilde{v})/(\tilde{x},\tilde{y})\} \xrightarrow{\alpha} P_1'}{B\langle\tilde{u};\tilde{v}\rangle \xrightarrow{\alpha} P_1'} \quad B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P_1$$

   By *INST* rule $\{\tilde{u}\}; ch(\tilde{u}) \vdash B\langle\tilde{u};\tilde{v}\rangle$ and names in $\tilde{u}$ are distinct. Since the definition of B is well-typed, we have $\{\tilde{x}\}; ch(\tilde{x}) \vdash x_1(z).P_1$. Since names in $\tilde{u}$ are distinct we have $\sigma = \{(\tilde{u},\tilde{v})/(\tilde{x},\tilde{y})\}$ is an injection on $\{\tilde{x}\}$. Then by Lemma 3.2 $\{\tilde{u}\}; ch(\tilde{u}) \vdash (x_1(z).P_1)\sigma$. Then by induction hypothesis $\rho'; f' \vdash P_1'$ for some $\rho' \subset \{\tilde{u}\} \cup bn(\alpha)$, and the theorem follows. □

Since well-typed terms are closed under transitions, 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.

An Example: Polyadic Communication

We show how the ability to temporarily assume a fresh name can be used to encode polyadic communication in Aπ. 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

$$\begin{aligned}
[\![\overline{x}\langle y_1,\ldots,y_n\rangle]\!] &= (\nu u)(\overline{x}u \mid S_1\langle u; y_1,\ldots,y_n\rangle) \\
S_i &\stackrel{def}{=} (u; y_i,\ldots,y_n)u(z).(\overline{z}y_i \mid S_{i+1}\langle u; y_{i+1},\ldots,y_n\rangle) \qquad 1 \le i < n
\end{aligned}$$

$$
\begin{array}{rcll}
S_n & \stackrel{def}{=} & (u; y_n)u(z).\overline{z}y_n & \\
[\![x(y_1, \ldots, y_n).P]\!] & = & x(u).(\nu z)(\overline{u}z \mid R_1\langle z, \hat{x}; u, \tilde{a}\rangle) & \\
R_i & \stackrel{def}{=} & (z, \hat{x}; u, \tilde{a})z(y_i).(\overline{u}z \mid R_{i+1}\langle z, \hat{x}; u, \tilde{a}\rangle) & 1 \leq i < n \\
R_n & \stackrel{def}{=} & (z, \hat{x}; u, \tilde{a})z(y_n).(\overline{u}z \mid [\![P]\!]) &
\end{array}
$$

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

3.3 Alternate Characterization of May Testing

As in any typed calculus, may testing in Aπ takes typing into account; an observer O can be used to test P only if $P|O$ is well typed. For a configuration P, define $rcp(P) = \rho$ if $\rho; f \vdash P$ for some f. Then $P|O$ is well-typed precisely when $rcp(P) \cap rcp(O) = \emptyset$. Thus, Definition 2.4 for $\stackrel{\sqsubset}{\sim}_\rho$ carries over to Aπ except that $P \stackrel{\sqsubset}{\sim}_\rho Q$ is defined only if $rcp(P), rcp(Q) \subset \rho$.

The alternate characterization of may testing for Aπ turns out to be the same as that for L$\pi_=$. This shows that of all the constraints enforced by the type system, only locality "weakens" the observer set. We modify the argument is Section 2.2 to establish the alternate characterization. All the definitions in Section 2.2 apply unchanged, unless mentioned otherwise. The relations $\ll_\rho$ on configurations is the same as in Definition 2.6, except that $P \ll_\rho Q$ is defined only if $rcp(P), rcp(Q) \subset \rho$. Lemma 2.1 holds with essentially the same proof. However, the canonical observer construction has to be changed since the observers defined for L$\pi_=$ need not be well-typed in Aπ.

Definition 3.3 (canonical observer) *For a well-formed trace s, we define an observer*

$$O(s) = (\nu\tilde{x}, z)(|_{y_i \in \chi} Proxy(s, y_i, z) \mid O'(s, z)), \text{ where } z \text{ fresh}$$

$\{\tilde{x}\}$ = set of names occurring as argument of bound input actions in s

χ = set of names occuring as subject of output actions in s

$O'(\epsilon, z) \triangleq \overline{\mu}\mu$

$O'((\hat{v})uv.s, z) \triangleq \overline{u}v|O'(s, z)$

$O'(\overline{u}v.s, z) \triangleq z(w_1, w_2).\texttt{case } w_1 \texttt{ of } (u : \texttt{case } w_2 \texttt{ of } (v : O'(s, z))) \quad w_1, w_2 \textit{ fresh}$

$O'(\overline{u}(v).s, z) \triangleq z(w, v).\texttt{case } w \texttt{ of } (u : O'(s, z)) \quad w \textit{ fresh}$

$Proxy(\epsilon, y, z) \triangleq 0$

$Proxy((\hat{v})uv.s, y, z) \triangleq Proxy(s, y, z)$

$$Proxy((\hat{v})\overline{u}v.s, y, z) \triangleq \begin{cases} y(w).(\overline{z}\langle y, w\rangle \mid Proxy(s, y, z)) & w \textit{ fresh} \quad \textit{if } u = y \\ Proxy(s, y, z) & \textit{otherwise} \end{cases}$$

In the above, $\triangleq$ *is used for macro definitions. The reader may verify that* $\chi - \{\tilde{x}\}; f \vdash O(s)$ *where* f *maps every name in its domain to* $\perp$*. Further, if* s *is* ρ*-well-formed we have* $rcp(O(s)) \cap \rho = \emptyset$*, because the set of names occurring as subject of output actions in a* ρ*-well-formed trace is disjoint from* ρ*.* □

The observer $O(s)$ consists of a collection of proxies and a central matcher. There is one forwarding proxy for each external name a configuration sends a message to while exhibiting s. The proxies forward messages to the matcher which analyzes the contents. This forwarding mechanism, which is absent in the canonical observers for L$\pi_=$, is essential for Aπ because of uniqueness of actor names. Note that the forwarding mechanism uses polyadic communication, whose encoding was shown in Section 3.2.2.

These differences in the canonical observer construction necessitate a fresh proof of Lemma 2.3. We extend the trace preorder $\prec$ defined in Table 2.2 with the following laws for commutativity of outputs.

L5	$s.(\hat{y})\overline{x}y.(\hat{v})\overline{u}v.s' \prec s.(\hat{v})\overline{u}v.(\hat{y})\overline{x}y.s'$	if $\hat{v} \cap \{x, y\} = \emptyset$ and $\hat{y} \cap \{u, v\} = \emptyset$
L6	$s.\overline{x}(z).\overline{y}z.s' \prec s'.\overline{y}(z).\overline{x}z.s$	if $y \neq z$ and $x \neq z$

Note that *L5* and *L6* are symmetric in that if $r \prec s$ by any of these laws, then $s \prec r$. Further, since outputs are asynchronous, the order of consecutive outputs in a trace is insignificant. Hence if $r \prec s$ by these laws and $P \stackrel{s}{\Longrightarrow}$, then $P \stackrel{r}{\Longrightarrow}$. Let $\preceq_|$ be the reflexive

transitive closure of this extended relation $\prec$. Note that since inputs are synchronous in our transition system, the parallel of Lemma 2.1 need not hold for $\preceq_|$.

Lemma 3.3 *For a well-formed trace s, $O(s) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$ implies $r \preceq_| s$.* □

Proof: The proof is by induction on length of s. For the base case, we have $s = \epsilon$. We have $O(\epsilon) \equiv \overline{\mu}\mu$. Hence $r = \epsilon$, and the lemma trivially holds. For the induction step, there are three cases:

1. $\mathbf{s = \overline{x}y.s'}$: Note that $O(\overline{x}y.s')$ first waits for a message $\overline{x}y$ before generating an event or sending any messages. From this observation, it follows that $\overline{r}$ is of form $(\hat{v_1})u_1v_1 \ldots (\hat{v_n})u_nv_n.xy.\overline{r_0}$, where $y \notin \cup_i \hat{v_i}$. Then we also have

$$O(s) \stackrel{xy}{\Longrightarrow} O(s') \stackrel{\overline{r'}}{\Longrightarrow}$$

   where $\overline{r'} = (\hat{v_1})u_1v_1 \ldots (\hat{v_n})u_nv_n.\overline{r_0}$. By induction hypothesis, $r' \preceq_| s'$. Then $\overline{x}y.r' \preceq_| \overline{x}y.s'$. By repeated application of **L5** we deduce $r \preceq_| \overline{x}y.r'$. The result follows from transitivity of $\preceq_|$.

2. $\mathbf{s = \overline{x}(y).s'}$: Note that $O(\overline{x}(y).s')$ first waits for a message $\overline{x}w$ for some w before generating an event or sending any messages. It follows that $\overline{r}$ is of form $(\hat{v_1})u_1v_1 \ldots (\hat{v_n})u_nv_n.(\hat{w})xw.\overline{r_0}$. We only consider the case where $\hat{w} = \emptyset$ and $w \notin \cup_{1 \leq n} \hat{v_i}$, i.e. w is received as free input; the other cases are simpler. Then we have

$$O(s) \stackrel{xw}{\Longrightarrow} O(s'\{w/y\}) \stackrel{\overline{r'}}{\Longrightarrow}$$

   where $\overline{r'} = (\hat{v_1})u_1v_1 \ldots (\hat{v_n})u_nv_n.\overline{r_0}$. By induction hypothesis, $r' \preceq_| s'\{w/y\}$. Then $\overline{x}w.r' \preceq_| \overline{x}w.(s'\{w/y\})$. By *L4*, $\overline{x}w.(s'\{w/y\}) \preceq_| s$. By repeated application of **L5** we deduce $r \preceq_| \overline{x}w.r'$. The result follows from transitivity of $\preceq_|$. Note that the case where $\hat{w} = \emptyset$ and $w \in \cup_{1 \leq n} \hat{v_i}$ would involve *L6*.

3. $\mathbf{s = xy.s'}$: It is easy to show

$$O(s) \equiv \overline{x}y \mid O(s')$$

There are three possible cases depending on whether $\overline{x}y$ fires, does not fire, or is consumed by $O(s')$. We consider only the first and the last case.

- $\overline{x}y$ fires. It follows that $\overline{r} = \overline{r_1}.\overline{x}y.\overline{r_2}$, where $y \notin bn(r_1)$. Then it is the case that

$$O(s') \stackrel{\overline{r_1.r_2}.\overline{\mu}\mu}{\Longrightarrow}$$

  By induction hypothesis, $r_1.r_2 \preceq_| s'$. Then $xy.r_1.r_2 \preceq_| xy.s'$. By repeated application of *L2*, we have $r \preceq_| xy.r_1.r_2$. The result follows by transitivity of $\preceq_|$.

- $\overline{x}y$ is consumed internally. Then it is the case that

$$O(s') \stackrel{\overline{r_1}.xy.\overline{r_2}.\overline{\mu}\mu}{\Longrightarrow}$$

  and $r = r_1.r_2$. By induction hypothesis $r_1.\overline{x}y.r_2 \preceq_| s'$. Then $xy.r_1.\overline{x}y.r_2 \preceq_| xy.s'$. By *L2* and *L3* we have $r = r_1.r_2 \preceq_| xy.r_1.\overline{x}y.r_2$, and the result follows by transitivity of $\preceq_|$.

4. $\mathbf{s} = \mathbf{x}(\mathbf{y}).\mathbf{s}'$: It is easy to show

$$O(s) \equiv (\nu y)(\overline{x}y \mid O(s'))$$

There are three possible cases depending on whether $\overline{x}y$ fires, does not fire, or is internally consumed. We consider only the case where it does not fire. Since $\overline{x}y$ never fires, all the interactions are performed by $(\nu y)(O(s'))$, that is

$$(\nu y)O(s') \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$$

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 $\overline{r} = \overline{r_1}.\overline{u}(w).\overline{r_2}$ where $w \notin n(r_1)$.

Then we can deduce

$$O(s'\{w/y\}) \stackrel{\overline{r'}.\overline{\mu}\mu}{\Longrightarrow}$$

where $\overline{r'} = \overline{r_1}.\overline{u}w.\overline{r_2}$. By induction hypothesis, $r' \preceq_{|} s'\{w/y\}$. Then $x(w).r' \preceq_{|} x(w).s'\{w/y\}$, and by *L1*, we have $r \preceq_{|} x(w).r'$. The result follows by transitivity of $\preceq_{|}$, and that s is alpha equivalent to $x(w).s'\{w/y\}$. □

Lemma 3.4 *Let $r \preceq_{|} s$ and $P \stackrel{r}{\Longrightarrow}$. Then there is $r' \preceq s$ such that $P \stackrel{r'}{\Longrightarrow}$.*

Proof: We only sketch the proof. Let $r = r_n \prec \ldots r_1 \prec s$. We skip the application of *L5* and *L6* in the chain above, and apply the other transformation to r in the same order. This is possible because *L5* and *L6* merely reorder consecutive outputs, and hence skipping an instance of these does not preclude the other transformations that follow. However, note that a following instance of *L3* might require additional applications of *L2* first. With this procedure we get a trace r' that differs from r in only the order of consecutive outputs, and $r' \preceq s$. Then since $P \stackrel{r}{\Longrightarrow}$, we have $P \stackrel{r'}{\Longrightarrow}$. □

Lemma 2.3 holds in Aπ as direct consequence of Lemmas 3.3 and 3.4. Now, Theorem 2.3, which establishes the alternate characterization, holds in Aπ with formally the same proof.

3.4 Variants of Aπ

We first consider a variant of Aπ, called A$\pi_{\neq}$, which is equipped with the ability to mismatch names. We use an if-then-else construct, and replace the *CASE* rule of Table 3.1 with the following.

$$\textit{COND:}\ \frac{\rho_1; f_1 \vdash C_1 \quad \rho_2; f_2 \vdash C_2}{\rho_1 \cup \rho_2; f_1 \oplus f_2 \vdash [x = y](C_1, C_2)}\ \texttt{if } f_1 \texttt{ and } f_2 \texttt{ are compatible}$$

For the operational semantics of A$\pi_{\neq}$, the *BRNCH* rule of Table 3.2, is replaced with the following

$$\textit{IF:}\ \frac{P_1 \xrightarrow{\alpha} P_1'}{[x=y](P_1,P_2) \xrightarrow{\alpha} P_1'}\ x=y \qquad \textit{ELSE:}\ \frac{P_2 \xrightarrow{\alpha} P_2'}{[x=y](P_1,P_2) \xrightarrow{\alpha} P_2'}\ x\neq y$$

The alternate characterization of may-testing for A$\pi_{\neq}$ is tighter than that for Aπ since laws *L3* and *L4* are not required. *L4* disappears because with mismatch capability, an observer can discriminate between free and bound inputs. Further, since all receptionist names of a configuration are treated as being private to the configuration, complementary actions cannot be exhibited by a configuration. This, together with the absence of *L4* makes *L3* unnecessary (see Lemma 3.5). To sum up, the interpretation of the may preorder $P \stackrel{\sqsubset}{\sim}_\rho Q$ now becomes: by consuming the same messages, Q can produce at least the same messages as P.

Let $\preceq^-$ be the reflexive transitive closure of laws *L1* and *L2* of Table 2.2. With minor modifications to the proof, Lemma 2.1 holds for A$\pi_{\neq}$ with $\preceq^-$ instead of $\preceq$. The relation $\ll_\rho$ is same as for Aπ except that $\preceq^-$ is used instead of $\preceq$. The canonical observer for Aπ is modified to check for bound inputs using the ability to mismatch names.

Definition 3.4 (canonical observer) *For a well-formed trace s and a set of names ρ, we define*

$$O(s,\rho) = (\nu\tilde{x}, z)(|_{x_i\in\chi} Proxy(s,x_i,z) \mid O'(s,\rho,z)),$$

where

$$O'(\epsilon,\rho,z) \stackrel{\triangle}{=} \overline{\mu}\mu$$
$$O'((\hat{v})uv.s,\rho,z) \stackrel{\triangle}{=} \overline{u}v|O'(s,\rho\cup\hat{v},z)$$
$$O'(\overline{u}v.s,\rho,z) \stackrel{\triangle}{=} z(w_1,w_2).[w_1=u\wedge w_2=v](O'(s,\rho,z),0) \qquad w_1,w_2\ fresh$$
$$O'(\overline{u}(v).s,\rho,z) \stackrel{\triangle}{=} z(w,v).[w=u\wedge v\notin\rho](O'(s,\rho\cup\{v\},z),0) \qquad w\ fresh$$

where $\tilde{x},\chi$ and Proxy are as defined in Definition 3.3. The reader may verify that $\chi-\{\tilde{x}\}; f \vdash O(s,\rho)$ where f maps every name in its domain to $\perp$. □

The canonical observer construction takes as an argument a set of names ρ which is presumed to contain all the free names in the configuration to be observed. Note how this set is updated as the interaction proceeds. To verify that an input it receives is bound, the observer checks if the input argument does not occur in ρ. This works because, if

$P \xrightarrow{(\hat{y})\overline{x}y}$ then $\hat{y} \neq \emptyset$ if and only if $y \notin fn(P)$. The operations $\wedge$ and $\notin$ can be encoded using the conditional construct. Note that $\notin$ uses the ability to take an action when a match fails.

Let $\preceq_{|}^{-}$ be the reflexive transitive closure of laws *L1, L2, L5, L6*. Following is the analogue of Lemma 3.3. Note that laws *L3* and *L4* are not used in the proof.

Lemma 3.5 *For some ρ', let s be a ρ'-well-formed trace such that $s = s_1.(\hat{y})xy.s_2$ implies $x \in rcp(s_1, \rho')$. Further, let r be a trace such that $P \stackrel{r}{\Longrightarrow}$ for some P such that $fn(P) \subset \rho$. Then $O(s, \rho) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$ implies $r \preceq_{|}^{-} s$.*

Remark: The condition on s simply says that s cannot exhibit a pair of complementary actions, i.e. an input and an output action with the same subject.

Proof: The proof is by induction on the length of s, and is similar to that of Lemma 3.3, which we modify. We only sketch the changes to cases 2,3, and 4.

1. $\mathbf{s} = \overline{\mathbf{x}}(\mathbf{y}).\mathbf{s'}$: We deduce $\overline{r}$ is of form $(\hat{v_1})u_1v_1 \ldots (\hat{v_n})u_nv_n.(\hat{w})xw.\overline{r_0}$, where $w \notin \rho$. We note that if $P \xrightarrow{(\hat{z})\overline{x}z}$, then $z \in fn(P) \cup \hat{z}$. Now, since $w \notin fn(P)$ and $P \stackrel{r}{\Longrightarrow}$, it follows that $w \in \cup_{1 \leq n} \hat{v_i} \cup \hat{w}$. Then we have

$$O(s, \rho) \stackrel{x(w)}{\Longrightarrow} O(s'\{w/y\}, \rho \cup \{w\}) \stackrel{\overline{r'}}{\Longrightarrow}$$

   where $\overline{r'} = (\hat{v_1'})u_1v_1 \ldots (\hat{v_n'})u_nv_n.\overline{r_0}$, and $\hat{v_i'} = \hat{v_i} - \{w\}$. Since $O(s, \rho) \stackrel{x(w)}{\Longrightarrow}$, we have $w \notin fn(O(s, \rho))$, which in turn implies $w \notin fn(s)$. It follows that $s'\{w/y\}$ is $\rho' \cup \{w\}$-well-formed. It is routine to verify the remaining premises for the induction step. Then by induction hypothesis, $r' \preceq_{|}^{-} s'\{w/y\}$. Then $\overline{x}(w).r' \preceq_{|}^{-} \overline{x}(w).(s'\{w/y\})$. By repeated application of *L5* and *L6* we deduce $r \preceq_{|}^{-} \overline{x}(w).r'$. The result follows from transitivity of $\preceq_{|}^{-}$, and the fact that we work modulo alpha equivalence on traces.

2. $\mathbf{s} = (\hat{\mathbf{y}})\mathbf{xy}.\mathbf{s'}$: We have

$$O(s, \rho) \equiv (\nu \hat{y})(\overline{x}y \mid O(s', \rho \cup \hat{y}))$$

By the condition on s, $x \in \rho'$. Then since s is ρ'-well-formed, x cannot be the target of an output in s'. It follows that $x \notin rcp(O(s', \rho \cup \hat{y}))$. This precludes the possibility of the message $\overline{x}y$ being consumed internally in cases 3 and 4 in the proof of Lemma 3.3. The argument for the other possibilities is very similar. □

Lemma 3.4 holds for A$\pi_{\neq}$. Theorem 2.3 holds with simple modifications to the proof, of which we present the modifications to the only-if part.

Theorem 3.2 *$P \stackrel{\sqsubset}{\sim}_\rho Q$ if and only if $P \ll_\rho Q$.*

Proof: (**only if**) Let $P \stackrel{\sqsubset}{\sim}_\rho Q$ and $P \stackrel{s}{\Longrightarrow}$ where s is ρ-well-formed. We have to show that there is a trace $r \preceq^- s$ such that $Q \stackrel{r}{\Longrightarrow}$. Since we work modulo alpha equivalence on traces, we can assume $bn(s) \cap fn(Q) = \emptyset$. Then it is easy to show that $O(s, fn(Q)) \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$. This can be zipped with $P \stackrel{s}{\Longrightarrow}$ to get $P|O(s, fn(Q)) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, that is $P \underline{may}\ O(s, fn(Q))$. From $P \stackrel{\sqsubset}{\sim}_\rho Q$, we have $Q \underline{may}\ O(s, fn(Q))$ and therefore $Q|O(s, fn(Q)) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. This can be unzipped into $Q \stackrel{r'}{\Longrightarrow}$ and $O(s, fn(Q)) \stackrel{\overline{r'}.\overline{\mu}\mu}{\Longrightarrow}$. By definition of $\stackrel{\sqsubset}{\sim}_\rho$, $rcp(P) \subset \rho$. Using this it is easy to show that if $s = s_1.(\hat{y})xy.s_2$ then $x \in rcp(s_1, \rho)$. Then from Lemma 3.5, it follows that $r' \preceq^-_| s$. From Lemma 3.4, there is $r \preceq^- s$ such that $Q \stackrel{r}{\Longrightarrow}$. □

We could consider a variant of Aπ with restricted matching capability along the lines described in Section 2.4. Note that since the tested names in a `case` construct are constants, i.e. are either free or bound by a restriction, it is possible to do away with non-determinism in actor behaviors by requiring that the names are all distinct. Proofs in Section 2.3 can be adapted to show that for this variant, the alternate characterization of may testing is the same as that of L$\pi^-_=$.

3.5 Discussion and Related Work

There has been considerable research on actor semantics in the past two decades. We set our contribution in the context of the most prominent works. A significant fraction of the research has been in formal semantics for high level concurrent programming languages based on the Actor Model. The prominent ones are [7, 29], where a core functional

language is extended with actor coordination primitives. The main aim of these works has been to design concurrent languages that could be useful in practice. Accordingly, the languages assume high-level computational notions as primitives, and are embellished with type systems that guarantee useful properties in object based settings. In contrast, Aπ is a basic calculus that makes only the ontological commitments inherent in the Actor Model, thus giving us a simpler framework for further theoretical investigations. The simplicity also makes it feasible to demarcate the effect of each ontological commitment on the theory. In Chapter 4, we show how Aπ can be used to give a translational semantics for a simple actor based programming language.

In [88, 89], actors are modeled in rewriting logic which is widely acknowledged as a universal model of concurrency [53, 59]. An actor system is modeled as a specific rewrite theory, and established techniques are used to derive the semantics of the specification and prove its properties. In a larger context, this effort belongs to a collection of works that have demonstrated that rewriting logic provides a good basis to unify many different concurrency theories. For example, there are also rewrite theory formulations of the π-calculus [98]. In comparison, although our work is more limited in scope, it establishes a connection between two popular models of concurrency that is deeper than is immediately available from representing the two models in a unified basis. It can be seen as a more elaborate investigation of the relationship between two specific rewrite theories, and provides a formal connection that helps in adapting and transferring results in one theory to the other.

There are several calculi that are inspired by the Actor Model and the π-calculus [32, 43, 76]. But these are neither entirely faithful to the Actor Model, nor directly comparable to π-calculus. For instance, they are equipped with primitives intrinsic to neither of the models, or ignore actor properties such as uniqueness and persistence. These works are primarily intended for investigation of object oriented concepts. An exception is [32] whose purpose is an algebraic formulation of actors, but the calculus is not comparable to π-calculus and lacks theoretical investigations beyond the basic formulation.

Semantic models proposed for actor computations include event diagrams [24], interaction diagrams and interaction paths [90]. These models represent possible interactions of an actor system at different levels of abstraction, with event diagrams the most concrete, and interaction paths the most abstract. The relation between these three domains is studied in [90], where each domain is given an algebraic structure, and maps between the domains are defined that are shown to preserve the algebraic structure. Such homomorphisms allows one to relate specifications based on different models and transfer results obtained in one domain to the other. We have shown that (a simple variant of) the interaction paths model provides an alternate characterization for may-testing in actor systems.

Chapter 4

A Simple Actor Language

Aπ can serve as the basis for actor based concurrent programming languages. As an illustration, we give a formal semantics for a simple actor language (SAL) by translating its programs into Aπ. The translation can be exploited to apply the characterizations established in Chapter 3 to reason about programs in SAL. To keep the discussion simple, we have considered a simple language, which nevertheless can serve as the core of a richer language.

In Sections 4.1 and 4.2, we show how booleans, natural numbers and operations on them can be represented as processes in Aπ. These data types, along with names, are assumed as primitive in SAL. Of course, this exercise is not entirely necessary, and in fact, a better strategy may be to directly consider an extended version of Aπ with basic data types. The characterizations for Aπ can be adapted in a straightforward manner to the extended calculus. We have chosen the other approach here, mainly to illustrate that the type system of Aπ does not reduce the expressive power. In Section 4.3, we describe SAL informally. In Section 4.4, we give the translational semantics for SAL. We conclude the chapter in Section 4.5, with a discussion of related work on semantics for actor based languages.

4.1 Booleans

Booleans are encoded as configurations with a single actor that is also a receptionist. In the following, $\underline{T}$ defines the receptionist behavior for *true*, and $\underline{F}$ for *false*.

$$\underline{T} \stackrel{def}{=} (x)x(u,v,c).\overline{c}u$$

$$\underline{F} \stackrel{def}{=} (x)x(u,v,c).\overline{c}v$$

The behaviors accept messages containing three names, of which the third name is assumed to be the customer name (see Section 3.2.2 for an encoding of polyadic communication). The behavior $\underline{T}$ replies back to the customer with the first name, while $\underline{F}$ replies back with the second name.

The negation function can be encoded as follows

$$Not \stackrel{def}{=} (x)x(u,c).(\nu v,y,z)(\overline{u}\langle y,z,v\rangle \mid v(w).\texttt{case}\ w\ \texttt{of}(y : \underline{F}\langle v\rangle, z : \underline{T}\langle v\rangle))$$

$Not(x)$ can be thought of as the function *not* available at name x. Evaluation of the function is initiated by sending a message containing a value and a customer name, to x. The customer eventually receives the negation of the value sent. The reader may verify that

$$Not\langle x\rangle \mid \underline{F}\langle u\rangle \mid \overline{x}\langle u,c\rangle \stackrel{\overline{c}(v)}{\Longrightarrow} \underline{T}\langle v\rangle$$

Following is the encoding of boolean *and*

$$\begin{aligned} And \stackrel{def}{=} (x)x(u,v,c).(\nu y,z_1,z_2)(&\overline{u}\langle z_1,z_2,y\rangle \mid \overline{v}\langle z_1,z_2,y\rangle \mid \\ &y(w_1).y(w_2).(\overline{c}y \mid \\ &\qquad \texttt{case}\ w_1\ \texttt{of}\ (\\ &\qquad\quad z_1 : \texttt{case}\ w_2\ \texttt{of}\ (z_1 : \underline{T}\langle y\rangle, z_2 : \underline{F}\langle y\rangle), \\ &\qquad\quad z_2 : \underline{F}\langle y\rangle\ ))) \end{aligned}$$

The reader may verify the following

$$And\langle x\rangle \mid \underline{T}\langle u\rangle \mid \underline{F}\langle v\rangle \mid \overline{x}\langle u,v,c\rangle \stackrel{\overline{c}\langle y\rangle}{\Longrightarrow} \underline{F}\langle y\rangle$$

The reader may also verify that for each behavior B defined above $\{x\};\{x \mapsto \bot\} \vdash B\langle x\rangle$.

4.2 Natural Numbers

Natural numbers are built from the constructors 0 and S. Accordingly, we define the following two behaviors.

$$\begin{aligned} Zero &\stackrel{def}{=} (x)x(u,v,c).\overline{c}\langle u,x\rangle \\ Succ &\stackrel{def}{=} (x,y)x(u,v,c).\overline{c}\langle v,y\rangle \end{aligned}$$

With this, natural numbers can be encoded as follows.

$$\begin{aligned} \underline{0}(x) &\stackrel{\triangle}{=} Zero\langle x\rangle \\ \underline{S^{n+1}0}(x) &\stackrel{\triangle}{=} (\nu y)(Succ\langle x,y\rangle \mid \underline{S^n 0}(y)) \end{aligned}$$

The number $S^n 0$ is encoded as a sequence of $n+1$ actors each pointing to the next, and the last one pointing to itself. The first n actors have the behavior $Succ$ and the last has behavior $Zero$. Only the first actor is the receptionist to the entire configuration. As in our encoding for booleans, both the behaviors accept messages with three names, the last of which is assumed to denote the customer. The behavior $Succ$ replies back to the customer with the second name and the name of next actor in the sequence, while $Zero$ replies back with the first name and its own name.

We only show the encoding of the addition operation, and hope the reader is convinced that it is possible to encode the others. Our aim is to define a behavior Add such that

$$Add\langle x\rangle \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u,v,c\rangle \stackrel{\overline{c}\langle w\rangle}{\Longrightarrow} \underline{S^{n+m}0}(w)$$

We first define a behavior $AddTo$ such that

$$AddTo\langle x\rangle \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^m 0}(v) \Longrightarrow \quad (\nu u)(\underline{S^{n+m} 0}(u) \mid \overline{c}u)$$

We will then use $AddTo$ to define Add.

$$\begin{aligned} AddTo \stackrel{def}{=} (x)x(u_1, u_2, c).(\nu y_1, y_2, w)(\overline{u_2}\langle y_1, y_2, w\rangle \mid \\ w(z_1, z_2).\texttt{case } z_1 \texttt{ of } (\\ y_1 : \overline{c}u_1, \\ y_2 : (\nu v)(Succ\langle v, u_1\rangle \mid \overline{x}\langle v, z_2, c\rangle \mid AddTo\langle x\rangle))) \end{aligned}$$

Lemma 4.1 $AddTo\langle x\rangle \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^m 0}(v) \Longrightarrow (\nu u)(\underline{S^{n+m} 0}(u) \mid \overline{c}u)$

Proof: We prove this by induction on m. For the base case, the reader may verify that

$$AddTo\langle x\rangle \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{0}(v) \Longrightarrow (\nu u)(\underline{S^n 0}(u) \mid \overline{c}u)$$

For the induction step, assuming the given proposition we show

$$AddTo\langle x\rangle \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^{m+1} 0}(v) \Longrightarrow (\nu u)(\underline{S^{n+m+1} 0}(u) \mid \overline{c}u)$$

$$\begin{aligned} & AddTo\langle x\rangle \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^{m+1} 0}(v) \\ & \quad \Longrightarrow AddTo\langle x\rangle \mid (\nu u)(\underline{S^{n+1} 0}(u) \mid \overline{x}\langle u, w, c\rangle) \mid \underline{S^m 0}(w) \\ & \quad \Longrightarrow (\nu u)(\underline{S^{n+m+1} 0}(u) \mid \overline{c}u) \qquad \text{by induction hypothesis} \end{aligned}$$

□

We are now ready to define Add.

$$Add \stackrel{def}{=} (x)x(u, v, c).(\nu y, z, w)(AddTo\langle y\rangle \mid \underline{0}(w) \mid \overline{y}\langle w, u, z\rangle \mid z(w).(\nu y)(AddTo\langle y\rangle \mid \overline{y}\langle w, v, c\rangle))$$

Lemma 4.2 $Add\langle x\rangle \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u, v, c\rangle \stackrel{\overline{c}(w)}{\Longrightarrow} \underline{S^{n+m} 0}(w)$

Proof:

$$\begin{aligned} & Add\langle x\rangle \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u, v, c\rangle \\ & \quad \stackrel{\tau}{\longrightarrow} \qquad (\nu y, z)(AddTo\langle y\rangle \mid (\nu w)(\underline{0}(w) \mid \overline{y}\langle w, u, z\rangle) \mid \underline{S^n 0}(u) \mid \\ & \qquad\qquad z(w).(\nu y)(AddTo\langle y\rangle \mid \overline{y}\langle w, v, c\rangle) \mid \underline{S^m 0}(v)) \end{aligned}$$

$$
\begin{array}{lll}
\Longrightarrow & (\nu y, z)((\nu w)(\underline{S^n 0}(w) \mid \overline{z}w) \mid z(w).(\nu y)(AddTo\langle y\rangle \mid \overline{y}\langle w, v, c\rangle) \mid \underline{S^m 0}(v)) & \\
 & & \text{(by Lemma 4.1)} \\
\xrightarrow{\tau} & (\nu y)(AddTo\langle y\rangle \mid (\nu w)(\underline{S^n 0}(w) \mid \overline{y}\langle w, v, c\rangle) \mid \underline{S^m 0}(v)) & \\
\Longrightarrow & (\nu w)(\underline{S^{n+m} 0}(w) \mid \overline{c}w) & \text{(by Lemma 4.1)} \\
\overset{\overline{c}(w)}{\Longrightarrow} & \underline{S^{n+m} 0}(w) & \square
\end{array}
$$

The reader may verify that for a natural number N, and each behavior B defined above, $\{x\}; \{x \mapsto \bot\} \vdash \underline{N}(x)$, and $\{x\}; \{x \mapsto \bot\} \vdash B\langle x\rangle$.

4.3 The Language SAL

A SAL program consists of a sequence of behavior definitions followed by a single (top level) command.

$$Pgm ::= BDef_1 \ \ ... \ \ BDef_n \ \ Com$$

The behavior definitions are templates for actor behaviors. The top level command creates an initial collection of actors and messages, and specifies the interface of the configuration to its environment.

4.3.1 Expressions

Three types of primitive values - booleans, integers and names - are presumed. There are literals for boolean and integer constants, but none for names. Primitive operations include $\wedge, \vee, \neg$ on booleans, $+, -, *, =$ on integers, and `case-of` for matching names. Expressions always evaluate to values of one of the primitive types. An expression may contain identifiers which may be bound by formal parameters of the behavior definition in which the expression occurs (see behavior definitions in Section 4.3.3). Identifiers are lexically scoped in SAL. We let e range over the syntactic domain of expressions.

4.3.2 Commands

Following is the syntax for SAL commands.

Com ::=	**send** $[e_1, ..., e_n]$ **to** x	**(message send)**
	become $B(e_1, ..., e_n)$	**(new behavior)**
	let $x_1 =$ [**recep**] **new** $B_1(e_1, ..., e_{i_1})$,	
	$\ldots\ x_k =$ [**recep**] **new** $B_k(e_1, ..., e_{i_k})$	
	in *Com*	**(actor creations)**
	if e **then** Com_1 **else** Com_2	**(conditional)**
	case x **of** $(y_1 : Com_1, \ldots, y_n : Com_n)$	**(name matching)**
	$Com_1 \mid\mid Com_2$	**(composition)**

Message Send: Expressions e_1 to e_n are evaluated, and a message containing the resulting tuple of values is sent to actor x. The message is asynchronous as the execution of the command does not involve actual delivery of the message.

New Behavior: This specifies a new behavior for the actor which is executing the command. The identifier B should be bound by a behavior definition. Expressions e_1 to e_n are evaluated and the results are bound to the parameters in the acquaintance list of B (see behavior definitions in Section 4.3.3). The resulting closure is the new behavior of the actor. A `become` command cannot occur in the top level command of an actor program, because such a command specifies the initial system configuration and not the behavior of a single actor.

Actor Creations: Actors with the specified behaviors are created, the identifiers $x_1, \ldots, x_n$, that are all required to be distinct, denote names of the actors, and the command *Com* is executed under the scope of these identifiers. The identifiers can be optionally tagged with the qualifier `recep`, only in the top level command of a program. The corresponding actors will be receptionists of the program configuration, and can thus receive messages from the environment. The other actor names are private to the

configuration. While the scope of the identifiers declared as receptionists is the entire top level command, the scope of the others is only the `let` command. A `let` command that uses the `recep` qualifier should not be nested under another `let` or conditional construct. Further, since actor names are unique, a name can not be declared as a receptionist more than once in the entire top level command.

Conditional: The expression e should evaluate to a boolean. If the result is true, command Com_1 is executed, else Com_2 is executed.

Name matching: The name x is matched against the names $y_1, \ldots, y_n$. If there is a match, the command corresponding to one of the matches is non-deterministically chosen and executed. If there is no match, then there is no further execution of the command.

Composition: The two composed commands are executed concurrently.

A couple of observations are in order here. First, there is no notion of sequential composition of commands. This is because all the actions an actor performs on receiving a message are concurrent. Second, message passing in SAL is analogous to *call-by-value* parameter passing; expressions in a `send` command are first evaluated and a message is created with the resulting values. Alternately, we can think of a *call-by-need* message passing scheme. But both the mechanisms are semantically equivalent because expressions do not involve recursions and hence their evaluations always terminate.

4.3.3 Behavior Definitions

The syntax of behavior definitions is as follows.

$BDef$ ::= **def** $<$ *beh name* $>$ ($<$ *acquaintence list* $>$)[$<$ *input list* $>$]
Com
end def

The identifier $<$ *beh name* $>$ is bound to an abstraction and the scope of this binding is the entire program. The identifiers in acquaintance list are formal parameters of this

abstraction, and their scope is the body *Com*. These parameters are bound during a behavior instantiation, and the resulting closure is an actor behavior. The identifiers in input list are formal parameters of this behavior, and their scope is the body *Com*. They are bound at the receipt of a message. The acquaintance and input lists contain all the free identifiers in *Com*. The reserved identifier *self* can be used in *Com* as a reference to the actor which has the behavior being defined. The execution of *Com* should always result in the execution of at most a single `become` command, else the behavior definition is said to be erroneous. This property is guaranteed statically by requiring that in any concurrent composition of commands, at most one of the commands can contain a *become*. If the execution of *Com* does not result in the execution of a *become* command, then the corresponding actor is assumed to take on a *sink* behavior that simply ignores all the received messages.

4.3.4 An Example

SAL is not equipped with high-level control flow structures such as recursion and iteration. However, such structures can be encoded as patterns of message passing [38]. The following implementation of the factorial function (adapted from [5]) shows how recursion can be encoded. The example also illustrates customer passing style of programming common in actor systems.

def $Factorial\ ()[val, cust]$
  become $Factorial()$ ||
  if $val = 0$
    then send [1] to $cust$
    else let $cont = FactorialCont(val, cust)$
      in send $[val - 1, cont]$ **to** $self$
end def

def $FactorialCont(val, cust)[arg]$

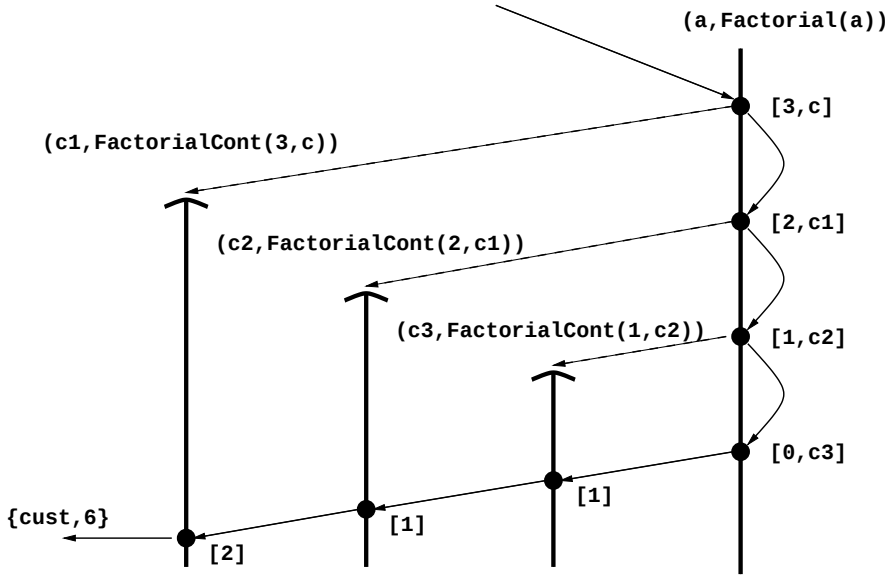


Figure 4.1: A diagram illustrating computation of factorial 3. The vertical lines denote time lines of actors. An arrow to the top of a time line denotes an actor creation. Other arrows denote messages.

send $[val * arg]$ **to** *cust*

end def

A request to a factorial actor includes a positive integer n and the actor name *cust* to which the result has to be sent. On receiving a message the actor creates a continuation actor *cont* and sends itself a message with contents $n - 1$ and *cont*. The continuation actor has n and *cust* as its acquaintances. Eventually a chain of continuation actors will be created each knowing the name of the next in the chain (see Figure 4.1). On receiving a message with an integer, the behavior of each continuation actor is to multiply the integer with the one it remembers and send the reply to its customer. The program can be proved correct by a simple induction on n. Note that the factorial actor can process many requests concurrently.

Following is a top level command that creates a factorial actor that is also a receptionist and sends a message with value 5 to it.

let x = [**recep**] **new** *Factorial*()
in send [5] **to** x

4.4 Formal Semantics of SAL

SAL expressions and commands are represented as Aπ terms, and their evaluation as computation in these terms. SAL behavior definitions are represented as definitions in Aπ. We now describe the translation of each syntactic domain of SAL.

4.4.1 Expressions

While SAL is equipped with booleans, integers and names as primitive types, only names are primitive in Aπ. In Sections 4.1 and 4.2, we discussed encodings of booleans, naturals, and operations on these types. The encoding of naturals can be extended to integers in a fairly straightforward manner (by tagging for example). Now that we have a representation of the basic constituents of expressions, what remains is the representation of dependencies between the evaluation of subexpressions of an expression.

The translation of an expression takes as an argument, the name of a customer to which the result of evaluation is to be sent. An identifier expression x is translated as

$$[\![x]\!]c = \overline{c}x$$

A constant (boolean or integer) expression e is translated as

$$[\![e]\!]c = (\nu y)(\underline{e}\langle y\rangle \mid \overline{c}y)$$

where $\underline{e}$ is the encoding of the constant e. For an $n-$ary operator Op, the expression $Op(e_1, ..., e_n)$ is encoded as

$$[\![Op(e_1, \ldots, e_n)]\!]c = (\nu y_1, \ldots, y_{n+1}, z)(Marshall(y_1, \ldots, y_{n+1}, z) \mid [\![e_1]\!]y_1 \mid \ldots \mid [\![e_n]\!]y_n \mid \overline{y_{n+1}}c \mid \underline{Op}\langle z\rangle)$$

where $\underline{Op}$ is the encoding of operator Op, and z, y_i are fresh. The expressions e_1 to e_n are concurrently evaluated. The configuration $Marshall(y_1, \ldots, y_{n+1}, z)$ marshals their results and the customer name into a single tuple, and forwards it to an internal actor that implements Op. The marshaling configuration is defined as

$$Marshall(y_1, \ldots, y_n, c) = (\nu u)(R\langle y_1, u\rangle \mid \ \ldots \ \mid R\langle y_n, u\rangle \mid S_1\langle u, y_1, \ldots, y_n, c\rangle)$$

where

$$R \stackrel{def}{=} (x, y)\ x(u).\overline{y}\langle u, x\rangle$$

$$\begin{aligned} S_i \stackrel{def}{=} (x, y_i, \ldots, y_n, v_1, \ldots, v_{i-1}, c) \\ x(v_i, w).\texttt{case}\ w\ \texttt{of}\ (\\ y_i : S_{i+1}\langle x, y_{i+1}, \ldots, y_n, v_1, \ldots, v_i, c\rangle \\ y_{i+1} \ldots y_n : S_i\langle x, y_i, \ldots, y_n, v_1, \ldots, v_{i-1}, c\rangle \mid \overline{x}\langle v_i, w\rangle) \\ \text{for } 1 \leq i < n \end{aligned}$$

$$S_n \stackrel{def}{=} (x, v_1, \ldots, v_{n-1}, c)\ x(v_n, w).\overline{c}\langle v_1, \ldots, v_n\rangle$$

By structural induction on an expression e and name x, it is easy to show that $\emptyset; \{\} \vdash [\![e]\!]x$.

4.4.2 Commands

Although the actions an actor performs on receiving a message are all concurrent, execution of commands may involve sequentiality. For example, expressions need to be evaluated before the results are used to send messages or instantiate a behavior. This sequentiality is represented as communication patterns in the configurations that encode commands. The translation of a command takes as an argument, the name of the actor which executes the command. In the following, we assume that the names introduced during the translation are all fresh.

Message send: We use the *Marshall* configuration to marshal the results of expression evaluations into a polyadic message to the target.

$$[\![\textbf{send } [e_1, \ldots, e_n] \textbf{ to } z]\!]x = (\nu y_1, \ldots, y_n)(Marshall(y_1, \ldots, y_n, z) \mid [\![e_1]\!]y_1 \mid \ \ldots \ \mid [\![e_n]\!]y_n)$$

New behavior: We use an actor's ability to temporarily assume a new name to wait for the results of expression evaluations before assuming the new behavior.

$$[\![\textbf{become } B(e_1, \ldots, e_n)]\!]x = (\nu y_1, \ldots, y_n, z)([\![e_1]\!]y_1 \mid \ \ldots \ \mid [\![e_n]\!]y_n \mid Marshall\langle y_1, \ldots, y_n, z\rangle \mid z(u_1, \ldots, u_n).\underline{B}\langle x, u_1, \ldots, u_n\rangle)$$

where $\underline{B}$ is the identifier for the translation of behavior definition B (see Section 4.4.3).

Actor creation: The identifiers used in the `let` command are used as names for the new actors. If not tagged by the `recep` qualifier, these names are bound by a restriction. The actors are created at the beginning of command execution, but they assume a temporary name until their behavior is determined.

$$[\![\textbf{let } \ y_1 = [\textbf{recep}] \textbf{ new } B_1(e_1, \ldots, e_{i_1}),$$
$$\ldots \ y_k = [\textbf{recep}] \textbf{ new } B_k(e_1, \ldots, e_{i_k}) \ \textbf{ in } \ Com]\!]x =$$
$$(\nu \tilde{y})([\![\textbf{become } B_1(e_1, \ldots, e_{i_1})]\!]y_1 \mid \ \ldots \ \mid [\![\textbf{become } B_k(e_1, \ldots, e_{i_k})]\!]y_k \mid [\![Com]\!]x)$$

where $\tilde{y}$ consists of all y_i which have not been qualified with `recep`.

Conditional: We use a temporary actor that waits for the outcome of the test before executing the appropriate command.

$$[\![\textbf{if } e \textbf{ then } Com_1 \textbf{ else } Com_2]\!]x =$$
$$(\nu u)([\![e]\!]u \mid u(z).(\nu v_1, v_2)(\overline{z}\langle v_1, v_2, u\rangle \mid u(w).\texttt{case } w \texttt{ of } (v_1 : [\![Com]\!]x, v_2 : [\![Com]\!]x))$$

Name matching:

$$[\![\textbf{case } z \textbf{ of}(y_1 : Com_1, \ldots, y_n : Com_n)]\!]x = \texttt{case } z \texttt{ of } (y_1 : [\![Com_1]\!]x, \ldots, y_n : [\![Com_n]\!]x)$$

Concurrent Composition: The translation of concurrent composition is just the composition of individual translations.

$$[\![Com_1 \mid\mid Com_2]\!]x = [\![Com_1]\!]x \mid [\![Com_2]\!]x$$

This completes the translation of commands. Let *Com* be a command such that in any of its subcommands that is a concurrent composition, at most one of the composed commands contains a `become`. Further, assume that a name is declared as a receptionist at most once, and that `let` constructs with receptionist declarations are not nested under other `let` or conditional constructs. Let x be a fresh name. Then by structural induction on *Com*, we can show that $\{x, \tilde{y}\}; f \vdash [\![Com]\!]x$ if *Com* contains a `become`, and $\emptyset; \{\tilde{y}\} \vdash [\![Com]\!]x$ otherwise, where $\tilde{y}$ is the set of all names declared as receptionists in *Com*, and f is a function that maps all names in its domain to $\perp$.

4.4.3 Behavior definitions

Behavior definitions in SAL are translated to behavior definitions in Aπ as follows

$$[\![\textbf{def } B(\tilde{u})[\tilde{v}] \;\; Com \;\; \textbf{end def}]\!] \;=\; \underline{B} \stackrel{def}{=} (self; \tilde{u})self(\tilde{v}).[\![Com]\!]self$$

Note that the implicitly available reference *self* in a SAL behavior definition becomes explicit in the acquaintance list after translation. Since the body of a behavior definition does not contain receptionist declarations, it follows that $\{self\}; \{self \mapsto \perp\} \vdash self(\tilde{v}).[\![Com]\!]self$. So the RHS is well-typed.

We have completed the translation of various syntactic domains in SAL, and are ready to present the overall translation of a SAL program. Recall that a SAL program consists of a sequence of behavior definitions and a single top level command.

$$[\![BDef_1 \;\; ... \;\; BDef_n \;\; Com]\!] = [\![BDef_1]\!] \;\; \ldots \;\; [\![BDef_n]\!] \;\; [\![Com]\!]x$$

where x is fresh. Since the top level command cannot contain a `become`, its translation does not use the argument x supplied. Indeed, $\{\tilde{y}\}; f \vdash [\![Com]\!]x$, where $\{\tilde{y}\}$ is the set of all names declared as receptionists in Com, and f maps all names in $\{\tilde{y}\}$ to $\perp$.

4.5 Discussion and Related Work

The translation we have given, can be exploited to use the testing theory developed for Aπ, to reason about SAL programs. Note that the characterization of may-testing for Aπ applies unchanged to SAL. This is because the set of experiments possible in SAL have the same distinguishing power as the experiments in Aπ. Specifically, the canonical observers constructed for Aπ (Section 3.3) are also expressible in SAL, and it follows immediately from Lemmas 3.3 and 3.4 that these observers have all the distinguishing power, i.e. are sufficient to decide $\stackrel{\sqsubset}{\sim}_{\rho}$.

Translational semantics for actor languages similar to SAL has been previously attempted. In [25] a simple actor language is translated into linear logic formulae, and computations are modeled as deductions in the logic. In [49] an actor-based object-oriented language is translated into HACL extended with records [48]. These translations provide a firm foundation for further semantic investigations. However, to reap the benefits of these translations, one still has to explicitly characterize actor properties such as locality and uniqueness in the underlying formal system, and identify the changes to the theory due to them. For instance, the asynchronous π-calculus can be seen as the underlying system of our translation, whereas only Aπ terms correspond to SAL programs, and the characterization of may testing for Aπ is very different from that for asynchronous π-calculus.

In [7], testing equivalence is studied on a practical actor language, which consists of a functional core extended with the actor primitives for concurrency. Although no alternate characterization for the operational notion of testing is established, proof techniques that embody sufficient conditions are presented. The techniques are used to derive a set of laws that could be used for instance, to prove correctness of certain compiler trans-

formations. In comparison, we have given an abstract characterization of may testing, that we believe is more generally applicable and easy to use. Although, SAL is a very simple language, it can be enriched with higher level programming constructs without altering the characterization. This is corroborated by the work in [54, 55], where a high level actor languages are translated to a more basic kernel languages (similar to SAL) in such a way that the source and its translation exhibit the same set of traces.

Chapter 5

Decidability Results for Testing Equivalences

In this chapter, we investigate decidability of may and must testing equivalences over π-calculus processes. We also establish lower bounds for computational resources required to decide these equivalences. As one might expect, may and must equivalences are undecidable over general π-calculus processes. We therefore focus on a simpler class of processes, namely asynchronously communicating finite state machines that are open to interaction with their environment. These systems can have an infinite state space since their state includes the multiset of undelivered messages and since message deliveries can be arbitrarily delayed. However, these systems constitute a very simple subclass of π-calculus processes; communication uses finite alphabet and does not involve name passing, there is no dynamic name generation, and the number of communicating machines is fixed. In fact, these systems constitute a subclass of asynchronous CCS [19]. Our choice of this simple model is justified because, as we will see, even simple extensions of this model lead to undecidability of may and must testing.

We define parameterized may and must preorders, $\stackrel{\sqsubset}{\sim}_\rho$ and $\stackrel{\sqsubset}{\sim}^M_\rho$ respectively, on asynchronous finite state machines (AFSM). We show that the may preorder $\stackrel{\sqsubset}{\sim}_\emptyset$ is decidable for AFSMs and is EXPSPACE-hard, and that the preorder $\stackrel{\sqsubset}{\sim}_\rho$ for $\rho \geq 2$ is undecidable. Similarly, we show that the parameterized must preorder $\stackrel{\sqsubset}{\sim}^M_\rho$ for $\rho \geq 2$ is undecidable.

The decidability of $\overset{\sqsubset M}{\sim}_{\emptyset}$ is much harder problem and is still open, but we present some partial results in this direction. Decidability results for may testing were previously known only for the simple class of finite state machines [45]. Our results extend this to a more expressive class of infinite state systems. Further (as we will see) our undecidability results sharply identify the decidability boundaries for may and must testing.

Following is the layout of this chapter. In Section 5.1, we formally define AFSMs and prove some useful properties. In Section 5.2, we related AFSMs to other popular asynchronous process models. In Section 5.3, we instantiate the framework of may and must testing on AFSMs. In Section 5.4, we present decidability results for may testing, and in Section 5.5 we present the results for must testing. We conclude the chapter in Section 5.6 with a discussion and comments on possible directions for further work.

5.1 Asynchronous Finite State Machines

We assume disjoint infinite sets of names $\mathcal{N}$ and co-names $\overline{\mathcal{N}}$, and a bijection $\bar{\cdot} : \mathcal{N} \to \overline{\mathcal{N}}$. We let $\sum$ range over finite sets of names, and write $\overline{\sum}$ to denote the set $\{\overline{a} \mid a \in \sum\}$. Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$ be an FSM with τ-moves. Specifically, Q is the finite set of states, $\sum \cup \overline{\sum}$ the finite alphabet set, $\rightarrow \subseteq Q \times (\sum \cup \overline{\sum} \cup \{\tau\}) \times Q$ the transition relation, q_0 the start state, and $F \subseteq Q$ the set of final states. We let p, q range over Q, and a, b, c over $\sum$. We call $\sum \cup \overline{\sum} \cup \{\tau\}$ the set of actions, and let α range over it. The actions in $\sum$ are called *input* actions, the actions in $\overline{\sum}$ the *output* actions, and τ the *internal* action. We write $\hat{\alpha}$ to mean α if $\alpha \neq \tau$, and ϵ otherwise. The set $\sum \cup \overline{\sum}$ is the set of visible actions, and we let β range over it. We define the complementation function $\bar{\cdot}$ on visible actions so that the complement of an input is the corresponding output, and vice versa.

We write $p \xrightarrow{\alpha} q$ instead of $(p, \alpha, q) \in \rightarrow$, $p \Longrightarrow q$ if $p \xrightarrow{\tau}{}^{*} q$, and $p \overset{\alpha}{\Longrightarrow} q$ if $p \Longrightarrow \xrightarrow{\alpha} \Longrightarrow q$. We call $(\sum \cup \overline{\sum})^*$ the set of traces, and let r, s, t range over it. For $s = \epsilon$ we write $p \xrightarrow{s} q$ if $p = q$, and $p \overset{s}{\Longrightarrow} q$ if $p \Longrightarrow q$. For $s = \beta.s'$ we write $p \xrightarrow{s} q$ if $p \xrightarrow{\beta} \xrightarrow{s'} q$, and $p \overset{s}{\Longrightarrow} q$ if $p \overset{\beta}{\Longrightarrow} \overset{s'}{\Longrightarrow} q$. We define $L(p) = \{s \mid p \overset{s}{\Longrightarrow} q,\ q \in F\}$, and $L(M) = L(q_0)$. For a set S, we write $\mathcal{P}(S)$ to denote the powerset of S, and $\{|S|\}$ to

denote the set of all (possibly infinite) multisets of S. We let B range over $\{|\sum|\}$. For a trace r, we write $\{|r|\}_i$ to denote the multiset of all input actions in r, and $\{|r|\}_o$ for the multiset of all output actions in r. We define $\{|r|\} = \{|r|\}_i \cup \{|r|\}_o$. The complementation function is lifted from the set of visible actions to multisets of visible actions the obvious way.

Definition 5.1 (asynchronous transitions) *Given an FSM $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$, we define the set of its configurations as $Q \times \{|\sum|\}$, and we define an (asynchronous) transition relation $\longrightarrow_A \subseteq (Q \times \{|\sum|\}) \times (\sum \cup \overline{\sum} \cup \tau) \times (Q \times \{|\sum|\})$ as*

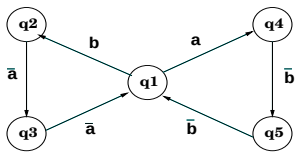
1. $(q, B) \xrightarrow{a}_A (q, B \cup \{a\})$.
2. $(q, B) \xrightarrow{\overline{a}}_A (q, B \setminus \{a\})$ *if* $a \in B$.
3. $(q, B) \xrightarrow{\tau}_A (q', B')$ *if any of the following is true*
   (a) $q \xrightarrow{\tau} q'$, $B' = B$.
   (b) $q \xrightarrow{a} q'$, $a \in B$ *and* $B' = B \setminus \{a\}$,
   (c) $q \xrightarrow{\overline{a}} q'$, $B' = B \cup \{a\}$.

The binary relations $\xrightarrow{s}_A, \Longrightarrow_A, \overset{s}{\Longrightarrow}_A$ on configurations are defined as expected. We define $L_A(q, B) = \{s \mid (q, B) \overset{s}{\Longrightarrow}_A (q', B'), q' \in F\}$. We write $L_A(q)$ as a shorthand for $L_A(q, \emptyset)$, and define the asynchronous language of M, $L_A(M) = L_A(q_0)$. □

We call an FSM M along with its asynchronous transition relation as an AFSM. The set of states of an AFSM is the set of its configurations. A configuration is composed of the control state of the AFSM and the message buffer. The buffer contains inputs received from the environment and outputs produced by the AFSM, which have not yet been consumed. Figure 5.1 illustrates asynchronous transitions of an example AFSM.

The terminology – AFSM – may be a bit misleading because the set of states of an AFSM is infinite. For instance, for the AFSM of Figure 5.1 we can show that

$$(q_1, \{\}) \overset{a}{\Longrightarrow}_A (q_1, \{a^n, b^m\}) \qquad \text{if } n, m \geq 0 \text{ and } n \equiv m + 1 \ (\mathsf{mod}\ 3)$$



$$
\begin{aligned}
(q_1,\{\}) &\xrightarrow{a}_A (q_1,\{a\}) \xrightarrow{\tau}_A (q_4,\{\}) \\
&\xrightarrow{\tau}_A (q_5,\{b\}) \xrightarrow{\tau}_A (q_1,\{b,b\}) \\
&\xrightarrow{\bar{b}}_A (q_1,\{b\}) \xrightarrow{\tau}_A (q_2,\{\}) \\
&\xrightarrow{\tau}_A (q_3,\{a\}) \xrightarrow{\tau}_A (q_1,\{a,a\})
\end{aligned}
$$

Figure 5.1: An asynchronous transition sequence of an example AFSM.

Following is some notation and simple facts that will be useful later on. Let $\#(a, B)$ denote the number of times a occurs in the multiset B. For a sequence of multisets B_i, we define $\sqcup_i B_i$ as the multiset which satisfies for all a, $\#(a, \sqcup_i B_i) = \max_i \#(a, B_i)$.

Lemma 5.1 *(1) If $B \subseteq B'$ then $L_A(q, B) \subseteq L_A(q, B')$. (2) If $B_1 \subseteq B_2 \subseteq \ldots$ and $B = \sqcup_i B_i$, then $L_A(q, B) = \cup_i L_A(q, B_i)$.* □

For an FSM M, we now characterize the relationship between $L(M)$ and $L_A(M)$. This characterization will be useful in Section 5.4.

Definition 5.2 *Given a set of names $\sum$, let $\triangleright$ be the smallest reflexive transitive relation on $(\sum \cup \overline{\sum})^*$ that is closed under the following rules*

1. $s_1.s_2 \triangleright s_1.a.s_2$
2. $s_1.\beta.a.s_2 \triangleright s_1.a.\beta.s_2$
3. $s_1.s_2 \triangleright s_1.a.\overline{a}.s_2$
4. $s_1.\overline{a}.s_2 \triangleright s_1.s_2$
5. $s_1.\overline{a}.\beta.s_2 \triangleright s_1.\beta.\overline{a}.s_2$
6. $s_1.\overline{a}.a.s_2 \triangleright s_1.s_2$

We lift $\triangleright$ to sets of traces as $R \triangleright S$ if for every $s \in S$ there is $r \in R$ such that $r \triangleright s$. We define the closure of S under the relation $\triangleright$, denoted $[S]_\triangleright$, as the smallest set that contains S and that is closed under $\triangleright$. □

Strictly speaking, in Definition 5.2, we have defined a family of relations indexed by the set $\sum$, and hence $\triangleright$ has to be annotated with $\sum$. But to keep the notation simple, we ignore this detail, and instead ensure that $\sum$ is clear from context.

The idea behind Definition 5.2 is that, if $s \triangleright r$ and $s \in L_A(M)$, then $r \in L_A(M)$. Rule 1 captures the fact that an AFSM can always receive an input from its environment, while rule 2 captures the fact that it can perform the inputs in any order. Rule 3 states

that complementary asynchronous input and outputs can always be performed; the input received can simply be buffered and output back to the environment in the next step. Rules 4-6 are duals of the first 3 rules. Rule 5 states that outputs can be buffered and emitted to the environment later, while rule 4 accounts for the case where an output is buffered and never emitted. A buffered output can also be internally consumed by a later input, and this is reflected in rule 6.

We are now ready to characterize the relationship between $L(M)$ and $L_A(M)$.

Theorem 5.1 *For an FSM M, $L_A(M) = [L(M)]_{\triangleright}$.*

Proof: Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$. First, $[L(M)]_{\triangleright} \subseteq L_A(M)$ is a consequence of the following two observations which are easy to prove: (i) $L(M) \subseteq L_A(M)$, and (ii) if $s \in L_A(M)$ and $s \triangleright r$ by a single (and hence arbitrarily many) application of rules in Definition 5.2, then $r \in L_A(M)$. Next, we show $L_A(M) \subseteq [L(M)]_{\triangleright}$. Suppose

$$(q_0, \emptyset) \xrightarrow{\alpha_1}_A (q_1, B_1) \xrightarrow{\alpha_2}_A \ldots \xrightarrow{\alpha_n}_A (q_n, B_n)$$

and $r = \hat{\alpha}_1.\cdots.\hat{\alpha}_n$. We prove the stronger statement that there is s such that $q \stackrel{s}{\Longrightarrow} q_n$, $s \triangleright r$, and $B_n = (\{|r|\}_i \cup \overline{\{|s|\}_o}) \setminus (\overline{\{|r|\}_o} \cup \{|s|\}_i)$. Intuitively, the message buffer B_n contains all the inputs from the environment and the outputs by M, that have neither been consumed by M nor output to the environment. From Definition 5.2, we see that the above expression for B_n encodes the number of times rules 1 and 4 are applied in any derivation of $s \triangleright r$; only for these rules does the expression evaluate to a non-empty multiset when r is set to the RHS and s to the LHS of the rule. Specifically, for all $b \in \mathcal{N}$, $\#(b, B_n)$ equals the number of applications of rules 1 or 4 of Definition 5.2 with the meta-variable a instantiated to b, in any derivation of $s \triangleright r$.

The proof is by induction on n. The base case $n = 0$ is trivial. For the induction step, we may assume there is s' such that $q \xrightarrow{s'} q_{n-1}$, $s' \triangleright r' = \hat{\alpha}_1.\cdots.\hat{\alpha}_{n-1}$, and $B_{n-1} = (\{|r'|\}_i \cup \overline{\{|s'|\}_o}) \setminus (\overline{\{|r'|\}_o} \cup \{|s'|\}_i)$. We now only consider the case where $\alpha_n = \overline{a}$; the other cases are similar. Then $a \in B_{n-1}$, and therefore for a given derivation of $s \triangleright r$,

there is an application of rule 1 or 4 of Definition 5.2 in the derivation. Let $s = s'$. We have two subcases:

- The derivation of $s' \triangleright r'$ contains an instance of rule 1. Then we can derive $s \triangleright r'.\overline{a} = r$ from a derivation of $s' \triangleright r'$, by replacing an instance of rule 1 with an instance of rule 3, and delaying the output introduced to the very end by repeated application of rule 5.

- The derivation of $s' \triangleright r'$ contains an instance of rule 4. Then we can derive $s \triangleright r'.\overline{a} = r$ from a derivation of $s' \triangleright r'$, by replacing an instance of rule 4 with repeating applications of rule 5 that delay the output $\overline{a}$ to the very end.

Note that in both cases $B_n = B_{n-1} \setminus \{a\} = (\{|r|\}_i \cup \overline{\{|s|\}_o}) \setminus (\overline{\{|r|\}_o} \cup \{|s|\}_i)$. Thus the induction hypothesis also holds for n. □

Note that Theorem 5.1 implies that $L(M) \subseteq L_A(M)$ and that $L_A(M)$ is closed under $\triangleright$.

5.2 Related Asynchronous Process Models

It is a relatively simple exercise to show that AFSMs are a special class of π-calculus processes by encoding a given AFSM as a π-calculus process. Popular models of concurrency besides the asynchronous π-calculus include Multiset Automata (MSA) [17], Petri nets [70], Vector Addition Systems [70], and asynchronous CCS [19]. Multiset Automata are a special class of Petri nets [17], which are in turn a special type of asynchronous CCS processes [33]. Asynchronous CCS is itself a fragment of π-calculus (see Figure 5.2). Vector addition systems are known to be equivalent to Petri nets [70].

AFSMs are simpler than all of these models and can be best seen as a special class kind of MSAs with τ transitions. The central difference between an MSA and an AFSM is that in an AFSM the labels on the transitions are intimately linked to the operations on the multiset buffer. AFSMs also exactly correspond to the fragment of asynchronous CCS without the restriction operator, and with the following two constraints

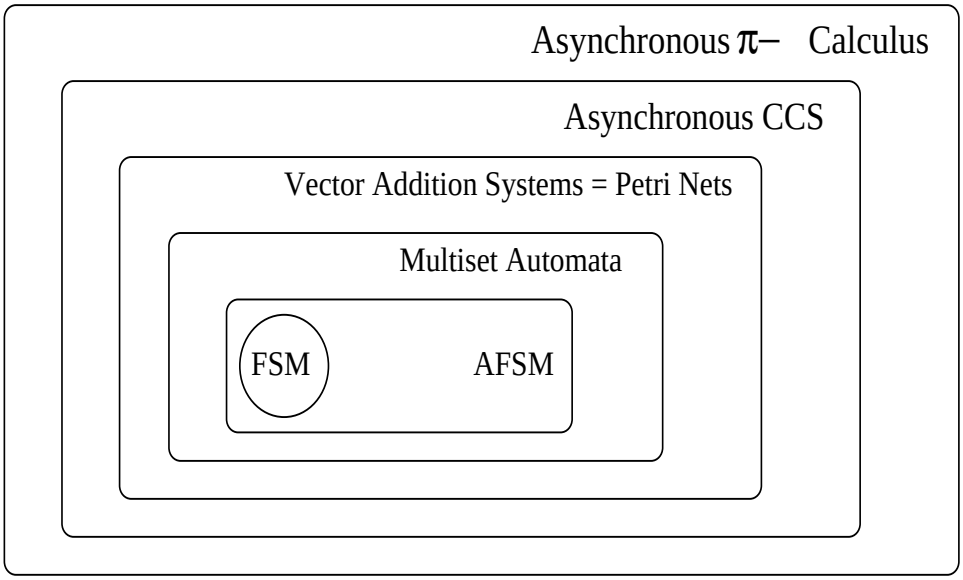


Figure 5.2: A hierarchy of asynchronous computational models.

- Every process identifier occurring inside a recursion is guarded with an action prefix.
- Whenever two processes are composed in a recursive definition one of them is a collection of messages, i.e. of form $\overline{a}_1 | \ldots | \overline{a_n}$.

Due to the simplicity of AFSMs, some of the verification problems we consider in the following sections are decidable over AFSMs, but are undecidable over MSAs and hence over all other more expressive process models. Further, since the parameterized may and must preorders emulate privatization of certain names to the process being tested, the undecidability results for parameterized may and must testing that we present in Sections 5.4 and 5.5, imply that the parameterized equivalences become undecidable over the fragment of asynchronous CCS mentioned above even if we allow the restriction operator to occur only outside recursion.

5.3 Testing Equivalences Over AFSMs

We now instantiate the notions of may and must testing on AFSMs, and present semantic characterizations that will be useful in the sections that follow. Note that the fact that

AFSMs are a special class of asynchronous π-calculus processes does not by itself imply that the characterizations of testing equivalences for asynchronous π-calculus also apply to AFSMs. This is because the set of observers that can be used for testing in π-calculus is strictly larger than that in AFSMs.

We begin with a formal definition of may testing on AFSMs.

Definition 5.3 (asynchronous experiment) *Let* $M_1 = (Q_1, \sum_1 \cup \overline{\sum_1}, \rightarrow_1, q_1, F_1)$ *and* $M_2 = (Q_2, \sum_2 \cup \overline{\sum_2}, \rightarrow_2, q_2, F_2)$. *An asynchronous experiment with* M_1 *and* M_2 *is of form* $(p|q, B)$, *where* $p \in Q_1$, $q \in Q_2$, *and* $B \in \{|\sum_1 \cup \sum_2|\}$. *We define a transition relation on asynchronous experiments as* $(p|q, B) \longrightarrow_A (p'|q', B')$ *if*

1. $a \in B, B' = B \setminus \{a\}$, *and* $p \xrightarrow{a} p', q = q'$ *or* $p = p', q \xrightarrow{a} q'$.
2. $B' = B \cup \{a\}$, *and* $p \xrightarrow{\overline{a}} p', q = q'$ *or* $p = p', q \xrightarrow{\overline{a}} q'$.
3. $B' = B$, *and* $p \xrightarrow{\tau} p', q = q'$ *or* $p = p', q \xrightarrow{\tau} q'$.

We define the relation, $\Longrightarrow_A$, *on asynchronous experiments as the reflexive transitive closure of* $\longrightarrow_A$. □

We are now ready to define parameterized may testing on AFSMs. In the following, for $\rho \subseteq \mathcal{N}$ we say that M *respects* the interface ρ, if $\rho \cap \sum = \emptyset$. Thus, if M respects ρ, it performs neither input nor output actions at names in ρ.

Definition 5.4 (parameterized asynchronous may testing)
We say M_1 *may* M_2 *if* $(q_1|q_2, \emptyset) \Longrightarrow_A (p_1|p_2, B)$ *for some* $p_2 \in F_2$. *For* $\rho \subseteq \mathcal{N}$, *we say* $M_1 \stackrel{\sqsubset}{\sim}_\rho M_2$ *if for every* M *that respects the interface* ρ, *we have* M_1 *may* M *implies* M_2 *may* M. *We say* $M_1 \simeq_\rho M_2$ *if* $M_1 \stackrel{\sqsubset}{\sim}_\rho M_2$ *and* $M_2 \stackrel{\sqsubset}{\sim}_\rho M_1$. *We write* $\stackrel{\sqsubset}{\sim}$ *as a shorthand for* $\stackrel{\sqsubset}{\sim}_\emptyset$, *and similarly* $\simeq$ *for* $\simeq_\emptyset$.

Theorem 5.2 characterizes the parameterized asynchronous may preorder. For a set of names ρ and a trace s, we write $s\lceil\rho$ for the trace obtained from s by deleting all the actions in $\rho \cup \overline{\rho}$. For a set of traces L, we define $L\lceil\rho$ to be the set of all traces s in L

such that $s\lceil\rho = s$. Note that $L\lceil\rho$ is not the usual lifting of the function $\cdot\lceil\rho$ on traces, to sets of traces.

Theorem 5.2 (characterization of parameterized may testing)
Let $M_1 = (Q_1, \sum_1 \cup \overline{\sum_1}, \rightarrow_1, q_1, F_1)$, *and* $M_2 = (Q_2, \sum_2 \cup \overline{\sum_2}, \rightarrow_2, q_2, F_2)$. *Let* $\sum = \sum_1 \cup \sum_2$, *and* $M_2' = (Q_2, \sum \cup \overline{\sum}, \rightarrow_2, q_2, F_2)$. *Then* $M_1 \stackrel{\sqsubset}{\sim}_\rho M_2$ *if and only if* $L_A(M_1)\lceil\rho \subseteq L_A(M_2')\lceil\rho$. □

We skip the proof as it is a simple adaptation of the proof for characterization of (unparameterized) may testing for asynchronous CCS [19], using the ideas presented in Chapter 2 to account for the parameterization. Note that, we consider M_2' instead of M_2 due to the following reason. If $\sum_1 \setminus (\sum_2 \cup \rho) \neq \emptyset$ then there is always an $s \in L_A(M_1)$ but $s \notin L_A(M_2)$. But on the other hand, since inputs of a process are not observable due to asynchrony, it is not necessary that $M_1 \stackrel{\sqsubset}{\not\sim}_\rho M_2$.

We now instantiate parameterized must testing on AFSMs.

Definition 5.5 (asynchronous must testing) *We say* M_1 *must* M_2 *if for every maximal computation*

$$(q_1|q_2, \emptyset) \longrightarrow_A (q_{11}|q_{21}, B_1) \longrightarrow_A (q_{12}|q_{22}, B_2) \longrightarrow_A (q_{13}|q_{23}, B_3) \longrightarrow_A \cdots$$

$q_2 \in F_2$ *or* $q_{2i} \in F_2$ *for some* i. *For an interface* ρ, *we say* $M_1 \stackrel{\sqsubset}{\sim}_\rho^M M_2$ *if for every FSM* M *that respects* ρ, *we have* M_1 *must* M *implies* M_2 *must* M. *We say* $M_1 \simeq_\rho^M M_2$ *if* $M_1 \stackrel{\sqsubset}{\sim}_\rho^M M_2$ *and* $M_2 \stackrel{\sqsubset}{\sim}_\rho^M M_1$. *We write* $\stackrel{\sqsubset}{\sim}^M$ *as a shorthand for* $\stackrel{\sqsubset}{\sim}_\emptyset^M$, *and similarly* $\simeq^M$ *for* $\simeq_\emptyset^M$. □

An aspect of process behavior that must testing is sensitive to is divergence, where a process may engage in an infinite sequence of internal transitions without allowing the environment's computation to progress. Divergence is observable in must testing, for example by the observer that simply makes an internal transition to an accept state; a process passes the test proposed by this observer if and only if it does not diverge.

Definition 5.6 (divergence) *Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$. We say a configuration (q, B) (asynchronously) converges, written $(q, B) \downarrow_A$, if there is no infinite sequence of transitions*

$$(q, B) \xrightarrow{\tau}_A (q_1, B_1) \xrightarrow{\tau}_A (q_2, B_2) \xrightarrow{\tau}_A \ldots$$

For $s \in (\sum \cup \overline{\sum})^*$, we say (q, B) converges along s, written $(q, B) \downarrow_A s$, if whenever s' is a prefix of s and $(q, B) \stackrel{s'}{\Longrightarrow}_A (q', B')$, it is the case that $(q', B') \downarrow_A$. We write $(q, B) \uparrow_A s$, if it is not the case that $(q, B) \downarrow_A s$. We write $q \downarrow_A$ as a shorthand for $(q, \emptyset) \downarrow_A$, and similarly for $q \downarrow_A s$, $q \uparrow_A$, and $q \uparrow_A s$. We write $M \downarrow_A s$ if $q_0 \downarrow_A s$, and similarly for $M \uparrow_A s$. □

Intuitively, $(q, B) \downarrow_A s$ means that the configuration (q, B) can never reach a diverging state after asynchronously exhibiting any prefix of s. Another aspect of the process behavior that is relevant to must testing is the set of all actions a process can perform after exhibiting a given trace.

Definition 5.7 (acceptance sets) *Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$. For $s \in (\sum \cup \overline{\sum})^*$ and a configuration (q, B) of M, we define*

$$(q, B) \ \underline{\textit{after}} \ s = \{(q', B') \mid (q, B) \stackrel{s}{\Longrightarrow}_A (q', B')\}$$

We write q $\underline{\textit{after}}$ s as a shorthand for $(q, \emptyset)$ $\underline{\textit{after}}$ s. For a set of configurations C and $L \subseteq \overline{\sum}$, we say C $\underline{must}$ L if and only if for each $(q, B) \in C$ there exists $\overline{a} \in L$ such that $(q, B) \stackrel{\overline{a}}{\Longrightarrow}_A$. We write M $\underline{\textit{after}}$ s for q_0 $\underline{\textit{after}}$ s. □

Informally, (q, B) $\underline{\text{after}}$ s is the set of all configurations that (q, B) can reach after performing a sequence of asynchronous transitions whose visible content is s. For a set of configurations C, C $\underline{must}$ L if every configuration in C, possibly after a sequence of internal transitions, can perform an output at least one of the co-names in L. Note that we are only concerned with $L \subseteq \overline{\sum}$, i.e only with sets of output actions, because asynchronous inputs are not observable. We are now ready to define an alternate preorder on AFSMs, which characterizes the must preorder.

Definition 5.8 (alternate must preorder) *Let* $M_1 = (Q_1, \sum \cup \overline{\sum}, \rightarrow_1, q_1, F_1)$, $M_2 = (Q_2, \sum \cup \overline{\sum}, \rightarrow_2, q_2, F_2)$. *For a set of names* ρ, *let* $\sum' = \sum \backslash \rho$. *We say* $M_1 \ll_\rho^m M_2$ *if for every* $L \subset \overline{\sum'}$ *and* $s \in (\sum' \cup \overline{\sum'})^*$ *if* $M_1 \downarrow_A s$ *then (a)* $M_2 \downarrow_A s$, *and (b)* M_1 *after* s *must* L *implies* M_2 *after* s *must* L. □

The following theorem characterizes the parameterized must preorder on AFSMs. We again skip the proof as it is a simple adaptation of the proof for characterization of (unparameterized) must preorder over asynchronous CCS [19].

Theorem 5.3 (characterization of must testing)
Let $M_1 = (Q_1, \sum_1 \cup \overline{\sum_1}, \rightarrow_1, q_1, F_1)$, *and* $M_2 = (Q_2, \sum_2 \cup \overline{\sum_2}, \rightarrow_2, q_2, F_2)$. *Let* $\sum = \sum_1 \cup \sum_2$, $M_1' = (Q_1, \sum \cup \overline{\sum}, \rightarrow_1, q_1, F_1)$, *and* $M_2' = (Q_2, \sum \cup \overline{\sum}, \rightarrow_2, q_2, F_2)$. *Then* $M_1 \stackrel{\sqsubset}{\sim}_\rho^M M_2$ *if and only if* $M_1' \ll_\rho^m M_2'$. □

5.4 Decidability Results for May Testing

We first show that the unparameterized may preorder $\stackrel{\sqsubset}{\sim}$ over AFSMs is decidable, and that the problem is EXPSPACE-hard. We then show that the parameterized may preorder $\stackrel{\sqsubset}{\sim}_\rho$ is undecidable for $|\rho| \geq 2$.

By Theorem 5.2, we have a decision procedure for $\stackrel{\sqsubset}{\sim}$ if we can decide $L_A(M_1) \subseteq L_A(M_2)$ for arbitrary M_1 and M_2. We now give a decision procedure for the language containment problem over AFSMs. The reader may note that in comparison the language containment problem over the more general class of MSA (with τ-transitions) is undecidable [40].

Deciding if $L_A(M_1) \subseteq L_A(M_2)$ involves comparing two infinite state systems. The following lemma provides a handle to deal with this problem.

Lemma 5.2 *Let* M_1 *and* M_2 *be FSMs with alphabet* $\sum_1$ *and* $\sum_2$ *respectively, and let* $\sum_1 \subseteq \sum_2$. *Then* $L_A(M_1) \subseteq L_A(M_2)$ *if and only if* $L(M_1) \subseteq L_A(M_2)$.

Proof: This is an easy consequence of Theorem 5.1. Since $L(M_1) \subseteq L_A(M_1)$, we have $L_A(M_1) \subseteq L_A(M_2)$ implies $L(M_1) \subseteq L_A(M_2)$. Conversely, suppose $L(M_1) \subseteq L_A(M_2)$.

Let $\triangleright_1$ be the relation as defined in Definition 5.2 with respect to the alphabet $\sum_1$, and similarly $\triangleright_2$ with respect to the alphabet $\sum_2$. Then $L(M_1) \triangleright_1 L_A(M_1)$, and $L_A(M_2)$ is closed under $\triangleright_2$. Since $\sum_1 \subseteq \sum_2$, $L_A(M_2)$ is also closed under $\triangleright_1$. Then $L(M_1) \subseteq L_A(M_2)$ implies $L_A(M_1) \subseteq L_A(M_2)$. □

For the case $\sum_1 \not\subseteq \sum_2$, it is easy to show that $L_A(M_1) \subseteq L_A(M_2)$ if and only if $L_A(M_1) = \emptyset$. Now, checking for emptiness of $L_A(M_1) \neq \emptyset$ is the same as checking for emptiness of $L(M_1)$, and decision procedures for this are well known. So, from now on we may assume that $\sum_1 \subseteq \sum_2$.

As a consequence of Lemma 5.2, in order to decide $L_A(M_1) \subseteq L_A(M_2)$, we only need to compare the (synchronous) transitions of a finite state system with (asynchronous) transitions of an infinite state system. Figure 5.3 shows a naive attempt at a decision procedure that exploits this simplification. The arguments to procedure contained are a control state p of $M_1 = (Q_1, \sum_1 \cup \overline{\sum_1}, \rightarrow_1, q_1, F_1)$ and a set of configurations C of $M_2 = (Q_2, \sum_2 \cup \overline{\sum_2}, \rightarrow_2, q_2, F_2)$. The idea is that the procedure returns true if and only if $L(p) \subseteq L_A(C)$, where $L_A(C) = \cup_{(q,B)\in C} L_A(q, B)$. Thus, to decide if $L_A(M_1) \subseteq L_A(M_2)$ the procedure is to be invoked with arguments $(q_1, M_1, \{(q_2, \emptyset)\}, M_2)$.

The procedure contained recursively matches the synchronous transitions of M_1 starting from p, with asynchronous transitions of M_2 starting from any configuration in C. Without loss of generality, we assume that M_1 does not have any τ transitions between its control states, because otherwise we can eliminate the τ actions by the usual τ-elimination procedure without changing $L(M_1)$. In line 8, we assume a subroutine reach such that for a trace s, $\mathsf{reach}(C, s, M_2) = \cup_{(p_2,B)\in C} \{(p_2', B') \mid (p_2, B) \stackrel{s}{\Longrightarrow}_A (p_2', B')\}$.

Figure 5.3 does not provide a decision procedure since the procedure contained need not terminate due to two reasons. First, the recursion in lines 6 and 9 is in general unbounded. Second, the set $\mathsf{reach}(C, \overline{a}, M)$ may not be finite. For instance for the

```
1       contained(p, M1, C, M2)
2           if p ∈ F1 and ε ∉ LA(C) then return false
3           for all a ∈ Σ1, p' ∈ Q1
4               if p --a-->1 p' then
5                   C' := {(p2, B ∪ {a}) | (p2, B) ∈ C}
6                   if not contained(p', M1, C', M2) then return false
7               if p --ā-->1 p' then
8                   C' := reach(C, ā, M2)
9                   if not contained(p', M1, C', M2,) then return false
10          end for
11          return true
12      end contained
```

Figure 5.3: A naive attempt at deciding the asynchronous language containment problem

AFSM M of Figure 5.1, we have

$$\mathsf{reach}(\{(q_1,\{a\})\},\overline{a},M) = \left\{ \begin{array}{ll} (q_1,\{a^n,b^m\}) & (q_2,\{a^{n+1},b^m\}) \\ (q_3,\{a^{n+2},b^m\}) & (q_4,\{a^n,b^{m+1}\}) \\ (q_5,\{a^n,b^{m+2}\}) & \end{array} \;\middle|\; \begin{array}{l} n,m \geq 0 \\ n \equiv m \ (\mathsf{mod}\ 3) \end{array} \right\}$$

Finally, we also have to provide a procedure to check if $\epsilon \in L_A(C)$ in line 2.

We use the following idea to bound the number of recursive calls. We define $C' \preceq C$ if for every $(q,B') \in C'$ there is $(q,B) \in C$ such that $B' \subseteq B$. Note that as a consequence of Lemma 5.1.1, $C' \preceq C$ implies $L_A(C') \subseteq L_A(C)$. But this implies that an invocation $\mathsf{contained}(p,M_1,C,M_2)$ is redundant if there was a previous invocation $\mathsf{contained}(p,M_1,C',M_2)$ such that $C' \preceq C$. This is because, if the previous invocation $\mathsf{contained}(p,M_1,C',M_2)$ returned true, then $L(p) \subseteq L_A(C') \subseteq L_A(C)$, and hence we know that $\mathsf{contained}(p,M_1,C,M_2)$ should return true. On the other hand, if $\mathsf{contained}(q,M_1,C',M_2)$ returned false, then the procedure would already have terminated by returning false.

The following lemma states a useful property of the relation $\preceq$.

Lemma 5.3 *Given a sequence $C_1, C_2, \ldots$, where $C_i \in \mathcal{P}(Q \times \{|\sum|\})$ are finite sets, there exist m, n such that $m < n$ and $C_m \preceq C_n$.*

Proof: A direct consequence of applying Higman's lemma [39] to natural numbers extended with ω. □

Note that finiteness of C_i does not preclude C_i from containing configurations (q, B) where B is infinite.

To avoid computation of the possibly infinite set $\mathsf{reach}\ (C, \overline{a}, M)$, we compute a finite set of configurations C' such that $L_A(C') = L_A(\mathsf{reach}(C, \overline{a}, M))$, and use C' instead in line 8 of Figure 5.3.

Definition 5.9 *For sets of configurations C_1 and C_2, we say C_2 covers C_1 if (1) $C_1 \preceq C_2$, and (2) $(q, B) \in C_2$ implies there are $B_1 \subseteq B_2 \subseteq \ldots$ such that $B = \sqcup_i B_i$ and $(q, B_i) \in C_1$.*

For instance, for the AFSM in Figure 5.1 and the set $C' = \{\ (q_i, \{a^\omega, b^\omega\})\ \mid\ 1 \leq i \leq 5\ \}$, we have C' *covers* $\mathsf{reach}(\{(q_1, \{a\})\}, \overline{a}, M)$. We write $a^\omega \in B$ to denote that B contains infinitely many a's, and adapt the usual multiset operations and relations accordingly. For instance, $\{|a, b^\omega|\} \cup \{|a, b|\} = \{|a, a, b^\omega|\}$, and $\{|a, b^\omega|\} \setminus \{|a, b|\} = \{|b^\omega|\}$. Following lemma is an easy consequence of Lemma 5.1.

Lemma 5.4 *If C_2 covers C_1 then $L_A(C_1) = L_A(C_2)$.*

Our plan is to compute a finite set of configurations C' such that C' *covers* $\mathsf{reach}(C, \overline{a}, M)$. We first consider the case where the set C contains a single configuration. We use Karp and Miller's algorithm for computing the coverability tree of Petri Nets [46], which applies to AFSMs since they are a special class of Petri nets. This is the subroutine cover shown in Figure 5.4. Roughly, $\mathsf{cover}((q, B), M)$ returns a set of configurations that covers all the configurations that M can reach starting from (q, B) and by a sequence of asynchronous transitions labeled with τ. In line 3, we write B^ω to denote $\{|a^\omega \mid a \in B|\}$. Note that, we have assumed that M has no τ-transitions (between the control states).

```
1    append(v, (q, B))
2        if ∃i s.t. v(i) = (q, B_i) and B ⊆ B_i then return v
3        if ∃i s.t. v(i) = (q, B_i) and B_i ⊆ B then return v.(q, B_i ∪ (B \ B_i)^ω)
4        return v.(q, B)
5    end append

6    cover((q, B), (Q, ∑ ∪ ∑̄, →, q_0, F))
7        V := {(q, B)}
8        repeat
9            V' := V;  V := ∅
10           for all v ∈ V'
11               let v = v'.(q', B')
12                   for all a ∈ ∑, q'' ∈ Q
13                       if q' --a--> q'' and a ∈ B' then
14                           V := V ∪ append(v, (q'', B' \ {a}))
15                       if q' --ā--> q'' then
16                           V := V ∪ append(v, (q'', B' ∪ {a}))
17                   end for all
18               end let
19           end for all
20       until V = V'
21       return configs(V)
22   end cover
```

Figure 5.4: Karp and Miller algorithm for computing the coverability sets.

The expression $\mathsf{configs}(V)$ returns the set of all configurations that occur in the paths (sequences of configurations) in V.

We recall from [46] that the procedure cover terminates for any input $((q, B), M)$, and returns a *finite* set of configurations such that $\mathsf{cover}(\{(q, B)\}, M)$ *covers* $\mathsf{reach}(\{(q, B)\}, \epsilon, M)$. For a given $\overline{a}$, we then extract a set C' from $\mathsf{cover}(\{(q, B)\}, M)$ such that C' *covers* $\mathsf{reach}(\{(q, B)\}, \overline{a}, M)$.

Lemma 5.5 *For $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$, $q \in Q$, and $B \in \{|\sum|\}$, the following statements are true.*

1. *$\epsilon \in L_A(q, B)$ if and only if $(q', B') \in \mathsf{cover}((q, B), M)$ for some $q' \in F$.*
2. *For a given $a \in \sum$, let $C = \{(q', B' \setminus \{a\}) \mid (q', B') \in \mathsf{cover}((q, B), M),\ a \in B'\}$. Then C covers $\mathsf{reach}(\{(q, B)\}, \overline{a}, M)$.*

Proof:

1. Suppose $\epsilon \in L_A(q, B)$. Then $(q, B) \Longrightarrow_A (q', B')$ for some $q' \in F$. Then, since C *covers* $\mathsf{reach}((q, B), \epsilon, M)$, we have $(q', B'') \in C$ for some $B'' \supseteq B'$. Conversely, suppose $(q', B') \in C$ for some $q' \in F$. Then again since C *covers* $\mathsf{reach}((q, B), \epsilon, M)$, we have $(q, B) \Longrightarrow_A (q', B'')$ for some $B'' \subseteq B'$, which implies $\epsilon \in L_A(q, B)$.

2. Suppose $(q', B') \in \mathsf{reach}(\{(q, B)\}, \overline{a}, M)$. Then, $(q, B) \stackrel{\overline{a}}{\Longrightarrow}_A (q', B')$, which implies $(q, B) \Longrightarrow_A (q', B' \cup \{a\}) \stackrel{\overline{a}}{\longrightarrow}_A (q', B')$. Then, $(q', B' \cup \{a\}) \in \mathsf{reach}(\{(q, B)\}, \epsilon, M)$, and since $\mathsf{cover}((q, B), M)$ *covers* $\mathsf{reach}(\{(q, B)\}, \epsilon, M)$, we have $(q', B'') \in \mathsf{cover}$ $((q, B), M)$ for some $B'' \supseteq B' \cup \{a\}$. But then $(q', B'' \setminus \{a\}) \in C$ and $B'' \setminus \{a\} \supseteq B'$. We have shown $\mathsf{reach}(\{(q, B)\}, \overline{a}, M) \preceq C$. Now, suppose that $(q', B') \in C$. Then $(q', B' \cup \{a\}) \in \mathsf{cover}((q, B), M)$. Since $\mathsf{cover}((q, B), M)$ *covers* $\mathsf{reach}(\{(q, B)\}, \epsilon, M)$, there are $B_1 \subseteq B_2 \subseteq \ldots$ such that $(q', B_i) \in \mathsf{reach}(\{(q, B)\}, \epsilon, M)$ and $\sqcup_i B_i = B' \cup \{a\}$. Then there is n such that $a \in B_i$ for all $i \geq n$. Then, we have $(q', B_i \setminus \{a\}) \in \mathsf{reach}(\{(q, B)\}, \overline{a}, M)$ for all $i \geq n$, and $\sqcup_{i \geq n}(B_i \setminus \{a\}) = B'$. Thus, we have shown that C *covers* $\mathsf{reach}(\{(q, B)\}, \overline{a}, M)$. □

```
1    contained(p_1, M_1, C, M_2)
2        L := L ∪ (p_1, C)
3        C'' := ∪_{(p,B)∈C} cover((p, B), M_2)
4        if p_1 ∈ F_1 and for all (p_2, B) ∈ C'' p_2 ∉ F_2 then return false
5        for all a ∈ Σ_1, p'_1 ∈ Q_1
6            if p_1 --a-->_1 p'_1 then
7                C' := {(p_2, B ∪ {a}) | (p_2, B) ∈ C}
8                if (p'_1, C') not covered by L then
9                    if not contained(p'_1, M_1, C', M_2) then return false
10           if p_1 --ā-->_1 p'_1 then
11               C' := {(p, B \ {a})|(p, B) ∈ C'', a ∈ B}
12               if (p'_1, C') not covered by L then
13                   if not contained(p'_1, M_1, C', M_2) then return false
14       end for
15       return true
16   end contained
```

Figure 5.5: An algorithm for deciding asynchronous language containment of AFSMs.

We are now ready to present the correct version of the procedure contained.

Theorem 5.4 *There is an algorithm, which given M_1 and M_2, decides if $L_A(M_1) \subseteq L_A(M_2)$.*

Proof: Figure 5.5 shows the algorithm, which differs from the procedure in Figure 5.3 as follows.

In line 4, instead of checking if $\epsilon \in L_A(C)$, we use Lemma 5.5.1 and check for the equivalent condition that for some $p_2 \in F_2$ and B, $(p_2, B) \in C''$ where $C'' = \cup_{(p,B)\in C}\mathsf{cover}((p, B), M_2)$. In line 11, we exploit Lemmas 5.5.2 and 5.4 to use the set $\{(p, B \setminus \{a\})|(p, B) \in C'', a \in B\}$ which is always finite (because the output of cover is finite), instead of $\mathsf{reach}(C, \overline{a}, M)$ which can in general be infinite.

To ensure termination, we use the variable L to remember all the inputs with which contained has been invoked so far, and we recursively call contained with input (p'_1, C') (lines 9 and 13) only if it is not redundant. The variable L is initially set to $\emptyset$. We say

(q, C) is *covered by* L if there is $(q, C') \in L$ such that $C' \preceq C$. Thus, an input (p'_1, C') is redundant if and only if it is covered by L.

We now show that contained terminates for an input (p, C) provided C is a finite set. The proof is by contradiction. Suppose contained doesn't terminate for an input (p, C), where C is a finite set. Then contained is called an infinite number of times with arguments, say (p_i, C_i), each of which is added to L. The sequence (p_i, C_i) has a subsequence (p_k, C_{k_i}) for some k, since $|Q_1|$ is finite. Since C is finite it follows that each C_{k_i} is finite. Then by Lemma 5.3, there are m, n such that $m < n$ and $C_{k_m} \preceq C_{k_n}$. But this is impossible because when contained is called with arguments (p_k, C_{k_n}), the argument is already covered by L. Contradiction. □

We now contrast our algorithm with the usual procedure for deciding synchronous language containment over FSMs. To decide $L(M_1) \subseteq L(M_2)$ the usual procedure is to first construct $\neg M_2$ such that $L(\neg M_2) = \overline{L(M_2)}$, then construct $M_1 \cap \neg M_2$ such that $L(M_1 \cap \neg M_2) = L(M_1) \cap L(\neg M_2)$, and finally check if $L(M_1 \cap \neg M_2) = \emptyset$. This procedure cannot be used for deciding $L_A(M_1) \subseteq L_A(M_2)$ because we cannot in general construct $\neg M_2$, i.e. the set of asynchronous languages of FSMs is not closed under complementation.

Although we do not have a clear upper bound on the running time of the algorithm contained, we show that the asynchronous language containment problem is EXPSPACE-hard.

Theorem 5.5 *The asynchronous language containment problem for FSMs is EXPSPACE-hard.*

Proof: We reduce the halting problem of counter machines of size n, whose counters are bounded by 2^{2^n}, to the given problem. The halting problem for such counter machines is known to be EXPSPACE-complete [42]. We use a construction first presented by Lipton [51]; for a more recent exposition, see [28]. Given a counter machine C, Lipton constructs an unlabeled Petri Net $P = (S, T, F, \mu)$ with designated places s_{init} and s_{halt}

such that P reaches any marking with a token in place s_{halt} if and only if C halts. In addition, Lipton's construction has the following properties

1. The number of places and transitions is $O(n)$, and P can be constructed in time that is a polynomial in n.

2. The initial marking μ has no tokens in any place, except s_{init} which has one token.

3. The set of places S can be partitioned into 3 sets: S_c, which keeps track of the control state of the counter machine C; S_r, which are used in some recursive computation that P makes; and S_v, which keeps track of the values of the counters of C and some additional variables needed in the simulation of C. Each of the sets S_c, S_r, S_v is of size $O(n)$, and the designated places s_{init} and s_{halt} belong to S_c.

4. The preset and postset of any transition t involve exactly one place in S_c, at most one place in S_r, and at most one place in S_v. In other words, $|S_c \cap {}^\bullet t| = 1$, $|S_r \cap {}^\bullet t| \leq 1$, and $|S_v \cap {}^\bullet t| \leq 1$; similar conditions hold for $t^\bullet$.

We will construct AFSMs M_1 and M_2 such that $L_a(M_1) \subseteq L_a(M_2)$ if and only if P reaches a marking with a token in place s_{halt}. The construction of M_1 and M_2 will be done in time which is a polynomial in n. We first describe M_2 that simulates the net P. Let $Q = \{r, w\} \times S_c \times (S_r \cup \{-\}) \times (S_v \cup \{-\})$, and let $\sum = S_r \cup S_v \cup \{A\}$ for $A \notin S_r \cup S_v$. Define,

$$M_2 = (Q, \sum \cup \overline{\sum}, \rightarrow, (r, \langle s_{\text{init}}, -, - \rangle), Q)$$

where the transition relation is defined by the following rules. For $s \in S_c$, $a \in S_r$, $v \in S_v$, $t \in T$, $x, x' \in S_r \cup \{-\}$ and $y, y' \in S_v \cup \{-\}$,

- $(r, \langle s, -, y \rangle) \xrightarrow{a} (r, \langle s, a, y \rangle)$
- $(r, \langle s, x, - \rangle) \xrightarrow{v} (r, \langle s, x, v \rangle)$
- $(r, \langle s, x, y \rangle) \xrightarrow{\tau} (w, \langle s', x', y' \rangle)$, where ${}^\bullet t = \{s, x, y\}$ and $t^\bullet = \{s', x', y'\}$

- $(w, \langle s, a, y\rangle) \xrightarrow{\bar{a}} (w, \langle s, -, y\rangle)$
- $(w, \langle s, x, v\rangle) \xrightarrow{\bar{v}} (w, \langle s, x, -\rangle)$
- $(w, \langle s, -, -\rangle) \xrightarrow{\tau} (r, \langle s, -, -\rangle)$
- $(r, \langle s_{\text{halt}}, x, y\rangle) \xrightarrow{\bar{A}} (w, \langle s_{\text{halt}}, x, y\rangle)$

A marking of places in the net P is encoded as the message buffer of M. To simulate a transition of P, M reads its message buffer and non-deterministically makes a transition that is enabled, i.e. whose preset has all places with non-zero marking. After the transition, M performs output actions so that its message buffer corresponds to the new marking that P reaches after its transition. Finally, M_2 emits a special message $\bar{A}$ when there is a token in s_{halt}. Now observe that $\bar{A} \in L_a(M_2)$ if and only if $(r, \langle s_{\text{init}}, -, -\rangle) \Rightarrow_A (r, \langle s_{\text{halt}}, -, -\rangle)$, which in turn can happen if and only if P reaches a marking with a token in s_{halt}. In addition, $|Q| = O(n^3)$, $\sum = O(n)$, and so $|M_2| = O(n^6)$. Moreover M_2 can be constructed in $O(n^6)$ time given P.

Finally, $M_1 = (\{q_1, q_2\}, \{A\} \cup \{\bar{A}\}, \rightarrow, q_1, q_2)$, where the only transition in M_1 is $q_1 \xrightarrow{\bar{A}} q_2$. Observe that $L_a = [\{\bar{A}\}]_{\triangleright}$. Hence, we have, $L_a(M_1) \subseteq L_a(M_2)$ if and only if $\bar{A} \in L_a(M_2)$ if and only if $(r, \langle s_{\text{init}}, -, -\rangle) \Rightarrow_A (r, \langle s_{\text{halt}}, -, -\rangle)$ if and only if P reaches a marking with a token in s_{halt} if and only if C halts. □

Now, by Theorems 5.5 and 5.2, the asynchronous may preorder relation $\overset{\sqsubset}{\sim}$ is decidable and is EXPSPACE-hard. Further, since $\simeq = \overset{\sqsubset}{\sim} \cap \overset{\sqsubset}{\sim}^{-1}$, it follows that deciding $\simeq$ is also EXPSPACE-hard. We now show that the parameterized may equivalence is in general undecidable.

Theorem 5.6 *The parameterized asynchronous may equivalence $\simeq_\rho$ on AFSMs is undecidable.*

Proof: We reduce the language equality problem for labeled Petri nets, which is known to be undecidable [35], to the given problem. Specifically, given Petri nets P_1 and P_2 we

construct AFSMs M_1, M_2, and ρ such that $L(P_1) \subseteq L(P_2)$ if and only if $L_A(M_1)\lceil\rho \subseteq L_A(M_2)\lceil\rho$. We are then done by Theorem 5.2.

We now construct for a given Petri net P an AFSM M that simulates P. The construction is along the lines of the one presented in the proof of Theorem 5.5, but it works for arbitrary Petri nets and also accounts for the transition labels of the net. Suppose $P = (S, T, F, \lambda, \mu)$ is a Petri net, where S and T are disjoint sets of places and transitions, $F \subseteq (S \times T) \cup (T \times S)$ is the flow relation, $\lambda : T \to L$ is the labeling function, and $\mu : S \to \mathbb{N}$ is the initial marking. Without any loss of expressive power we may assume that μ leaves a single token at exactly one of the places. Define

$$M = (\{r, w\} \times \mathcal{P}(S), \sum \cup \overline{\sum}, \to, (w, \{\mu\}), \{r, w\} \times \mathcal{P}(S))$$

where $\sum = S \cup L \cup \{A\}$ for some $A \notin S \cup L$, $\{\mu\}$ denotes the singleton $\{s\}$ such that $\mu(s) \neq 0$, and the transition relation $\to$ is defined by the following rules. For $X \subseteq \mathcal{P}(S)$, $s \in S$ and $t \in T$:

- $(r, X) \xrightarrow{s} (r, X \cup \{s\})$ if $s \notin X$.
- $(r, X) \xrightarrow{\lambda(t)} (w, (X \setminus {}^\bullet t) \cup t^\bullet)$ if ${}^\bullet t \subseteq X$.
- $(w, X) \xrightarrow{\overline{s}} (w, X \setminus \{s\})$ if $s \in X$.
- $(w, \emptyset) \xrightarrow{\overline{A}} (r, \emptyset)$.

where ${}^\bullet t = \{s \mid s \in S,\ (s, t) \in F\}$ and $t^\bullet = \{s \mid s \in S,\ (t, s) \in F\}$ are the preset and postset of the transition t.

Now, given Petri nets P_1 and P_2, let M_1 and M_2 be the AFSMs constructed as above. Let $\rho = S_1 \cup S_2$, and let $\mathcal{L}$ be the set used to label transition in P_1 and P_2. First, suppose $L(P_1) \not\subseteq L(P_2)$. Then there is r such that $r \in L(P_1)$ and $r \notin L(P_2)$. Suppose $r = l_1.l_2.\cdots.l_n$. Let $r' = \overline{A}.l_1.\overline{A}.l_2.\overline{A}.\cdots.l_n.\overline{A}$. Then $r' \in L_A(M_1)\lceil\rho$, but $r' \notin L_A(M_2)\lceil\rho$, and hence $L_A(M_1)\lceil\rho \not\subseteq L_A(M_2)\lceil\rho$. Now, suppose $L(P_1) \subseteq L(P_2)$. Let

$$\begin{aligned} L_1 &= \{\overline{A}.l_1.\overline{A}.\cdots.l_n.\overline{A} \mid l_1.\cdots.l_n \in L(P_1)\} \\ L_2 &= \{\overline{A}.l_1.\overline{A}.\cdots.l_n.\overline{A} \mid l_1.\cdots.l_n \in L(P_2)\} \end{aligned}$$

It is clear from the construction of M_1 and M_2 that $L_A(M_1)\lceil\rho = [L_1]_\triangleright$ and $L_A(M_2)\lceil\rho = [L_2]_\triangleright$, where the relation $\triangleright$ is defined with respect to the alphabet $\mathcal{L} \cup \{A\}$. Now, since $L(P_1) \subseteq L(P_2)$, we have $L_1 \subseteq L_2$, which implies $[L_1]_\triangleright \subseteq [L_2]_\triangleright$ and hence $L_A(M_1)\lceil\rho \subseteq L_A(M_2)\lceil\rho$. □

Note that the undecidability of $\simeq_\rho$ implies undecidability of $\stackrel{\sqsubset}{\sim}_\rho$. Further, it is known that the language equality problem for labeled Petri nets is undecidable even for Petri nets with two unbounded places [44]. From this it can be easily shown that $\simeq_\rho$ is undecidable even for ρ such that $|\rho| = 2$.

5.5 Decidability Results for Must Testing

We first show that the parameterized must equivalence $\simeq_\rho^M$ is undecidable.

Theorem 5.7 *The parameterized asynchronous must equivalence $\simeq_\rho^M$ on FSMs is undecidable.*

Proof: We reduce the halting problem of counter machines to the given problem. We exploit Jancar's proof of undecidability of bisimilarity of Petri nets [44]. For a given counter machine C, Jancar constructs two labeled petri nets with the same underlying net, $P_1 = (S, T, F, \lambda, \mu_1)$ and $P_2 = (S, T, F, \lambda, \mu_2)$, which are bisimilar if and only if C does not halt. In particular, P_1 and P_2 have the following properties.

1. For all $t \in T$, $\lambda(t) \neq \tau$.
2. If C halts, then $L(P_1) \neq L(P_2)$.
3. If C does not halt, then P_1 and P_2 are bisimilar.

Let $\lambda : T \to L$, and $\sum = L \cup S \cup \{A\}$ for $A \notin L \cup S$. Let $M_1 = (Q, \sum \cup \overline{\sum}, \to, q_1, F)$ and $M_2 = (Q, \sum \cup \overline{\sum}, \to, q_2, F)$ be the FSMs constructed out of P_1 and P_2 as in the proof of Theorem 5.6. Actually, the nets that Jancar constructs have initial markings that may not be of the type assumed in the construction of the AFSMs. But our construction can be modified the obvious way to account for this; the AFSM initially performs an

appropriate number of output actions. Further, this modification can be done in such a way that, since P_1 and P_2 differ only in their initial marking, M_1 and M_2 differ only in their initial states.

We show that for $\rho = S$, $M_1 \simeq_\rho^M M_2$ if and only if C does not halt. From property 1 above, it follows that for every $r \in (\sum \cup \overline{\sum})^*$, $M_1 \downarrow_A r$ and $M_2 \downarrow_A r$. Now, we have two cases to consider:

- C halts: From property 2, $L(P_1) \neq L(P_2)$. Without loss of generality, we may assume, there is $l_1.\ldots.l_n \in L^*$ such that $l_1.\ldots.l_n \in L(P_1)$, $l_1.\ldots.l_{n-1} \in L(P_2)$, but $l_1.\ldots.l_n \notin L(P_2)$. Then, for $r = \overline{A}.l_1.\overline{A}.\ldots.l_{n-1}.\overline{A}.l_n$, we have M_1 $\underline{\mathsf{after}}$ r $\underline{\mathsf{m\not{u}st}}$ $\{\overline{l}_n\}$ but M_2 $\underline{\mathsf{after}}$ r $\underline{must}$ $\{\overline{l}_n\}$. Then by Theorem 5.3, $M_1 \not\simeq_\rho^M M_2$.

- C does not halt: From property 3, P_1 and P_2 are bisimilar. Suppose for some $r \in ((L \cup \{A\}) \cup \overline{(L \cup \{A\})})^*$ and $X \subseteq \overline{L \cup \{A\}}$, M_1 $\underline{\mathsf{after}}$ r $\underline{\mathsf{m\not{u}st}}$ X. We show M_2 $\underline{\mathsf{after}}$ r $\underline{\mathsf{m\not{u}st}}$ X. With a similar argument for the converse, it follows by Theorem 5.3 that $M_1 \simeq_\rho^M M_2$. By our assumption, $(q_1, \emptyset) \stackrel{r}{\Longrightarrow}_A (p_1, B_1)$ for some (p_1, B_1) such that (p_1, B_1) $\underline{\mathsf{m\not{u}st}}$ X. Then $X \cap \overline{B_1} = \emptyset$ and by Theorem 5.1, for some $r_1 \in (\sum \cup \overline{\sum})^*$, $q_1 \stackrel{r_1}{\Longrightarrow} p_1$ and $r_1 \triangleright r$. Moreover, as shown in the proof of Theorem 5.1, $B_1 = (\{|r|\}_i \cup \overline{\{|r_1|\}_o}) \setminus (\overline{\{|r|\}_o} \cup \{|r_1|\}_i)$. Further, since we know that $M_1 \downarrow_A r$, without loss of generality we may assume that $(p_1, B_1) \stackrel{\tau}{\not\longrightarrow}$. Now, since P_1 and P_2 are bisimilar, we have $L(P_1) = L(P_2)$. Then from the way M_1 and M_2 are constructed, it follows that there is $r_2 \in (\sum \cup \overline{\sum})^*$ such that $q_2 \stackrel{r_2}{\Longrightarrow} p_2$ and $r_2\lceil\rho = r_1\lceil\rho$. Then $(q_2, \emptyset) \stackrel{r}{\Longrightarrow}_A (p_2, B_2)$, for some B_2 such that for all $a \in L \cup \{A\}$, $\#(a, B_2) = \#(a, B_1)$. Then $X \cap \overline{B_2} = \emptyset$. Now, for all $\overline{l} \in X \cap \overline{L}$, it cannot be the case that $(p_2, B_2) \stackrel{\overline{l}}{\Longrightarrow}_A$, because $\overline{l} \notin B_2$ and M_2 never performs output actions in $\overline{L}$. Thus if $X \subset \overline{L}$ then it follows that (p_2, B_2) $\underline{\mathsf{m\not{u}st}}$ X and hence M_2 $\underline{\mathsf{after}}$ r $\underline{\mathsf{m\not{u}st}}$ X. On the other hand, suppose $\overline{A} \in X$. Then $A \notin B_2$, and hence the only way $(p_2, B_2) \stackrel{\overline{A}}{\Longrightarrow}_A$ is if $p_2 \stackrel{l}{\Longrightarrow}$ for some $l \in B_2 \cap L$. Now, since $(p_1, B_1) \stackrel{\tau}{\not\longrightarrow}$, we have $p_1 \stackrel{l}{\not\Longrightarrow}$. But then from $q_1 \stackrel{r_1}{\Longrightarrow} p_1 \stackrel{l}{\not\Longrightarrow}$, $q_2 \stackrel{r_2}{\Longrightarrow} p_2 \stackrel{l}{\Longrightarrow}$, and $r_1\lceil\rho = r_2\lceil\rho$, it would follow that the Petri nets P_1 and P_2 are not bisimilar, which

is a contradiction. Hence $(p_2, B_2) \overline{A}\!\!\not\Rightarrow$, and thus in all cases (p_2, B_2) $\underline{\not\mathsf{must}}$ X. This implies M_2 $\underline{\mathsf{after}}$ r $\underline{\not\mathsf{must}}$ X.

We have shown that $M_1 \simeq_\rho^M M_2$ if and only if the counter program C does not halt. Hence the asynchronous parameterized must equivalence is undecidable. □

Theorem 5.7 implies that the parameterized preorder $\overset{\sqsubset}{\sim}{}_\rho^M$ is also undecidable. The decidability of unparameterized must preorder $\overset{\sqsubset}{\sim}{}^M$ is still open. We end this section with a partial result in this direction.

For an FSM $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$ we define the asynchronous divergence language $L_a^\uparrow(M) = \{s \mid s \in (\sum \cup \overline{\sum})^*,\ q_0 \uparrow_A s\}$. By Theorem 5.3, given FSMs M_1 and M_2 with alphabet $\sum \cup \overline{\sum}$, a necessary condition for $M_1 \overset{\sqsubset}{\sim}{}^M M_2$ is that $L_a^\uparrow(M_2) \subseteq L_a^\uparrow(M_1)$. Note that, checking for this condition is an important problem by itself; by viewing M_2 as an implementation of the specification M_1, it checks that the implementation diverges only as allowed by the specification.

Now, we present a decision procedure which given M_1 and M_2, checks if $L_a^\uparrow(M_2) \subseteq L_a^\uparrow(M_1)$. We use the following lemma, which Rackoff proved for the more general case of Petri nets [75]. Since AFSMs are just a special class of Petri nets, this result is also applicable for AFSMs.

Lemma 5.6 (Rackoff [75]) *Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$ be an FSM of size n. Then for $q \in Q$, and a finite $B \in \{|\sum|\}$, $(q, B) \uparrow_A$, if and only if there is transition sequence*

$$(q, B) \xrightarrow{\tau}_A (q_1, B_1) \xrightarrow{\tau}_A \ldots \xrightarrow{\tau}_A (q_k, B_k) \xrightarrow{\tau}_A (q', B')$$

and $1 \leq i \leq k$ such that $q_i = q'$, $B_i \subseteq B'$ and $k \leq 2^{cn\ log\ n}$ for some constant c that is independent of n, q and B. □

We now reduce the problem of deciding containment of asynchronous divergence languages to the problem of containment of asynchronous languages, for which we gave a decision procedure in Section 5.4.

Lemma 5.7 *For an FSM $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$, there is a finite $B_0 \in \{|\sum|\}$ such that for all $q \in Q$, if $(q, B) \uparrow_A$ then $(q, B \cap B_0) \uparrow_A$.*

Proof: Let $n = |Q \cup \sum|$. We show the multiset $B_0 \in \{|\sum|\}$ such that $\#(a, B_0) = 2^{cn \ log \ n}$ for all $a \in \sum$, where c is the constant in Lemma 5.6, satisfies the property stated above. Now, suppose $(q, B) \uparrow_A$. Then by Lemma 5.6, there is a transition sequence $(q, B) \Longrightarrow_A (q', B_1) \Longrightarrow_A (q', B_2)$ of length $\leq 2^{cn \ log \ n}$ for some $B_2 \supseteq B_1$. Let $B' = B \cap B_0$, and $B'' = (B' \cup (B_1 \setminus B)) \setminus (B \setminus B_1)$. Then clearly, $(q, B') \Longrightarrow_A (q', B'') \Longrightarrow_A (q', B'' \cup (B_2 \setminus B_1))$, the length of which is the same as that of $(q, B) \Longrightarrow_A (q', B_2)$. Then again by Lemma 5.6, $(q, B') \uparrow_A$. □

Lemma 5.8 *For an FSM M, there is an FSM M' such that $L_a^{\uparrow}(M) = L_a(M')$.*

Proof: Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$. We construct $M' = (Q', \sum \cup \overline{\sum}, \rightarrow', q', F')$ that simulates M, and at any time can non-deterministically choose to examine the contents of its message buffer. If it finds the buffer to be large enough for M to be able to diverge from its current state, then M' jumps to an accepting state.

Specifically, let $Q = \{q_1, \ldots, q_n\}$. For each q_i, define the set $\mathcal{B}_i$ as follows. If $(q_i, B) \downarrow_A$ for every B then $\mathcal{B}_i = \emptyset$. Else $\mathcal{B}_i$ is the finite set $\{B_{i1}, \ldots, B_{ik}\}$ such that if $(q_i, B) \uparrow_A$ then $B \supseteq B_{ij}$ for some $B_{ij} \in \mathcal{B}_i$, and $B_{il} \subseteq B_{im}$ implies $B_{il} = B_{im}$. As a consequence of Lemma 5.7, we know that such a $\mathcal{B}_i$ exists. In fact, it can be computed as follows. Let B_0 be the multiset produced by Lemma 5.7. Enumerate all $B \subseteq B_0$, and check for each if $(q_i, B) \uparrow_A$ (Lemma 5.6 gives us a procedure for this), and let $\mathcal{B}_i$ be the set of all such minimal B's for which $(q_i, B) \uparrow_A$.

Let $Q' = \{(q_i, B) \mid B = \emptyset, \ or \ B \subseteq B' \ for \ some \ B' \in \mathcal{B}_i\} \cup \{f\}$ (note that $\mathcal{B}_i$ may be empty), $q' = (q_0, \emptyset)$, and $F' = \{f\}$. The transition function $\rightarrow'$ is defined as

$$
\begin{array}{lcll}
(q_i, \emptyset) & \stackrel{\alpha}{\longrightarrow}' & (q_j, \emptyset) & \text{if } q_i \stackrel{\alpha}{\longrightarrow} q_j \\
(q_i, B) & \stackrel{a}{\longrightarrow}' & (q_i, B \cup \{a\}) & \text{if } (B \cup \{a\}) \subseteq B' \text{ for some } B' \in \mathcal{B}_i \\
(q_i, B) & \stackrel{\tau}{\longrightarrow}' & f & \text{if } B = B' \text{ for some } B' \in \mathcal{B}_i \\
f & \stackrel{\overline{a}}{\longrightarrow}' & f & \text{for all } a \in \sum
\end{array}
$$

Note that the transition (sub)graph of M' with only the nodes $(q_i, \emptyset)$ is isomorphic to the transition graph of M. Thus M' can simulate M. But at any time, M' can non-deterministically choose to "examine" the message buffer contents. It is easy to check that $L_a(M') = L_a^{\uparrow}(M)$. □

Theorem 5.8 *There is an algorithm, which given FSMs M_1 and M_2, decides if $L_a^{\uparrow}(M_1) \subseteq L_a^{\uparrow}(M_2)$.*

Proof: An immediate consequence of Lemma 5.8 and Theorem 5.4. Note that the proof of Lemma 5.8 not only shows the existence of M', but also effectively constructs it. □

5.6 Discussion and Related Work

We have introduced a class of asynchronous infinite state systems called AFSMs, and related it to other asynchronous process models in the literature. We have investigated decidability of may and must testing equivalences over AFSMs. We have shown that the unparameterized may preorder is decidable over AFSMs. In comparison, decision procedures for may testing were previously known only for the simple class of finite state machines [45]. We have also shown that the parameterized may and must testing equivalences are undecidable over AFSMs. The decidability of unparameterized must testing is still open.

AFSMs relate to lossy channel systems investigated in [3, 4, 15]. Specifically, they can be viewed as finite control systems interacting with an unreliable (or noisy) buffer, where messages can be randomly lost to or received from the environment. But unlike in typical lossy channel systems where the message losses are invisible, message losses and additions are the only visible actions in AFSMs. The idea is that these transitions are viewed as interactions between the process and its environment, and we are only interested in the observable behavior of such open systems.

Lossy channel systems with ordered message deliveries such as the lossy FIFO channel systems [3] have been used for modeling and verifying communication protocols [4].

AFSMs (with unordered message deliveries) can serve as convenient abstractions for verification of such systems with ordered message deliveries [15]. Specifically, one can abstract systems with ordered message deliveries as asynchronously communicating finite state machines inorder to approximately decide verification problems which are either undecidable on the concrete system or for which there are no decision procedures. For instance, decision procedures are known for only checking the language containment of a lossy FIFO channel system and a *finite* state system [2], or more generally for model checking lossy systems with respect to sub-logics of μ-calculus [15]. In comparison, we have given decision procedures for checking language containment between two infinite state AFSMs. Further, AFSMs can be used to specify properties that are not regular and hence not expressible in μ-calculus. One such example is the language of an AFSM with a singleton alphabet, one state which is both an initial and a final state, and no transitions. The language of this machine is the set of all traces in which every prefix has at least as many inputs as outputs, which is not regular.

An important lesson we learn from our investigation is that asynchrony makes verification of testing equivalences extremely difficult. Specifically, both the unparameterized may and must testing are decidable in PSPACE over FSMs [45] which are the synchronous analogue of AFSMs. In comparison, by Theorem 5.5, deciding the unparameterized may preorder over AFSMs is EXPSPACE-hard. Further, the parameterized may and must testing over FSMs are also decidable in PSPACE roughly due to the following reason - comparing FSMs M_1 and M_2 according to a may or must preorder that is parameterized by a set of names ρ, is the same as comparing the FSMs by an unparameterized preorder after removing in both the FSMs all the transitions $\xrightarrow{\alpha}$ with $\alpha \in \rho \cup \overline{\rho}$. A similar situation does not hold for AFSMs.

Our investigations leave some problems open, of which the major ones are the following. First, although we have shown that deciding language containment for AFSMs is EXPSPACE-hard, we do not have a clear upper bound on its complexity. Moreover, the lower bound follows from showing a lower bound on a very special class of membership problem for AFSMs, which is in fact decidable in EXPSPACE-space. Thus, obtaining im-

proved upper and lower bounds for our problems is an important future exercise. In our paper, we consider three problems for AFSMs: membership, language containment and divergence language containment. Our reductions demonstrate that these problems are of increasing computational difficulty. In the absence of clear upper and lower bounds, even investigating the complexity of these problems relative to each other would be a useful next step. Another interesting direction to explore would be look at the model checking problem for AFSMs with respect to modal logics, in the vein of [17, 15]. These problems may be amenable to tighter complexity analysis.

Another problem of interest for future research is deciding bisimilarity of AFSMs. We conjecture that bisimilarity is decidable over AFSMs, and this is supported by the fact that language equivalence (which is coarser than bisimilarity) is decidable over AFSMs (Theorem 5.4). In contrast, note that bisimilarity is undecidable over the more expressive asynchronous model of MSA's [17].

Chapter 6

Executable Specification in Maude

In this chapter, we give an executable specification of asynchronous π-calculus [92] and the Actor model, and the may testing preorder between finitary (non-recursive) processes in these models. Specifically, we specify the π-calculus and the Actor model as theories in rewriting logic, and use the Maude tool that supports rewriting logic, for executing these specifications.

We consider the variant of asynchronous π-calculus with both match and mismatch capabilities on names (Section 2.1), and specify its labeled transition semantics in rewriting logic. This specification uses conditional rewrite rules with rewrites in conditions and the CINNI calculus [86] for managing names and bindings in the π-calculus. We then specify the type system described in Sections 3.2.1 and 3.4, in membership equational logic, to obtain an executable specification of $A\pi_{\neq}$, which is the variant of Actor model described in Section 3.4.

We then obtain an executable specification of may preorder over finitary processes in both these calculi using the metalevel facilities in Maude. We use the metalevel facilities to compute the set of all traces exhibited by a given process, and use this to decide the may preorder between processes by comparing their trace sets according to the alternate characterization of may testing presented in Theorem 2.1 (Section 2.1) and Theorem 3.2 (Section 3.4). Note that the may preorder over the variant of asynchronous π-calculus is unparameterized, whereas the preorder over $A\pi_{\neq}$ is parameterized. The techniques

presented in this Chapter can be easily adapted to obtain an executable specification of L$\pi_{=}$ (Section 2.2), L$\pi^{-}_{=}$(Section 2.4), Lπ (Section 2.3), and the parameterized may preorder over them.

Following is the layout of this chapter. Section 6.1.1 describes the specification of the syntax of asynchronous π-calculus, together with the corresponding CINNI operations we use. Section 6.1.2 describes the operational semantics specified by means of conditional rewrite rules. Section 6.1.3 presents the specification of trace semantics, and Section 6.2 contains the specification of the may preorder over asynchronous π-calculus. Section 6.3 describes the specification of the type system for A$\pi_{\neq}$, and the parameterized may preorder over it. Section 6.4 concludes the paper with a discussion of related research and directions for further work.

6.1 Specification of Asynchronous π-Calculus

We consider the variant of asynchronous π-calculus with both match and mismatch capability of names, that is described at the end of Section 2.1.

6.1.1 Syntax

The sort `Chan` is used to represent channel names and each of the non-constant syntax constructors is declared as `frozen`, so that the corresponding arguments cannot be rewritten by rules; this will be justified at the end of Section 6.1.2.

```
sort Chan .
sorts Guard Trm .


op  _(_) : Chan Qid -> Guard .
op  nil  :  -> Trm .
op  _<_>  :  Chan Chan -> Trm [frozen] .
op  _._  :  Guard Trm -> Trm [frozen] .
op  _|_  :  Trm Trm -> Trm [frozen assoc comm] .
op  new[_]_  :  Qid Trm -> Trm [frozen] .
```

```
op  if_=_then_else_fi  :  Chan Chan Trm Trm -> Trm [frozen] .
op  !_  :  Trm -> Trm [frozen] .
```

To represent substitution on π-calculus processes (and traces, see Section 6.1.3) at the language level we use CINNI as a calculus for explicit substitutions [86]. This gives a first-order representation of terms with bindings and capture-free substitutions, instead of going to the metalevel to handle names and bindings. The main idea in such a representation is to keep the bound names inside the binders as it is, but to replace its use by the name followed by an index which is a count of the number of binders with the same name it jumps before it reaches the place of use. Following this idea, we define terms of sort `Chan` as indexed names as follows.

```
op  _{_} : Qid Nat -> Chan [prec 1] .
```

We introduce a sort of substitutions `Subst` together with the following operations:

```
op [_:=_] : Qid Chan -> Subst .
op [shiftup_] : Qid -> Subst .
op [shiftdown_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
```

The first two substitutions are basic substitutions representing *simple* and *shiftup* substitutions; the third substitution is a special case of *simple* substitution; the last one represents complex substitution where a substitution can be lifted using the operator `lift`. The intuitive meaning of these operations is described in Table 6.1 (see [86] for more details). Using these, explicit substitutions for π-calculus processes are defined equationally. Some interesting equations are the following:

```
vars  CX CY : Chan .
eq  S (CX(Y) . P ) = (S CX)(Y) . ([lift Y S] P) .
eq  S (new [X] P) = new [X] ([lift X S] P) .
```

[a := x]	[shiftup a]	[shiftdown a]	[lift a S]
a{0} ↦ x	a{0} ↦ a{1}	a{0} ↦ a{0}	a{0} ↦ [shiftup a] (S a{0})
a{1} ↦ a{0}	a{1} ↦ a{2}	a{1} ↦ a{0}	a{1} ↦ [shiftup a] (S a{1})
...	...	...	...
a{n+1} ↦ a{n}	a{n} ↦ a{n+1}	a{n+1} ↦ a{n}	a{n} ↦ [shiftup a] (S a{n})
b{m} ↦ b{m}	b{m} ↦ b{m}	b{m} ↦ b{m}	b{m} ↦ [shiftup a] (S b{m})

Table 6.1: The CINNI operations.

6.1.2 Operational Semantics

We define the sort `Action` and the corresponding operations as follows:

```
sorts  Action ActionType .
ops i o : -> ActionType .
op  f : ActionType Chan Chan -> Action .
op  b : ActionType Chan Qid -> Action .
op  tauAct : -> Action .
```

The operators `f` and `b` are used to construct free and bound actions respectively. Name substitution on actions is defined equationally as expected.

The inference rules in Table 2.1 are modeled as conditional rewrite rules with the premises as conditions of the rule.[1] Since rewrites do not have labels unlike the labeled transitions, we make the label a part of the resulting term; thus rewrites corresponding to transitions in the operational semantics are of the form $P \Rightarrow \{\alpha\}Q$.

Because of the *INP* and *OPEN* rules, the transitions of a term can be infinitely branching. Specifically, in case of the *INP* rule there is one branch for every possible name that can be received in the input. In case of the *OPEN* rule, there is one branch for every name that is chosen to denote the private channel that is being emitted (note that the transition rules are defined only modulo α-equivalence). To overcome this problem, we define transitions over pairs of the form `[CS] P`, where `CS` is a set of channel names containing all the names that the environment with which the process interacts, knows about. The set `CS` expands during bound input and output interactions when private names are exchanged between the process and its environment.

[1] The symmetric versions missing in the table need not be implemented because the process constructors `_+_` and `_|_` have been declared as commutative.

The infinite branching due to the *INP* rule is avoided by allowing only the names in the environment set `CS` to be received in free inputs. Since `CS` is assumed to contain all the free names in the environment, an input argument that is not in `CS` would be a private name of the environment. Now, since the identifier chosen to denote the fresh name is irrelevant, all bound input transitions can be identified to a single input. With these simplifications, the number of input transitions of a term become finite. Similarly, in the *OPEN* rule, since the identifier chosen to denote the private name emitted is irrelevant, instances of the rule that differ only in the chosen name are not distinguished.

We discuss in detail the implementation of only a few of the inference rules; the reader is referred to Appendix B for a complete list of all the rewrite rules for Table 2.1.

```
sorts EnvTrm TraceTrm .
subsort EnvTrm < TraceTrm .
op  [_]_ : Chanset Trm -> EnvTrm [frozen] .
op  {_}_ : Action TraceTrm -> TraceTrm [frozen] .
```

Note that the two operators are also declared above with the `frozen` attribute, forbidding in this way rewriting of their arguments, as justified at the end of this section.

The following non-conditional rule is for free inputs.

```
rl [Inp] : [CY CS] ((CX(X) . P) + SUM) =>
              {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .
```

The next rule we consider is the one for bound inputs. Since the identifier chosen to denote the bound argument is irrelevant, we use the constant `'U` for all bound inputs, and thus `'U{0}` denotes the fresh channel received. Note that in contrast to the *BINP* rule of Table 2.1, we do not check if `'U{0}` is in the free names of the process performing the input, and instead we shift up the channel indices appropriately, in both the set of environment names `CS` and the process `P` in the righthand side and condition of the rule. This is justified because the transition target is within the scope of the bound name in the input action. Note also that the channel `CX` in the action is not shifted down because it is out of the scope of the bound argument. The set of environment names is expanded

by adding the received channel `'U{0}` to it. Finally, we use a special constant `flag` of sort `Chan`, to ensure termination. We add an instance of `flag` to the environment set of the rewrite in condition, so that the *BINP* rule is not fired again while evaluating the condition. Without this check, we will have a non-terminating execution in which the *BINP* rule is repeatedly fired.

```
crl [BInp] : [CS] P => {b(i,CX,'U)} ['U{0} [shiftup 'U] CS] P1
             if (not flag in CS) /\
                CS1 := flag 'U{0} [shiftup 'U] CS /\
                [CS1] [shiftup 'U] P => {f(i,CX,'U{0})} [CS1] P1 .
```

The following rule treats the case of bound outputs.

```
crl [Open] : [CS] (new [X] P) => {[shiftdown X] b(o,CY,X)} [X{0} CS1] P1
             if CS1 := [shiftup X] CS /\
                [CS1] P => {f(o,CY,X{0})} [CS1] P1 /\ X{0} =/= CY .
```

Like in the case of bound inputs, we identify all bound outputs to a single instance in which the identifier `X` that appears in the restriction is chosen as the bound argument name. Note that in both the righthand side of the rule and in the condition, the indices of the channels in `CS` are shifted up, because they are effectively moved across the restriction. Similarly, the channel indices in the action in the righthand side of the rule are shifted down since the action is now moved out of the restriction. Note also that the exported name is added to the set of environment names, because the environment that receives this exported name can use it in subsequent interactions.

The *PAR* inference rule is implemented by two rewrite rules, one for the case where the performed action is free, and the other where the action is bound. The rewrite rule for the latter case is discussed next, while the one for the former case is simpler and appears in the appendix.

```
var IO : ActionType
crl [Par] : [CS] (P | Q) =>
               {b(IO,CX,Y)} [Y{0} ([shiftup Y] CS)] (P1 | [shiftup Y] Q)
            if [CS] P => {b(IO,CX,Y)} ([CS1] P1) .
```

Note that the side condition of the *PAR* rule in Table 2.1, which avoids confusion of the emitted bound name with free names in `Q`, is achieved by shifting up channel indices in `Q`. This is justified because the righthand side of the rule is under the scope of the bound output action. Similarly, the channel indices in the environment are also shifted up. Further, the set of environment names is expanded by adding the exported channel `Y{0}`.

Finally, we consider the rewrite rule for *CLOSE*. The process `P` emits a bound name `Y`, which is received by process `Q`. Since the scope of `Y` after the transition includes `Q`, the rewrite involving `Q` in the second condition of the rule is carried out within the scope of the bound name that is emitted. This is achieved by adding the channel `Y{0}` to the set of environment names and shifting up the channel indices in both `CS` and `Q` in the rewrite. Note that since the private name being exchanged is not emitted to the environment, we neither expand the set `CS` in the righthand side of the rule nor shift up the channel indices in it.

```
crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
              if [CS] P => {b(o,CX,Y)} [CS1] P1 /\
                 [Y{0} [shiftup Y] CS] [shiftup Y] Q =>
                                  {f(i,CX,Y{0})} [CS2] Q1 .
```

We conclude this section with the following note. The operator {_}_ is declared `frozen` because further rewrites of the process term encapsulated in a term of sort `TraceTrm` are useless. This is because all the conditions of the transition rules only involve one step rewrites (the righthand side of these rewrites can only match a term of sort `TraceTrm` with a single action prefix). Further note that, to prevent rewrites of a term to a non well-formed term, all the constructors for π-calculus terms (Section 6.1.1) have been declared `frozen`; in the absence of this declaration we would have for instance rewrites of the form `P | Q => {A}.P1 | Q` to a non well-formed term.

6.1.3 Trace Semantics

We introduce a sort `Trace` as supersort of `Action` to specify traces.

```
sorts  Trace  TTrace .
subsort  Action < Trace .
op  epsilon : -> Trace .
op  _._ : Trace Trace -> Trace [assoc id: epsilon] .
op  [_] : Trace -> TTrace .
```

We define the operator `[_]` to represent a complete trace. The motivation for doing so is to restrict the equations and rewrite rules defined over traces to operate only on a complete trace instead of a part of it. The following equation defines α-equivalence on traces. Note that in a trace `TR1.b(IO,CX,Y).TR2` the action `b(IO,CX,Y)` binds the identifier `Y` in `TR2`.

```
ceq [TR1 . b(IO,CX,Y) . TR2] =
       [TR1 . b(IO,CX,'U) . [Y := 'U{0}] [shiftup 'U] TR2]
    if Y =/= 'U .
```

Because the operator `op {_}_ :  Action TraceTrm -> TraceTrm` is declared as `frozen`, a term of sort `EnvTrm` can rewrite only once, and so we cannot obtain the set of finite traces of a process by simply rewriting it multiple times in all possible ways. The problem is solved as in [96], by specifying the trace semantics using rules that generate the transitive closure of one step transitions as follows:

```
sort TTrm .
op  [_] : EnvTrm -> TTrm [frozen] .
var TT : TraceTrm .

crl [reflx] : [ P ] => {A} Q if P => {A} Q .
crl [trans] : [ P ] => {A} TT
              if P => {A} Q /\ [ Q ] => TT /\ [ Q ] =/= TT .
```

We use the operator `[_]` to prevent infinite loops while evaluating the conditions of the rules above. If this operator were not used, then the lefthand side of the rewrite in the condition would match the lefthand side of the rule itself, and so the rule itself

could be used in order to solve its condition. This operator is also declared as `frozen` to prevent useless rewrites inside `[_]`.

We can now use the `search` command of Maude 2.0 to find all possible traces of a process. The traces appear as prefix of the one-step successors of a `TTrm` of the form `[[CS] P]`. For instance, the set of all traces exhibited by `[mt] new ['y] ('x0 < 'y0 > | 'x0('u) . nil)` (where `mt` denotes the empty channel set), can be obtained by using the following `search` command.

```
Maude> search [ [mt] new ['y] ('x{0} < 'y{0} > | 'x{0}('u) . nil) ] =>!
X:TraceTrm .
search in APITRACESET : [[mt]new['y]('x{0} < 'y{0} > | 'x{0}('u) . nil)] =>!
X:TraceTrm .

Solution 1 (state 1)
states: 7  rewrites: 17344 in 110ms cpu (150ms real) (157672 rewrites/second)
X:TraceTrm --> {b(i, 'x{0}, 'u)}['u{0}]new['y](nil | 'x{0} < 'y{0} >)

Solution 2 (state 2)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {tauAct}[mt]new['y](nil | nil)

Solution 3 (state 3)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}['y{0}]nil | 'x{0}('u) . nil

Solution 4 (state 4)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(i, 'x{0}, 'u)}{b(o, 'x{0}, 'y)}['y{0} 'u{0}]nil | nil

Solution 5 (state 5)
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}{b(i, 'x{0}, 'u)}['y{0} 'u{0}]nil | nil

Solution 6 (state 6)
```

```
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
X:TraceTrm --> {b(o, 'x{0}, 'y)}{f(i, 'x{0}, 'y{0})}['y{0}]nil | nil

No more solutions.
states: 7  rewrites: 17344 in 110ms cpu (170ms real) (157672 rewrites/second)
```

The command returns all `TraceTrm`s that can be reached from the given `TTrm`, and that are terminating (the '`!`' in `=>!` specifies that the target should be terminating). The required set of traces can be obtained by simply extracting from each solution `{a1}...{an}TT` the sequence `a1...an` and removing all `tauAct`s in it. Thus, we have obtained an executable specification of the trace semantics of asynchronous π-calculus.

6.2 Specification of the May Preorder

We encode the trace preorder defined by the laws *L1, L2* and *L3* in Table 2.2 as rewrite rules on terms of the sort `TTrace` of complete traces. Note that *L4* is not required for the variant of asynchronous π-calculus with mismatch capability on names.

The relation $r \prec s$ *if cond*, is encoded as `s => r if cond`. The reason for this form of representation will be justified soon. The function $(\{y\})\cdot$ on traces is defined equationally by the operation `bind`. The constant `bot` of sort `Trace` is used by the bind operation to signal error.

```
op  bind : Qid Trace -> Trace .
op  bot  : -> Trace .
var TR  : Trace .    var  IO  : ActionType.

ceq TR . bot = bot  if TR =/= epsilon .
ceq bot . TR = bot  if TR =/= epsilon .

eq  bind(X , epsilon) = epsilon .

eq  bind(X , f(i,CX,CY) . TR ) = if CX =/= X{0} then
           if CY == X{0} then ([shiftdown X] b(i, CX , X)) . TR
```

```
              else ([shiftdown X] f(i, CX , CY)) . bind(X , TR) fi
                 else bot fi .

eq  bind(X , b(IO,CX,Y) . TR) =  if CX =/= X{0} then
           if X =/= Y then ([shiftdown X] b(i, CX , Y)) . bind(X , TR)
              else ([shiftdown X] b(IO, CX , Y)) . bind(X , swap(X,TR)) fi
                 else bot fi .
```

The equation for the case where the second argument to `bind` begins with a free output is not shown as it is similar. Note that the channel indices in actions until the first occurrence of `X{0}` as the argument of a free input are shifted down as these move out of the scope of the binder `X`. Further, when a bound action with `X` as the bound argument is encountered, the `swap` operation is applied to the remaining suffix of the trace. The swap operation simply changes the channel indices in the suffix so that the binding relation is unchanged even as the binder `X` is moved across the bound action. This is accomplished by simultaneously substituting `X{0}` with `X{1}`, and `X{1}` with `X{0}`. Finally, note that when `X{0}` is encountered as the argument of a free input, the input is converted to a bound input. If `X{0}` is first encountered at any other place, an error is signaled by returning the constant `bot`.

The encoding of the preorder relation on traces is now straightforward.

```
crl [Drop] : [ TR1 . b(i,CX,Y) . TR2 ] => [ TR1 . bind(Y , TR2) ]
             if bind(Y , TR2) =/= bot .

rl  [Delay] : [ ( TR1 . f(i,CX,CY) . b(IO,CU,V) . TR2 ) ] =>
              [ ( TR1 . b(IO,CU,V) . ([shiftup V] f(i, CX , CY)) . TR2 ) ] .

crl [Delay] : [ ( TR1 . b(i,CX,Y) . f(IO,CU,CV) . TR2 ) ] =>
                 [ ( TR1 . bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) ) ]
              if bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) =/= bot .

crl [Annihilate] : [ ( TR1 . b(i,CX,Y) . f(o,CX,Y{0}) . TR2 ) ] =>
                      [ TR1 . bind(Y , TR2) ]
                   if bind(Y , TR2) =/= bot .
```

Note that in the first `Delay` rule, the channel indices of the free input action are shifted up when it is delayed across a bound action, since it gets into the scope of the bound argument. Similarly, in the second `Delay` rule, when the bound input action is delayed across a free input/output action, the channel indices of the free action are shifted down by the `bind` operation. The other two subcases of the `Delay` rule, namely, where a free input is to be delayed across a free input or output, and where a bound input is to be delayed across a bound input or output, are not shown as they are similar. Similarly, for `Annihilate`, the case where a free input is to be annihilated with a free output is not shown.

We now describe our implementation of verification of the may preorder between finite processes, i.e. processes without replication, by exploiting the trace-based characterization of may testing discussed in Section 6.2. Let $[\![P]\!]$ be the set of traces exhibited by the process P, and define $[\![P]\!] \preceq [\![Q]\!]$ if for every $s \in [\![Q]\!]$ there is $r \in [\![P]\!]$ such that $r \preceq s$, where $\preceq$ is the reflexive transitive closure of the laws *L1, L2* and *L3* in Table 2.2. Recall from Section 2.1 that $P \stackrel{\sqsubset}{\sim} Q$ if and only if $[\![Q]\!] \preceq [\![P]\!]$.

The finiteness of a process P only implies that the length of traces in $[\![P]\!]$ is bounded, but the number of traces in $[\![P]\!]$ can be infinite (even modulo α-equivalence) because the *INP* rule is infinitely branching. To avoid the problem of having to compare infinite sets, we observe that

$$[\![Q]\!] \precsim [\![P]\!] \quad \text{if and only if} \quad [\![Q]\!]_{fn(P,Q)} \precsim [\![P]\!]_{fn(P,Q)},$$

where for a set of traces S and a set of names ρ we define $S_\rho = \{s \in S \mid fn(s) \subseteq \rho\}$. Now, since the traces in $[\![P]\!]$ and $[\![Q]\!]$ are finite in length, it follows that the sets of traces $[\![P]\!]_{fn(P,Q)}$ and $[\![Q]\!]_{fn(P,Q)}$ are finite modulo α-equivalence. In fact, the set of traces generated for `[[fn(P,Q)] P]` by our implementation described in Section 6.1.2, contains exactly one representative from each α-equivalence class of $[\![P]\!]_{fn(P,Q)}$.

Given processes P and Q, we generate the set of all traces (modulo α-equivalence) of `[[fn(P,Q)] P]` and `[[fn(P,Q)] Q]` using the metalevel facilities of Maude 2.0. As

mentioned in Section 6.1.3, these terms, which are of sort `TTrm`, can be rewritten only once. The term of sort `TraceTrm` obtained by rewriting contains a finite trace as a prefix. To create the set of all traces, we compute all possible one-step rewrites. This computation is done at the metalevel by the function `TTrmtoNormalTraceSet` that uses two auxiliary functions `TTrmtoTraceSet` and `TraceSettoNormalTraceSet`.

```
op  TTrmtoTraceSet : Term -> TermSet .
op  TraceSettoNormalTraceSet : TermSet -> TermSet .
op  TTrmtoNormalTraceSet : Term -> TermSet .

eq  TTrmtoNormalTraceSet(T) = TraceSettoNormalTraceSet(TTrmtoTraceSet(T)) .
```

The function `TTrmTraceSet` uses the function `allOneStepAux(T,N)` that returns the set of all one-step rewrites (according to the rules in Sections 6.1.2 and 6.1.3, which are defined in modules named `PISEMANTICS` and `PITRACE`, of the term `T` which is the metarepresentation of a term of sort `TTrm`, skipping the first `N` solutions. In the following equations, the operator `_u_` stands for set union.

Notice the use of the operation `metaSearch`, which receives as arguments the metarepresented module to work in, the starting term for search, the pattern to search for, a side condition (empty in this case), the kind of search (which may be `'*` for zero or more rewrites, `'+` for one or more rewrites, and `'!` for only matching normal forms), the depth of search, and the required solution number. It returns the term matching the pattern, its type, and the substitution produced by the match; to keep only the term, we use the projection `getTerm`.

```
op  PITRACE-MOD : -> Module .
eq  PITRACE-MOD = ['PITRACE] .
var N : MachineInt .    vars T X : Term .

op  allOneStepAux : Term MachineInt Term -> TermSet .
op  TraceTermToTrace : Term -> Term .

eq  TTrmtoTraceSet(T) = allOneStepAux(T,0,'X:TraceTrm) .
```

```
eq  allOneStepAux(T,N,X) =
    if metaSearch(PITRACE-MOD,T,X,nil,'+,1,N) == failure
    then 'epsilon.Trace
    else TraceTermToTrace(getTerm(metaSearch(PITRACE-MOD,T,X,nil,'+,1,N)))
         u allOneStepAux(T,N + 1,X) fi .
```

The function `TraceTrmToTrace` (whose equations are not shown), used in `allOneStepAux`, extracts the trace `a1.a2...an` out of a metarepresentation of a term of sort `TraceTrm` of the form `{a1}{a2}...{an}TT`. The function `TraceSettoNormalTraceSet` uses the metalevel operation `metaReduce` to convert each trace in a trace set to its α-normal form. The operation `metaReduce` takes as arguments a metarepresented module and a metarepresented term in that module, and returns the metarepresentation of the fully reduced form of the given term using the equations in the given module, together with its corresponding sort or kind. Again, the projection `getTerm` leaves only the resulting term.

```
eq  TraceSettoNormalTraceSet(mt) = mt .
eq  TraceSettoNormalTraceSet(T u TS) =
       getTerm(metaReduce(TRACE-MOD,'`[_`] [ T ]))
       u TraceSettoNormalTraceSet(TS) .
```

We implement the relation $\precsim$ on sets defined in Section 6.2 as the predicate `<<`. We check if $P \stackrel{\sqsubset}{\sim} Q$ by computing this predicate on the metarepresented trace sets $[\![P]\!]_{fn(P,Q)}$ and $[\![Q]\!]_{fn(P,Q)}$ as follows. For each (metarepresented) trace `T` in $[\![P]\!]_{fn(P,Q)}$, we compute the reflexive transitive closure of `T` with respect to the laws shown in Table 2.2. The laws are implemented as rewrite rules in the module `TRACE-PREORDER`. We then use the fact that $[\![Q]\!]_{fn(P,Q)} \precsim [\![P]\!]_{fn(P,Q)}$ if and only if for every trace `T` in $[\![P]\!]_{fn(P,Q)}$ the closure of `T` and $[\![Q]\!]_{fn(P,Q)}$ have a common element.

```
op TRACE-PREORDER-MOD : -> Module .
eq TRACE-PREORDER-MOD = ['TRACE-PREORDER] .
var  N : MachineInt .    vars T T1 T2 X : Term .
var  TS TS1 TS2 : TermSet .
```

```
op  _<<_ : TermSet TermSet -> Bool .
op  _<<<_ : TermSet Term -> Bool .
op  TTraceClosure : Term -> TermSet .
op  TTraceClosureAux : Term Term MachineInt -> TermSet .
op  _maypre_  :  Term Term -> Bool .
eq  TS2 << mt = true .
eq  TS2 << (T1 u TS1) = TS2 <<< T1 and TS2 << TS1 .
eq  TS2 <<< T1 = not disjoint?(TS2 , TTraceClosure(T1)) .
eq  T1 maypre T2 = TTrmtoNormalTraceSet(T2) << TTrmtoNormalTraceSet(T1) .
```

The computation of the closure of `T` is done by the function `TTraceClosure`. It uses `TTraceClosureAux` to compute all possible (multi-step) rewrites of the term `T` using the rules defined in the module `TRACE-PREORDER`, again by means of the metalevel operation `metaSearch`.

```
eq  TTraceClosure(T) = TTraceClosureAux(T,'TT:TTrace,0) .
eq  TTraceClosureAux(T,X,N) =
    if metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N) == failure
    then mt
    else getTerm(metaSearch(TRACE-PREORDER-MOD,T,X,nil,'*,maxMachineInt,N))
         u TTraceClosureAux(T,X,N + 1) fi .
```

This computation is terminating as the number of traces to which a trace can rewrite using the trace preorder laws is finite modulo α-equivalence. This follows from the fact that the length of a trace is non-increasing across rewrites, and the free names in the target of a rewrite are also free names in the source. Since the closure of a trace is finite, `metaSearch` can be used to enumerate all the traces in the closure. Note that although the closure of a trace is finite, it is possible to have an infinite rewrite that loops within a subset of the closure. Further, since `T` is a metarepresentation of a trace, `metaSearch` can be applied directly to `T` inside the function `TTraceClosureAux(T,X,N)`.

We end this section with a small example, which checks for the may-testing preorder between the processes $P = a(u).b(v).(\nu w)(\overline{w}v|\overline{a}u) + b(u).a(v).(\overline{b}u|\overline{b}w)$ and $Q =$

$b(u).(\overline{b}u|\overline{b}w)$. We define constants `TP` and `TQ` of sort `TTrm`, along with the following equations:

```
eq TP = [['a{0} 'b{0} 'w{0}]
         'a{0}('u) . 'b{0}('v) . new['w]('w{0} < 'v{0} > | 'a{0} < 'u{0} >)
             + 'b{0}('u) . 'a{0}('v) . ('b{0} < 'u{0} > | 'b{0} < 'w{0} >)]

eq TQ = [['a{0} 'b{0} 'w{0}]
         'b{0}('u) . ('b{0} < 'u{0} > | 'b{0} < 'w{0} >)]
```

The metarepresentation of these `TTrm`s can now be obtained by using `'TP.TTrm` and `'TQ.TTrm`, and we can then check for the may-testing preorder between the given processes as follows:

```
Maude> red 'TP.TTrm maypre 'TQ.TTrm .
reduce in PITRACESET : 'TP.TTrm maypre 'TQ.TTrm .
rewrites: 791690 in 2140ms cpu (2160ms real) (361422 rewrites/second)
result Bool: true
Maude> red 'TQ.TTrm maypre 'TP.TTrm .
reduce in PITRACESET : 'TQ.TTrm maypre 'TP.TTrm .
rewrites: 664833 in 1620ms cpu (1640ms real) (410390 rewrites/second)
result Bool: false
```

Thus, we have $P \stackrel{\sqsubset}{\precsim} Q$, but $Q \stackrel{\sqsubset}{\not\precsim} P$. The reader can check that indeed, $[\![Q]\!]_{fn(P,Q)} \precsim [\![P]\!]_{fn(P,Q)}$, but $[\![P]\!]_{fn(P,Q)} \not\precsim [\![Q]\!]_{fn(P,Q)}$.

6.3 Specification of A$\pi_{\neq}$

We now give an executable specification of A$\pi_{\neq}$ (Section 3.4) and of the parameterized may preorder over it. We first extend the syntax of asynchronous π-calculus specified in Section 6.1.1 with recursive definitions, and specify the type system that captures the Actor primitives. We then extend the semantics specified in Section 6.1.2 by adding a new rewrite rule for behavior instantiations, and finally adapt the techniques presented in Section 6.2 to implement the parameterized may preorder.

We introduce the sort `Defn` for recursive definitions and the sort `Context` to represent a collection of definitions.

```
sorts  Defn Context .
subsort Defn < Context .


op  _:=(_;_)_  :  Qid QidTuple QidTuple Term -> Defn [prec 8] .
op  emptycontext  : -> Context .
op  _,_  :  Context Context -> Context [assoc id: emptycontext prec 9] .
op  context : -> Context .
```

The constant `context` is used to keep all the recursive definitions that can be used in specifying processes. This syntax for recursive definitions and some useful operations on them are defined in the module APICONTEXT shown in Appendix B.

Behavior instantiations and name substitutions on instantiations are defined as follows.

```
op  _<_;_>  :  Qid ChanTuple ChanTuple -> Term [prec 2] .
eq  S B < t1 ; t2 > = B < S t1 ; S t2 > .
```

We now specify the type system for A$\pi_{\neq}$. The function $f : \rho \rightarrow \rho^*$ that is used to keep track of the temporary names of actors (see Section 3.2.1) is represented as a set of pairs; the pair (x, y) denoting the fact that the actor y has temporarily assumed the name x. We introduce the sort `NameMap` to represent these name mapping functions.

```
sorts  LiftChan Pair NameMap.
subsort Chan < LiftChan .


op * : -> LiftChan .
op bot : -> LiftChan .
op (__) : Chan LiftChan -> Pair .
```

The constructors for `NameMap`, which are not shown above, are the usual set constructors. Following is the specification of the operator $\oplus$ over name maps, that was introduced in Definition 3.1.

```
op  _oplus_  :  NameMap NameMap -> NameMap .
op  delete  :  Chan NameMap -> NameMap .
op  domain  :  NameMap -> Chanset .

eq  emptymap oplus M = M .
eq  { ( X  X' ) } oplus M = if (X' =/= bot or not X in domain(M))
                            then { ( X X' ) } cup delete(X , M)
                            else M fi .
eq  { ( X  X' ) , E } oplus M = if (X' =/= bot or not X in domain(M))
                                then { ( X X' ) } cup ({ E } oplus delete(X , M))
                                else { E } oplus M fi .

eq  delete(X , emptymap) = emptymap .
eq  delete(X , { ( Y Y' ) }) = if X == Y then emptymap else { ( Y Y' ) } fi .
eq  delete(X , { ( Y Y' ) , E }) = delete(X , { ( Y Y' ) }) cup delete(X , { E }) .

eq  domain(emptymap) = emptymap .
eq  domain({ ( X X' ) }) = { X } .
eq  domain({ ( X X' ) , E }) = { X } cup domain({ E }) .
```

The following operators represent the other functions on name maps, that are introduced in Definition 3.1.

```
op  hide  :  Qid NameMap -> NameMap .
op  compatible?  :  NameMap NameMap -> Bool .
```

Recall that for a well typed A$\pi_{\neq}$ term P, there is a unique set of names ρ and a unique name mapping function f such that $\rho; f \vdash P$. The set ρ contains the receptionist names and f keeps track of temporary names of actors in P. We now define operations that compute these ρ and f for a given P, assuming that P is well typed.

```
op  recep  :  Term -> Chanset .
op  chanMap  :  Term -> NameMap .
op  chain : ChanTuple -> ChanMap .
```

```
eq  recep(nil) = emptychanset .
eq  recep(CX < CY >) = emptychanset .
eq  recep(CX ( X ) . P) = { CX } cup down(X , recep(P)) .
eq  recep(P | Q) = recep(P) cup recep(Q) .
eq  recep(new [ X ] P) = down(X , recep(P)) .
eq  recep([ CX = CY ] ( P , Q )) = recep(P) cup recep(Q) .
eq  recep(B < t1 ; t2 >) = toset(t1) .

eq  chanMap(nil) = emptymap .
eq  chanMap(CX < CY >) = emptymap .
eq  chanMap(CX ( X ) . P) =
                 if | down(X , recep(P)) \ { CX } | == 0  then { ( CX bot ) }
                 else { ( CX pick(down(X , recep(P)) \ { CX }) ) ,
                        ( pick(down(X , recep(P)) \ { CX }) bot ) } fi .
eq  chanMap(P | Q) = chanMap(P) oplus chanMap(Q) .
eq  chanMap(new [ X ] P) = hide(X , chanMap(P)) .
eq  chanMap(if CX = CY then P else Q fi)) = chanMap(P) oplus chanMap(Q) .
eq  chanMap(B < t1 ; t2 >) = chain(t1) .
```

The operator `chain` implements the function defined in Definition 3.2. For each channel `Y{n}` in the set `CS`, `down(X,CS)` decrements the index `n` if `Y` = `X`, and leaves `Y{n}` unchanged otherwise. The function `pick(CS)` returns some element in `CS`.

We are now ready to specify the type rules. We introduce a sort `ActorConfig` for well typed terms. The type rules are all encoded as conditional membership axioms.

```
sort ActorConfig .
subsort ActorConfig < Term .

op  chainCondition  :  NameMap Chan Chanset -> Bool .

mb  nil : ActorConfig .
mb  CX < CY > : ActorConfig .
cmb CX ( X ) . P  : ActorConfig if  P : ActorConfig and
```

```
                        s(s(0)) > | down(X , recep(P)) \ { CX } |  and
                        chainCondition(chanMap(P),[shiftup X] CX, recep(P)) and
                        not X { 0 } in recep(P) .
cmb P | Q : ActorConfig  if  P : ActorConfig and
                    Q : ActorConfig and recep(P) cap recep(Q) == emptychanset .
cmb if CX = CY then P else Q fi : ActorConfig  if  P : ActorConfig and
                    Q : ActorConfig and compatible?(chanMap(P), chanMap(Q)) .
cmb new [X] P : ActorConfig  if  P : ActorConfig .
cmb B < t1 ; t2 > : ActorConfig if defined?(B , D) and
                                arity1(B,context) == length(t1) and
                                arity2(B,context) == length(t2) and
                                if length(t1) == s(s(0))
                                then pick(t1 , s(0)) =/= pick(t2 , s(s(0)))
                                else true fi .

eq  chainCondition( emptymap , CX , emptychanset ) = true .
eq  chainCondition( { ( CY bot ) } , CX , { CY } ) = true .
ceq chainCondition( { ( CX CY ) , ( CY bot ) } , CX , { CX , CY } ) = true
                                                                  if CX =/= CY .
```

In the type rule for behavior instantiations, note that we refer to the constant `context` that is defined in module `APICONTEXT` (see Appendix B). This constant contains the collection of all the behavior definitions that can be used in specifying processes. The operators `arity1` and `arity2` are also defined in `APICONTEXT`.

Recall that type checking a term involves type checking all the behavior definitions it uses. We define the operator `welldefined?` to check if all behavior definitions in a given context are well typed.

```
op  welldefined? : Context -> Bool .
op  welltyped?  :  Context Context -> Bool .
op  qidTupleToChanMap  :  Qidtuple -> Pairset .

eq  welldefined?(context) = unique?(context) and wellformed?(context) and
                            welltyped?(emptycontext , context) .
```

```
eq  welltyped?(D,emptycontext) = true .
eq  welltyped?(D,(B := (q1 ; q2) CX(X) . P , D')) =
                              CX(X) . P : ActorConfig and
                              recep(CX(X) . P) == tochanset(q1 , 0) and
                              chanMap(CX(X) . P) == qidTupleToChanMap(q1) and
                              welltyped?((D,B := (q1 ; q2) CX(X) . P),D') .

eq  qidTupleToChanMap(emptyqidtuple) = emptychanset .
eq  qidTupleToChanMap(X) = { ( (X{0}) bot ) } .
eq  qidTupleToChanMap(X Y q) = { ( (X{0}) (Y{0}) ) } cup qidTupleToChanMap(Y q) .
```

The operator `unique?` checks if each behavior identifier has a unique definition. The operator `wellformed?` checks for syntactic constrains on each definition $B \stackrel{def}{=} (\tilde{u};\tilde{v})x(y).P$, such as $1 \leq \mathit{length}(\tilde{u}) \leq 2$, $u_1 == x$ and that the identifiers in the tuple $\tilde{u},\tilde{v}$ are all distinct. These operators are defined in the module `APICONTEXT` shown in Appendix B. The operator `welltyped?` checks if for each definition $B \stackrel{def}{=} (\tilde{u};\tilde{v})x(y).P$, the judgment $\{\tilde{u}\}; ch(\tilde{u}) \vdash P$ holds. For a tuple of identifiers `QT` and natural `n`, `tochanset(QT,n)` returns a set of channels that contains the channel `X{n}` for each identifier `X` in `QT`.

For the operational semantics of A$\pi_{\neq}$ we only need to introduce the following rewrite rule for behavior instantiations, in addition to the rewrite rules for asynchronous π-calculus given in Section 6.1.2.

```
op  defn : Qid Context -> Defn .
op  inst : Defn Chantuple -> Term .

crl [Behv] : [CS1] B < CT1 ; CT2 > => [CS2] P
                              if [CS1] inst(defn(B), CT1 CT2) => [CS2] P .
```

The operator `defn` returns the definition of a given constant from the implicit context of definitions, and the operator `inst` instantiates a behavior definition with a given tuple

of channel names. The reader is referred to the module APICONTEXT in Appendix B for the definition of these operators.

We now describe our implementation of verification of parameterized may preorder between finite (non-recursive) A$\pi_{\neq}$ processes. Following the ideas in Section 6.2, we exploit the fact that $P \stackrel{\sqsubset}{\sim}_\rho Q$ if and only if $[\![Q]\!] \precsim_\rho [\![P]\!]$ if and only if $[\![Q]\!]_{fn(P,Q)} \precsim_\rho [\![P]\!]_{fn(P,Q)}$. We use the metalevel facilities in Maude to compute the trace sets $[\![P]\!]_{fn(P,Q)}$ and $[\![Q]\!]_{fn(P,Q)}$ as before, and compare these trace sets according to Theorem 3.2. The main difference here as opposed to the procedure described in Section 6.2 is that we consider only the ρ-well-formed traces in $[\![Q]\!]_{fn(P,Q)}$, and compute the reflexive transitive closure of such traces according to only the laws *L1* and *L2* of Table 2.2.

```
op APITRACE-MOD : -> Module .
eq APITRACE-MOD = ['APITRACE] .

op  << : TermSet TermSet Term -> Bool .
op  <<< : TermSet Term Term -> Bool .
op  maypre  :  Term Term Term -> Bool .

eq  <<(TS2,mt,T2) = true .
eq  <<(TS2, (T1 u TS1), T2) = <<<(TS2, T1, T2) and <<(TS2, TS1, T2) .
eq  <<<(TS2,T1,T2) = not (metarwf(T1,T2) and disjoint?(TS2 , TTraceClosure(T1))) .
eq  metarwf(T1,T2) = (getTerm(metaReduce(APITRACE-MOD, 'rwf[T1 , T2])) == 'true.Bool) .
eq  maypre(T1,T2,T3) = <<(TTrmtoNormalTraceSet(T2), TTrmtoNormalTraceSet(T1), T3) .
```

The operators `<<` and `maypre`, which implement $\precsim_\rho$ and $\stackrel{\sqsubset}{\sim}_\rho$ respectively, now take an additional third argument which is a metarepresentation of the set of channel names ρ. The definition of `TTraceClosure` in Section 6.2 is modified so that it computes the closure of a metarepresented trace T according to only the laws *L1* and *L2* of Table 2.2. In the equation for `<<<`, we consider only the ρ-well-formed traces; `metarwf(T1,T2)` checks for a given metarepresentation `T1` of a trace and a metarepresentation `T2` of a set of channel names ρ, whether the trace is ρ-well-formed. The function `metarwf` uses the metalevel facility `metaReduce` in Maude. The function `rwf`, which checks if a trace is ρ-well-formed,

is assumed to be defined in the module `APITRACE`. Following is a specification of `rwf`, which closely follows Definition 2.5.

```
op  rwf : Trace Chanset -> Bool [frozen] .

eq  rwf(epsilon , CS) = true .
eq  rwf(f(i,CX,CY) . TR , CS) = rwf(TR , CS) .
eq  rwf(b(i,CX,Y) . TR , CS) = rwf(TR , [shiftup Y] CS) .
eq  rwf(f(o,CX,CY) . TR , CS) = (not CX in CS) and rwf(TR , CS) .
eq  rwf(b(o,CX,Y) . TR , CS) = (not CX in CS) and rwf(TR , Y{0} [shiftup Y] CS) .
```

6.4 Discussion and Related Work

We have described an executable specification in Maude of the operational semantics of an asynchronous version of the π-calculus and A$\pi_{\neq}$. In addition, we have also specified the unparameterized may preorder for π-calculus processes, and the parameterized may preorder for A$\pi_{\neq}$ processes. The new features introduced in Maude 2.0, including rewrites in conditions, the `frozen` attribute, and the `metaSearch` operation, have been essential for the development of this executable specification.

The first specification of the π-calculus operational semantics in rewriting logic was developed by Viry in Elan [97]. The operational semantics specified by Viry was in the reduction style, where first an equivalence is imposed on syntactic processes (typically to make syntax more abstract with respect to properties of associativity and/or commutativity of some operators), and then some reduction or rewrite rules express how the computation proceeds by communication between processes. In contrast, we have specified an operational semantics in the labeled transition system style according to the SOS approach introduced by Plotkin [73]. Viry's specification makes use of de Bruijn indexes, explicit substitutions, and reduction strategies in Elan [14]. This presentation was later improved by Stehr [86] by making use of a generic calculus for explicit substitutions, known as *CINNI*, which combines the best of the approaches based on standard variables and de Bruijn indices.

Our work took the work described above as a starting point, together with recent work by Verdejo and Martí-Oliet [96] showing how to use the new features of Maude 2.0 in the implementation of a semantics in the labeled transition system style for CCS. For instance, the ideas of using conditional rewrite rules with rewrites in conditions to represent the transition rules, and using the `frozen` attribute to control the application of rules, were first introduced in [96].

An interesting direction of further work is to extend our implementation to include the algorithm in Section 5.4 that is applicable to a large class of non-finitary asynchronous π-calculus processes. This would first involve extracting the asynchronous finite state machine model (if possible) out of a given π-calculus processes. Another direction of work is to look for interesting concrete applications to which these implementations can be applied.

Chapter 7

Conclusion

We have investigated the theory of may testing equivalence over several variants of π-calculus. The variants incorporate computational phenomena such as asynchrony, locality, the object-paradigm, and restricted name matching, which are not part of the basic π-calculus. For each variant, we presented a characterization of a generalized version of may testing, which provides a powerful proof technique for establishing equivalences between programs. We exploited this characterization to get a complete axiomatization of the may preorder for the finitary fragment of each variant, and a fully automated algorithm for establishing may equivalences over a class of infinite state systems. We have also presented an executable specification of the variants and may testing over them, in Maude 2.0.

We have shown how the theory of testing over π-calculus and its variants can be applied to concurrent object-based models such as the Actor model. A practical sequel to this would be to apply our results to the language of Specification Diagrams (SDs) [84, 85]. SDs provide a graphical notation for specifying message passing behaviors of open distributed object systems. They have an intuitive appeal like other popular graphical languages such as UML [80] and MSC [77]. SDs share several similarities to the various calculi we have studied; they allow dynamic generation of names and name passing as in the π-calculus, they have asynchronous communication and enforce locality as in Lπ, and they enforce the uniqueness property of names as in Aπ. In addition, they

are equipped with imperative features such as variables, environments, and assignments, and logical features such as assertions and constraints which are more appropriate for specification languages. By recasting SDs as an extension of asynchronous π-calculus with locality, it is likely that both our theory of may testing and the implementation techniques in Maude can be lifted in a straightforward manner to SDs. The result would be a tool that can be used to execute diagrams and establish semantic correspondence between different diagrams. The language of SDs is designed in such a way that one can describe systems at various levels of abstraction, ranging from high-level specifications to concrete diagrams with low-level implementation details. Our theory of may testing would provide powerful techniques to establish semantic correspondence between such diagrams at different levels of abstraction.

Another direction of work would be to complete the testing scenario by considering must testing over all the variants of π-calculus. We have so far only briefly considered a few algorithmic questions related to must testing in Chapter 5. The characterization of must testing on asynchronous π-calculus is not known, although characterizations over asynchronous CCS are known [19]. Further, the effect of other computational features of interest such as locality, object-paradigm, and restricted name matching is not known. Must testing in Aπ poses an additional challenge due to fairness. Unlike in may testing, it is known that fairness makes a difference in must testing [67].

Bibliography

[1] M. Abadi and A. Gordon. Reasoning about cryptographic protocols in the spi calculus. In *CONCUR'97: Concurrency Theory, Springer Lecture Notes in Computer Science, volume 1243*, pages 59–73, July 1997.

[2] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Tsay Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[3] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *IEEE International Symposium on Logic in Computer Science*, 1993.

[4] Parosh Aziz Abdulla and Bengt Jonsson. Channel representations in protocol verification. In *CONCUR*, pages 1–15, 2001.

[5] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[6] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[7] G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 1996.

[8] G. Agha and P. Thati. An algebraic theory of actors and its application to a simple object-based language. In *Festschrift in honor of Ole-Johan Dahl.* Springer Verlag, 2003. to be published in Lecture Notes in Computer Science.

[9] R. Amadio, I.Castellani, and D. Sangiorgi. On Bisimulations for Asynchronous π-Calculus. In *Proceedings of CONCUR '96*. Springer-Verlag, 1996. LNCS 1119.

[10] M. Boreale and R. De Nicola. Testing Equivalences for Mobile Processes. In *Third International Conference on Concurrency Theory, LNCS 630*, pages 2–16. Springer-Verlag, August 1992. LNCS 630.

[11] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes.

[12] M. Boreale, R. De Nicola, and R. Pugliese. A theory of may testing for asynchronous languages. In *Foundations of Software Science and Computation Structures*, pages 165–179, 1999. LNCS 1578.

[13] M. Boreale and D. Sangiorgi. Bisimulation in Name Passing Calculi without Matching. *Proceedings of LICS*, 1998.

[14] P. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[15] Ahmed Bouajjani and Richard Mayr. Model checking lossy vector addition systems. In *STACS*, pages 323–333, 1999.

[16] G. Boudol. Asynchrony and the π-Calculus. Technical Report 1702, Department of Computer Science, Inria Univeristy, May 1992.

[17] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification over Infinite States. In *Handbook of Process Algebra*, pages 545–623. Elsevier Publishing, 2001.

[18] C. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[19] I. Castellini and M. Hennesy. Testing theories for asynchronous languages. In *FSTTCS*, pages 90–101, 1998. LNCS 1530.

[20] S. Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis, Department of Computer Science, University of Edinburgh, 1993.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[23] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In Kokichi Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[24] W.D. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, AI Laboratory, 1981.

[25] J. Darlington and Y. K. Guo. Formalizing actors in linear logic. In *International Conference on Object-Oriented Information Systems*, pages 37–53. Springer-Verlag, 1994.

[26] D.Sangiorgi. A theory of bisimulation for π-calculus. In *Proceedings of CONCUR*, 1993. LNCS 715.

[27] D.Sangiorgi and D. Walker. Some results on barbed equivalences in π-calculus. In *Proceedings of CONCUR*, 2001.

[28] J. Esparza. Decidability and Complexity of Petri Net problems — An Introduction. In *Advances in Petri Nets*, volume 1491 of *Lecture Notes inComputer Science*, pages 374–428. Springer, 1998.

[29] F.Dagnat, M.Pantel, M.Colin, and P.Salle. Typing concurrent objects and actors. In *L'Objet – Mthodes formelles pour les objets (L'OBJET)*, volume 6, pages 83–106, 2000.

[30] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join Calculus. *Proceedings of POPL*, 1996.

[31] S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach for Synchronization*. MIT Press, November 1996.

[32] M. Gaspari and G. Zavattaro. An Algebra of Actors. Technical Report UBLCS-97-4, Department of Computer Science, Univeristy of Bologna (Italy), May 1997.

[33] U. Goltz and A. Mycroft. On the relationship of CCS and petri nets. In *Proceedings of ICALP*, pages 196–208, 1984. LNCS 172.

[34] M . Hack. *Decidability questions for Petri nets*. PhD thesis, Massachusetts Institute of Technolgy, Lobratory of Computer Science, 1976.

[35] M. Hack. Decision problems for Petri nets and vector addition systems. Technical Report MAC, Memo 53, MIT, 1975.

[36] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[37] M. Hennessy and J. Rathke. Typed behavioral equivalences for processes in the presence of subtyping. Technical report, University of Sussex, Computer Science, March 2001.

[38] C. Hewitt. Viewing Control Structures as Patterns of Message Passing. *Journal of Artificial Intelligence*, 8(3):323–364, September 1977.

[39] G. H. Higman. Ordering by divisibility in abstract algebras. *Proceedings of London Mathematical Society*, 3:326–336, 1952.

[40] Y. Hirshfeld. Petri nets and the equivalence problem. In *Lecture Notes in Computer Science 832*, pages 165–174. Springer Verlag, 1993.

[41] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Fifth European Conference on Object-Oriented Programming*, July 1991. LNCS 512, 1991.

[42] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesely, 1979.

[43] J-L.Colao, M.Pantel, and P.Salle. Analyse de linarit par typage dans un calcul d'acteurs primitifs. In *Actes des Journes Francophones des Langages Applicatifs (JFLA)*, 1997.

[44] P. Jancar. Undecidability of bisimilarity for petri nets and some related problems. *Theoretical Computer Science*, 148:281–301, 1995.

[45] P.C. Kanellakis and S.A.Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):48–68, May 1990.

[46] R. Karp and R. Miller. Parallel program schemata. *Journal of Computing System Science*, 3:147–195, 1969.

[47] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-calculus. $23^{rd} ACM$ *Symposium on Principles of Programming Languages*, pages 358–371, 1996.

[48] N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, pages 137–166, 1994.

[49] N. Kobayashi and A. Yonezawa. Towards foundations of concurrent object-oriented programming – types and language design. *Theory and Practice of Object Systems*, 1(4), 1995.

[50] A. J. Korenjak and J. E. Hopcroft. Simple deterministic langauges. In *IEEE Symposium on Automata and Switching Theory*, pages 36–46, 1966.

[51] R. Lipton. The Reachability Problem Requires Exponential Space. Technical Report 62, Yale University, 1976.

[52] C. Manning. Acore: The design of a core actor language and its compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, 1987.

[53] N. Mart'i-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework, 1993.

[54] I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.

[55] I.A. Mason and C.Talcott. A semantically sound actor translation. In *ICALP 97*, pages 369–378, 1997. LNCS 1256.

[56] M.Boreale and D.Sangiorgi. Some congruence properties for π-calculus bisimilarities. In *Theoretical Computer Science 198*, 1998.

[57] M. Merro. On equators in asynchronous name-passing calculi without matching. *Electronic Notes in Theoretical Computer Science*, 27, 1999.

[58] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceeding of ICALP '98*. Springer-Verlag, 1998. LNCS 1443.

[59] J. Meseguer. Rewriting Logic as a Unified Model of Concurrency. Technical Report SRI-CSI-90-02, SRI International, Computer Science Laboratory, February 1990.

[60] R. Milner. A complete inference system for a class of regular behaviors. *Journal of Computing and Systems Science*, 28:439–466, 1984.

[61] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[62] R. Milner. Interactions, turing award lecture. *Communications of the ACM*, 36(1):79–97, January 1993.

[63] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[64] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, Part II. Technical Report ECS-LFCS-89-86, Department of Computer Science, Edinburgh University, June 1989.

[65] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[66] E. F. Moore. Gedanken experiments on sequential machines. In *Automata studies*, pages 129–153, 1956.

[67] V. Natarajan and R. Cleaveland. Divergence and fair testing. In *International Conference on Automata Languages and Programming (ICALP)*, pages 648–659, 1995.

[68] R. De Nicola and M. Hennesy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[69] J. Parrow and D. Sangiorgi. Algebraic Theories of Name-Passing Calculi. *Information and Computation*, 120(2), 1995.

[70] J. L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981.

[71] B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[72] B. C. Pierce and D. N. Turner. Pict: A programming Language Based on the π-Calculus. Technical Report CSCI-476, Indiana University, March 1997.

[73] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, September 1981.

[74] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57. IEEE Computer Society Press, 1977.

[75] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

[76] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR*, pages 474–488, 2000. LNCS 1877.

[77] ITU-T Recommendation Z.120. Message sequence charts, 1996.

[78] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the π-calculus process algebra. In *Proceedings of the Pacific Symposium of Biocomputing*, pages 459–470, 2001.

[79] R.Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of* 19^{th} *International Colloquium on Automata, Languages and Programming (ICALP '92)*. Springer Verlag, 1992. LNCS 623.

[80] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wisely, 1998.

[81] D. Sangiorgi. Typed π-Calculus at Work: A Proof of Jone's Transformation on Concurrent Objects. *Theory and Practice of Object-Oriented Systems*, 1997.

[82] D. Sangiorgi. An Interpretation of Typed Objects into Typed π-Calculus. *Information and Computation*, 143(1), 1998.

[83] D. Sangiorgi. The Name Discipline of Uniform Receptiveness. *Theoretical Computer Science*, 221, 1999.

[84] S. Smith and C. Talcott. Modular reasoning for actor specification diagrams. In *Formal Methods in Object-Oriented Distributed Systems*. Kluwer Academic Publishers, 1999.

[85] S. Smith and C. Talcott. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation*, 2002. To appear.

[86] M. O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In Kokichi Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[87] D.C. Sturman. *Modular Specification of Interaction in Distributed Computing*. PhD thesis, University of Illinois at Urbana Champaign, 1996.

[88] C. Talcott. An Actor Rewriting Theory. In *Electronic Notes in Theoretical Computer Science 5*, 1996.

[89] C. Talcott. Interaction Semantics for Components of Distributed Systems. In E.Najm and J.B. Stefani, editors, *Formal Methods for Open Object Based Distributed Systems*. Chapman & Hall, 1996.

[90] C. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

[91] P. Thati. Towards an Algebraic Formulation of Actors. Master's thesis, Computer Science, University of Illinois at Urbana Champaign, 2000.

[92] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous π-calculus and may testing in Maude 2.0. In *International Conference on Rewriting Logic and its Applications*, 2002. Electronic Notes in Theoretical Computer Science, vol. 71.

[93] P. Thati, R. Ziaei, and G. Agha. A theory of may testing for actors. In *Formal Methods for Open Object-based Distributed Systems*, March 2002.

[94] P. Thati, R. Ziaei, and G. Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In *Proceedings of the* 9^{th} *International Conference on Algebraic Methodology and Software Technology*, pages 222–238. Springer Verlag, September 2002. LNCS.

[95] N. Venkatasubramanian and C. Talcott. Meta-Architectures for Resource Management in Open Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, August 1995.

[96] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In *International Conference on Rewriting Logic and its Applications*, 2002. Electronic Notes in Theoretical Computer Science, vol. 71.

[97] P. Viry. Input/output for ELAN. In José Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 51–64. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[98] P. Viry. A rewriting implementation of π-calculus. Technical Report TR-96-30, 26 1996.

[99] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

APPENDIX A

Proofs for Chapter 2

Proof of Lemma 2.6.1: Since we work modulo alpha equivalence on process, without loss of generality we may assume the hygiene condition $w, y \notin bn(P_0)$. The proof is by induction on the derivation of $P_0 \xrightarrow{(\hat{v})uv} P_1$. For the base case, we have $P_0 = u(z).Q$, $P_1 = Q\{v/z\}$, the last derivation step is an application of *INP* rule, and $\hat{v} \cap fn(P_0) = \emptyset$. By hygiene condition $z \neq w, y$. Let $P_0' \in P_0[w/y]$. Then $P_0' = u(z).Q'$ for some $Q' \in Q[w/y]$. By *INP* rule we have, $u(z).Q' \xrightarrow{uv} Q'\{v/z\}$. Since $w \notin \hat{v}$ it follows that $u(z).Q' \xrightarrow{(\hat{v})uv} Q'\{v/z\}$. Now, since $z \neq w, y$, we have $Q'\{v/z\} \in Q\{v/z\}[w/y]$ and the lemma follows.

For the induction step, there are three cases.

1. $P_0 = Q_0|R$, $P_1 = Q_1|R$, and the last derivation step is an application of *PAR* rule:

$$PAR: \quad \frac{Q_0 \xrightarrow{(\hat{v})uv} Q_1}{Q_0|R \xrightarrow{(\hat{v})uv} Q_1|R} \quad \hat{v} \cap fn(R) = \emptyset$$

   Then $P_0' = Q_0'|R'$ for some $Q_0' \in Q_0[w/y]$ and $R' \in R[w/y]$. By induction hypothesis we have $Q_0' \xrightarrow{(\hat{v})uv} Q_1'$ for some $Q_1' \in Q_1[w/y]$. From $w \notin \hat{v}$ and $\hat{v} \cap fn(R) = \emptyset$ we deduce $\hat{v} \cap fn(R') = \emptyset$. Then by *PAR* rule we have $Q_0'|R' \xrightarrow{(\hat{v})uv} Q_1'|R'$, and the lemma follows from the fact that $Q_1'|R' \in P_1[w/y]$.

2. $P_0 = (\nu z)Q_0$, $P_1 = (\nu z)Q_1$ and the last derivation step is an application of *RES* rule:

$$RES: \quad \frac{Q_0 \xrightarrow{(\hat{v})uv} Q_1}{(\nu z)Q_0 \xrightarrow{(\hat{v})uv} (\nu z)Q_1} \quad z \notin \{u, v\}$$

   By the hygiene condition, we have $z \neq w, y$. Then $P_0' = (\nu z)Q_0'$ for some $Q_0' \in Q_0[w/y]$. By induction hypothesis, $Q_0' \xrightarrow{(\hat{v})uv} Q_1'$ for some $Q_1' \in Q_1[w/y]$. Then by *RES* rule we have $(\nu z)Q_0' \xrightarrow{(\hat{v})uv} (\nu z)Q_1'$, and the lemma follows from the fact that $(\nu z)Q_1' \in P_1[w/y]$.

3. $P_0 = !Q_0$, $P_1 = Q_1$ and the last derivation step is an application of *REP* rule:

$$REP: \quad \frac{Q_0 | !Q_0 \xrightarrow{(\hat{v})uv} Q_1}{!Q_0 \xrightarrow{(\hat{v})uv} Q_1}$$

   Then $P_0' = !Q_0'$ for some $Q_0' \in Q_0[w/y]$. Since $Q_0' | !Q_0' \in (Q_0 | !Q_0)[w/y]$, by induction hypothesis we have $Q_0' | !Q_0' \xrightarrow{(\hat{v})uv} Q_1'$ for some $Q_1' \in Q_1[w/y]$. Then by *REP* rule we have $!Q_0' \xrightarrow{(\hat{v})uv} Q_1'$, and the lemma follows. □

Proof Lemma 2.6.2: Since we work modulo alpha equivalence on process, without loss of generality we may assume the hygiene condition $w, y \notin bn(P_0)$. The proof is by induction on the derivation of $P_0 \xrightarrow{(\hat{v})\overline{u}v} P_1$. There are two base cases.

1. $P_0 = \overline{u}v$, $P_1 = 0$ and the last derivation step is an application of *OUT* rule.

$$OUT: \quad \overline{u}v \xrightarrow{\overline{u}v} 0$$

   Then $P_0' = \overline{u'}v'$ for some $u' \in u[w/y]$ and $v' \in v[w/y]$. By *OUT* rule we have $\overline{u'}v' \xrightarrow{\overline{u'}v'} 0$, and the lemma follows.

2. $P_0 = (\nu v)Q_0$, $P_1 = Q_1$ and the last derivation step is an application of *OPEN* rule.

$$OPEN: \quad \frac{Q_0 \xrightarrow{\overline{u}v} Q_1}{(\nu v)Q_0 \xrightarrow{\overline{u}(v)} Q_1} \quad u \neq v$$

By hygiene condition, we have $v \neq w, y$. Then $P_0' = (\nu v)Q_0'$ for some $Q_0' \in Q_0[w/y]$. By induction hypothesis $Q_0' \xrightarrow{\overline{u'}v'} Q_1'$ for some $u' \in u[w/y]$, $v' \in v[w/y]$, and $Q_1' \in Q_1[w/y]$. Further, since $y \neq v$ we have $v' = v$, and since $w \neq v$ we have $u' \neq v$. Then by *RES* rule we have $(\nu v)Q_0' \xrightarrow{\overline{u'}(v)} Q_1'$, and the lemma follows.

For the induction step there are three cases.

1. $P_0 = Q_0|R$, $P_1 = Q_1|R$, and the last derivation step is an application of *PAR* rule.

$$PAR: \quad \frac{Q_0 \xrightarrow{(\hat{v})\overline{u}v} Q_1}{Q_0|R \xrightarrow{(\hat{v})\overline{u}v} Q_1|R} \quad \hat{v} \cap fn(R) = \emptyset$$

   Then $P_0' = Q_0'|R'$ for some $Q_0' \in Q_0[w/y]$ and $R' \in R[w/y]$. By induction hypothesis, $Q_0' \xrightarrow{\alpha} Q_1'$ for some $Q_1' \in Q_1[w/y]$ and α as stated in the lemma. We have $bn(\alpha) = \hat{v}$. From $w \notin \hat{v}$ and $\hat{v} \cap fn(R) = \emptyset$ we deduce $bn(\alpha) \cap fn(R') = \emptyset$. Then by *PAR* rule $Q_0'|R' \xrightarrow{\alpha} Q_1'|R'$, and the lemma follows from the fact that $Q_1'|R' \in P_1[w/y]$.

2. $P_0 = (\nu z)Q_0$, $P_1 = (\nu z)Q_1$, and the last derivation step is an application of *RES* rule:

$$RES: \quad \frac{Q_0 \xrightarrow{(\hat{v})\overline{u}v} Q_1}{(\nu z)Q_0 \xrightarrow{(\hat{v})\overline{u}v} (\nu z)Q_1} \quad z \neq u, v$$

   By hygiene condition we have $z \neq w, y$. Then $P_0' = (\nu z)Q_0'$ for some $Q_0' \in Q_0[w/y]$. By induction hypothesis, $Q_0' \xrightarrow{\alpha} Q_1'$ for some $Q_1' \in Q_1[w/y]$ and α as stated in the lemma. Further since $z \notin \{w, y, u, v\}$ we have $z \notin n(\alpha)$. Then by *RES* rule we have $(\nu z)Q_0' \xrightarrow{\alpha} (\nu z)Q_1'$, and the lemma follows from the fact that $(\nu z)Q_1' \in P_1[w/y]$.

3. $P_0 = !Q_0$, $P_1 = Q_1$ and the last derivation step is an application of *REP* rule:

$$REP: \quad \frac{Q_0|!Q_0 \xrightarrow{(\hat{v})\overline{u}v} Q_1}{!Q_0 \xrightarrow{(\hat{v})\overline{u}v} Q_1}$$

Then $P_0' = !Q_0'$ for some $Q_0' \in Q_0[w/y]$. Since $Q_0'|!Q_0' \in (Q_0|!Q_0)[w/y]$, by induction hypothesis we have $Q_0'|!Q_0' \xrightarrow{\alpha} Q_1'$ for some $Q_1' \in Q_1[w/y]$ and α as stated in the lemma. Then by *REP* rule we have $!Q_0' \xrightarrow{\alpha} Q_1'$, and the lemma follows. □

Proof of Lemma 2.6.3: Since we work modulo alpha equivalence on process, without loss of generality we may assume the hygiene condition $w, y \notin bn(P_0, P_1)$. The proof is by induction on derivation of $P_0 \xrightarrow{\tau} P_1$. There are two base cases depending on the last derivation step.

1. $P_0 = Q_0|R_0$, $P_1 = Q_1|R_1$ and the last derivation step is

$$COM: \quad \frac{Q_0 \xrightarrow{\overline{u}v} Q_1 \quad R_0 \xrightarrow{uv} R_1}{Q_0|R_0 \xrightarrow{\tau} Q_1|R_1}$$

   Then $P_0' = Q_0'|R_0'$ for some $Q_0' \in Q_0[w/y]$ and $R_0' \in R_0[w/y]$. By Lemma 2.6.2, $Q_0' \xrightarrow{\overline{u'}v'} Q_1'$ for some $Q_1' \in Q_1[w/y]$, $u' \in u[w/y]$ and $v' \in v[w/y]$. There are two subcases.

   (a) $u' = u$: Since random substitution on processes does not change input prefixes, it follows that $R_0' \xrightarrow{uv'} R_1'$, $R_1' \in R_1[w/y]$. Using *COM* we have $Q_0'|R_0' \xrightarrow{\tau} Q_1'|R_1'$. Then condition 1 of lemma holds with $P_1' = Q_1'|R_1'$.

   (b) $u' = w$: Then $u = y$. Since random substitution on processes does not change input prefixes, it follows that $R_0' \xrightarrow{yv'} R_1'$, $R_1' \in R_1[w/y]$. Then by *PAR* rule we have $P_0' = Q_0'|R_0' \xrightarrow{\overline{w}v'} Q_1'|R_0' \xrightarrow{yv'} Q_1'|R_1'$. Then condition 2 of lemma holds with $z = v'$ and $\hat{z} = \emptyset$, $Q = Q_1|R_1$, and $P_1' = Q_1'|R_1'$.

2. $P_0 = Q_0|R_0$, $P_1 = (\nu v)(Q_1|R_1)$ and the last derivation step is

$$CLOSE: \quad \frac{Q_0 \xrightarrow{\overline{u}(v)} Q_1 \quad R_0 \xrightarrow{uv} R_1}{Q_0|R_0 \xrightarrow{\tau} (\nu v)(Q_1|R_1)} \quad v \notin fn(R_0)$$

Then $P_0' = Q_0'|R_0'$ for some $Q_0' \in Q_0[w/y]$ and $R_0' \in R_0[w/y]$. By hygiene condition $v \neq w, y$. Then by Lemma 2.6.2, we have $Q_0' \xrightarrow{\overline{u'}(v)} Q_1'$ for some $Q_1' \in Q_1[w/y]$ and $u' \in u[w/y]$. There are two subcases.

(a) $u' = u$: Since random substitution on processes does not change input prefixes, it follows that $R_0' \xrightarrow{uv} R_1'$, $R_1' \in R_1[w/y]$. From $v \neq w, y$ and $v \notin fn(R_0)$ we deduce $v \notin fn(R_0')$. Using *COM* we have $Q_0'|R_0' \xrightarrow{\tau} (\nu v)(Q_1'|R_1')$. Then condition 1 of lemma holds with $P_1' = (\nu v)(Q_1'|R_1')$.

(b) $u' = w$: Then $u = y$. Since random substitution on processes does not change input prefixes, it follows that $R_0' \xrightarrow{yv} R_1'$, $R_1' \in R_1[w/y]$. From $v \neq w, y$ and $v \notin fn(R_0)$ we deduce $v \notin fn(R_0')$. Then by *PAR* rule we have $P_0' = Q_0'|R_0' \xrightarrow{\overline{w}(v)} Q_1'|R_0' \xrightarrow{yv} Q_1'|R_1'$. Then condition 2 of the lemma holds with $z = v$ and $\hat{z} = \{v\}$, $Q = Q_1|R_1$, and $P_1' = Q_1'|R_1'$.

For the induction step, there are three cases.

1. $P_0 = Q_0|R$, $P_1 = Q_1|R$, and the last derivation step is an application of *PAR* rule:

$$PAR: \quad \frac{Q_0 \xrightarrow{\tau} Q_1}{Q_0|R \xrightarrow{\tau} Q_1|R}$$

Then $P_0' = Q_0'|R'$ for some $Q_0' \in Q_0[w/y]$ and $R' \in R[w/y]$. By induction hypothesis we have two cases.

(a) $Q_0' \xrightarrow{\tau} Q_1'$ for some $Q_1' \in Q_1[w/y]$. Then by *PAR* rule, $P_0' = Q_0'|R' \xrightarrow{\tau} Q_1'|R'$ and condition 1 of the lemma holds with $P_1' = Q_1'|R'$.

(b) $Q_1 \equiv (\nu \hat{z})S$, $w, y \notin \hat{z}$, $Q_0' \xrightarrow{(\hat{z})\overline{w}z.yz} S'$ for some $S' \in S[w/y]$. Without loss of generality we may assume $\hat{z} \cap fn(R) = \emptyset$. Then $P_1 \equiv (\nu \hat{z})(S|R)$, and $\hat{z} \cap fn(R') = \emptyset$. Then by *PAR* rule $P_0' = Q_0'|R' \xrightarrow{(\hat{z})\overline{w}z.yz} S'|R'$. Then condition 2 of the lemma holds.

2. $P_0 = (\nu v)Q_0$, $P_1 = (\nu v)Q_1$ and the last derivation step is an application of *RES* rule:

$$\textit{RES:}\ \frac{Q_0 \xrightarrow{\tau} Q_1}{(\nu v)Q_0 \xrightarrow{\tau} (\nu v)Q_1}$$

   By hygiene condition, we have $v \neq w, y$. Then $P_0' = (\nu v)Q_0'$ for some $Q_0' \in Q_0[w/y]$. By induction hypothesis we have two cases.

   (a) $Q_0' \xrightarrow{\tau} Q_1'$ for some $Q_1' \in Q_1[w/y]$. Then by *RES* rule, $(\nu v)Q_0' \xrightarrow{\tau} (\nu v)Q_1'$ and condition 1 of the lemma holds with $P_1' = (\nu v)Q_1'$.

   (b) $Q_1 \equiv (\nu \hat{z})S$, $w, y \notin \hat{z}$, $Q_0' \xrightarrow{(\hat{z})\overline{w}z.yz} S'$ for some $S' \in S[w/y]$. Without loss of generality we may assume $v \notin \hat{z}$. There are two subcases:

   - $v = z$. Then $\hat{z} = \emptyset$. Then by *OPEN* rule $P_0' \equiv (\nu v)Q_0' \xrightarrow{\overline{w}(v).yv} S'$. Then condition 2 of the lemma holds with $z = v$, $\hat{z} = \{v\}$, $P_1 \equiv (\nu v)S$ and $P_1' = S'$.
   - $v \neq z$: Then by *RES* rule $P_0' \equiv (\nu v)Q_0' \xrightarrow{(\hat{z})\overline{w}z.yz} (\nu v)S'$. Then condition 2 of the lemma holds with $P_1 \equiv (\nu \hat{z})(\nu v)S$ and $P_1' = (\nu v)S'$.

3. The case where the last derivation step is an application of *REP* rule is straightforward. □

Proof of Lemma 2.9: Since we work modulo alpha equivalence on processes, without loss of generality we may assume the hygiene condition $w \notin bn(P)$. The proof is by induction on the derivation of $P \xrightarrow{xy} P_1$. For the base case, we have $P = x(z).Q$, $P_1 = Q\{y/z\}$, and the last derivation step is an application of *INP* rule. Then, by *INP* rule and $w \notin fn(P)$, we have $x(z).Q \xrightarrow{x(w)} Q\{w/z\}$. Furthermore, by locality, z occurs only in output terms in $Q\{w/z\}$. Therefore, $Q\{w/z\} \in Q\{y/z\}[w/y]$, from which the lemma follows.

For the induction step, there are three cases.

1. $P = Q|R$, $P_1 = Q_1|R$, and the last derivation step is an application of PAR rule:

$$PAR: \quad \frac{Q \xrightarrow{xy} Q_1}{Q|R \xrightarrow{xy} Q_1|R}$$

From $w \notin fn(P)$, it follows that $w \notin fn(Q)$ and $w \notin fn(R)$. Since $w \notin fn(Q)$, by induction hypothesis we have $Q \xrightarrow{x(w)} Q'_1$ for some $Q'_1 \in Q_1[w/y]$. Then, since $w \notin fn(R)$, by PAR rule we have $Q|R \xrightarrow{x(w)} Q'_1|R$. By letting $P'_1 = Q'_1|R$, the lemma follows from $Q'_1|R \in P_1[w/y]$.

2. $P = (\nu z)Q$, $P_1 = (\nu z)Q_1$ and the last derivation step is an application of RES rule:

$$RES: \quad \frac{Q \xrightarrow{xy} Q_1}{(\nu z)Q \xrightarrow{xy} (\nu z)Q_1} \quad z \notin \{x, y\}$$

By hygiene condition $w \neq z$ and hence $w \notin fn(Q)$. Then by induction hypothesis, $Q \xrightarrow{x(w)} Q'_1$ for some $Q'_1 \in Q_1[w/y]$. Now, since $z \notin \{x, w\}$, by RES rule we have $(\nu z)Q \xrightarrow{x(w)} (\nu z)Q'_1$, and the lemma follows from $(\nu z)Q'_1 \in P_1[w/y]$.

3. $P = !Q$, $P_1 = Q_1$ and the last derivation step is an application of REP rule:

$$REP: \quad \frac{Q|!Q \xrightarrow{xy} Q_1}{!Q \xrightarrow{xy} Q_1}$$

Since $w \notin fn(Q|!Q)$, by induction hypothesis, $Q|!Q \xrightarrow{x(w)} Q'_1$ for some $Q'_1 \in Q_1[w/y]$. Then by REP rule we have $!Q \xrightarrow{x(w)} Q'_1$, and the lemma follows. □

Proof of Lemma 2.11: Let $r \in F(s, x, y)$. We prove by induction on the derivation of r, that r is ρ-well-formed. The base case where $r = \epsilon \in F(\epsilon, x, y)$ is obvious. For the induction step there are three cases one for each rule of Definition 2.8.

1. $s = (\hat{v})uv.s'$, $r = (\hat{v})uv.r'$ and $r' \in F(s', x, y)$. Suppose $r = (\hat{v})uv.r_1.(\hat{w})\overline{z}w.r_2$. Now, s' is ρ-well-formed, and by induction hypothesis r' is ρ-well-formed. Then we have $z \notin rcp(r_1, \rho) = rcp((\hat{v})uv.r_1, \rho)$. Hence r is ρ-well-formed.

2. $s = (\hat{v})\overline{u}v.s'$, $r = (\hat{v})\overline{u}v.r'$ and $r' \in F(s', x, y)$. Suppose $r = (\hat{v})\overline{u}v.r_1.(\hat{w})\overline{z}w.r_2$. Now, s' is $(\rho \cup \hat{v})$-well-formed. Then by induction hypothesis r' is $(\rho \cup \hat{v})$-well-formed. Then $z \notin rcp(r_1, \rho \cup \hat{v}) = rcp((\hat{v})\overline{u}v.r_1, \rho)$. Further, since s is ρ-well-formed, $u \notin \rho$. Hence r is ρ-well-formed.

3. $r = (\hat{w})xw.\overline{y}w.[\hat{w}]r'$, for some $r' \in F(s, x, y)$. Let $r = (\hat{w})xw.\overline{y}w.[\hat{w}](r_1.(\hat{v})\overline{u}v.r_2)$. Since $[\hat{w}]r' \neq \bot$, we have $r = (\hat{w})xw.\overline{y}w.([\hat{w}]r_1).(\hat{v})\overline{u}v.r_2'$ for some r_2'. Now, by induction hypothesis r' is ρ-well-formed. Then $u \notin rcp(r_1, \rho)$. Since $[\hat{w}]r_1$ changes only the first bound input with argument w in r_1 (if any), it follows that $rcp(r_1, \rho) = rcp([\hat{w}]r_1, \rho) = rcp((\hat{w})xw.\overline{y}w.[\hat{w}]r_1)$. Now, since $y \notin \rho$, we conclude that r is ρ-well-formed. □

Lemma A.1 (Boreale et al. [11]) *If $P \stackrel{s}{\Longrightarrow}$ then $P\{z/y\} \stackrel{s\{z/y\}}{\Longrightarrow}$.* □

We say $s_1.\overline{x}w.s_2\{w/y\} \preceq_{\{w/y\}} s_1.\overline{x}(y).s_2$. If s_3 is normal and $s_1 \preceq_{\sigma_1} s_2 \preceq_{\sigma_2} s_3$, then we say $s_1 \preceq_{\sigma_1 \oplus \sigma_2} s_3$, where

$$\sigma_1 \oplus \sigma_2 = \begin{cases} \sigma_1(x) & \text{if } \sigma_1(x) \neq x \\ \sigma_2(x) & \text{if } \sigma_2(x) \neq x \\ x & \text{otherwise} \end{cases}$$

Note that, normality of s_3 implies that $\sigma_1 \oplus \sigma_2$ is well-defined. The reader may check the following simple lemma.

Lemma A.2 *If $s \preceq_\sigma r$ then $len(s) = len(r)$. Further, if $s = s_1.s_2$, r is normal and $r = r_1.r_2$ with $len(r_1) = len(s_1)$, then there exist σ_1, σ_2 such that $s_1 \preceq_{\sigma_1} r_1$, $s_2 \preceq_{\sigma_2} r_2\sigma_1$, and $\sigma = \sigma_1 \oplus \sigma_2$.*

Proof: By induction on the length of derivation of $s \preceq_\sigma r$. □

Lemma A.3 *For a finite process P (with no replication), if $P \stackrel{s.(\hat{z})yz}{\Longrightarrow} Q$ and $\hat{z} \cap \{w\} = \emptyset$, then there is $P' \in P[w/y]_i$ such that $P' \stackrel{s.(\hat{z})wz}{\Longrightarrow} Q$.*

Proof: By induction on the derivation of $P \stackrel{s.(\hat{z})yz}{\Longrightarrow} Q$. □

We define random substitution on substitutions as follows

$$\sigma[w/y] = \{\sigma[u \mapsto v] \mid v \in \sigma(u)[w/y]\}$$

Lemma A.4 *For clarity, in the following, we write $P[w/y]_o$ for random output substitution on processes instead of $P[w/y]$ (as in definition 2.7). Let $y \in \rho$, s is ρ-well-formed, $t \in T(s, \rho)$, $s \preceq_\sigma t$, P be a finite process, and $P \stackrel{s}{\Longrightarrow} Q$. Then for every $P_1 \in P[w/y]_o$ there is $P' \in P_1[w/y]_i$, $Q' \in Q[w/y]_o$, $\sigma' \in \sigma[w/y]$, such that $P' \stackrel{s'}{\Longrightarrow} Q'$ and $s' \preceq_{\sigma'} t$.*

Proof: Without loss of generality we may assume that s and t are $\rho \cup \{w\}$-normal. Let $P_1 \in P[w/y]_o$. The proof is by induction on the length of computation $P \stackrel{s}{\Longrightarrow} Q$. The base case is obvious. For the induction step, let

$$P \stackrel{s_1}{\Longrightarrow} Q_1 \stackrel{\alpha}{\longrightarrow} Q$$

There are two cases depending on α.

- $\alpha \neq \tau$: Since $s_1.\alpha \preceq_\sigma t$, by Lemma A.2, we have $t = t_1.\theta$, and for some σ_1, σ_2 such that $\sigma = \sigma_1 \oplus \sigma_2$, $s_1 \preceq_{\sigma_1} t_1$ and $\alpha \preceq_{\sigma_2} \theta\sigma_1$. By induction hypothesis there exist $P' \in P_1[w/y]_i$, $Q'_1 \in Q_1[w/y]_o$, $\sigma'_1 \in \sigma_1[w/y]$, such that $P' \stackrel{s'_1}{\Longrightarrow} Q'_1$ such that $s'_1 \preceq_{\sigma'_1} t_1$. There are two subcases.
  - $\alpha = (\hat{v})\overline{u}v$: Since s is ρ-well-formed and $y \in \rho$ we deduce $u \neq y$. We only consider the case where $\alpha = \overline{u}y$. The case where $v \neq y$ is simpler. Since s is ρ-well-formed, by Lemma 2.12, so is t. The subject of outputs in a ρ-well-formed template are not bound by previous bound outputs. Therefore,

$\theta\sigma_1 = \theta$. Therefore, since $\overline{u}y \preceq_{\sigma_2} \theta\sigma_1$, we deduce $\theta = \overline{u}(v_1)$ for some v_1, and $\sigma_2 = \{y/v_1\}$. Since $Q_1 \xrightarrow{\overline{u}y}$, by Lemma 2.6.2 we have $Q_1' \xrightarrow{\overline{u}y'} Q'$ for some $Q' \in Q[w/y]_o$ and $y' \in y[w/y]$. Let $\alpha' = \overline{u}y'$, $\sigma_2' = \{y'/v_1\}$, and $\sigma' = \sigma_1' \oplus \sigma_2'$. For the same reason as for $\theta\sigma_1 = \theta$, we have $\theta\sigma_1' = \theta$. Then we have $\alpha' \preceq_{\sigma_2'} \theta\sigma_1'$. Then since $s_1 \preceq_{\sigma_1'} t_1$, $s_1'.\alpha' \preceq_{\sigma'} t_1.\theta$. Further $\sigma' \in \sigma[w/y]$. Now the lemma holds with $P' \stackrel{s_1'.\alpha'}{\Longrightarrow} Q'$.

– $\alpha = (\hat{v})uv$: We only consider the cases where α is uy and $(\hat{v})yv$, $y \notin \{u, v\}$. The case where α is yy is similar to these two, and the case where α is uv for $y \neq u, v$ is simple. Note that we have σ_2 is identity, and therefore $\sigma = \sigma_1$ and $\alpha = \theta\sigma_1$.

  * $\alpha = uy$: From $\alpha = \theta\sigma_1$ we deduce $\theta = u_1v_1$, $\sigma_1(u_1) = u$, and $\sigma_1(v_1) = y$. Let $y' = \sigma_1'(v_1)$. From $\sigma_1' \in \sigma_1[w/y]$ it follows that $y' \in \{y, w\}$. From Lemmas 2.6.1 and 2.9 it follows $Q_1' \xrightarrow{uy'} Q'$ for some $Q' \in Q[w/y]_o$. Then we have $s_1'.uy' \preceq_{\sigma_1'} t_1.\theta$, because $\theta\sigma_1' = uy'$. The lemma holds with $P' \stackrel{s_1'.uy'}{\Longrightarrow} Q'$.

  * $\alpha = (\hat{v})yv$: From $\alpha = \theta\sigma_1$ we deduce $\theta = (\hat{v_1})u_1v_1$, $\sigma_1(u_1) = y$, $\sigma_1(v_1) = v$. By Lemma 2.6.1 we have $Q_1' \xrightarrow{(\hat{v})yv} Q'$ for some $Q' \in Q[w/y]_o$. Since $\sigma_1' \in \sigma_1[w/y]$, we have $\sigma_1'(u_1) \in \{w, y\}$. There are two cases. If $\sigma_1'(u_1) = y$: Then we have $s_1'.(\hat{v})yv \preceq_{\sigma_1'} t_1.\theta$, because $\theta\sigma_1' = (\hat{v})yv$. The lemma holds with $P' \stackrel{s_1'.(\hat{v})yv}{\Longrightarrow} Q'$. On the other hand, if $\sigma_1'(u_1) = w$, then by Lemma A.3, there is $P'' \in P'[w/y]_i$ such that $P'' \stackrel{s_1'.(\hat{v})wv}{\Longrightarrow} Q'$ (note that since t is $\rho \cup \{w\}$-normal $\{w\} \cap \hat{v} = \emptyset$). Note that $P'' \in P_1[w/y]_i$. Then we have $s_1'.(\hat{v})wv \preceq_{\sigma_1'} t_1.\theta$, because $\theta\sigma_1' = (\hat{v})wv$. The lemma holds with $P'' \stackrel{s_1'.(\hat{v})wv}{\Longrightarrow} Q'$.

- $\alpha = \tau$: Then $s_1 = s$, and $s_1 \preceq_\sigma t$. Then by induction hypothesis, there exist $P' \in P_1[w/y]_i$, $Q_1' \in Q_1[w/y]_o$, $\sigma' \in \sigma[w/y]$, such that $P' \stackrel{s'}{\Longrightarrow} Q_1'$ for $s' \preceq_{\sigma'} t$. From $Q_1 \xrightarrow{\tau} Q$, by Lemma 2.6.3, we have two cases:

– There is $Q' \in Q[w/y]_o$ such that $Q'_1 \xrightarrow{\tau} Q'$. The lemma follows trivially with $P' \stackrel{s'}{\Longrightarrow} Q'$.

– We have $Q \equiv (\nu \hat{z})R$, $w, y \notin \hat{z}$, and there is $R' \in R[w/y]_o$ such that $Q'_1 \xrightarrow{(\hat{z})\overline{w}z} \xrightarrow{yz} R'$. Then applying Lemma A.3 to $P' \stackrel{s'.(\hat{z})\overline{w}z.yz}{\Longrightarrow} R'$ we have, there is $P'' \in P'[w/y]_i$ such that $P'' \stackrel{s'.(\hat{z})\overline{w}z.wz}{\Longrightarrow} R'$. But then $P'' \stackrel{s'}{\Longrightarrow} (\nu \hat{z})R'$. Now, since $w, y \notin \hat{z}$, we have $(\nu \hat{z})R' \in Q[w/y]_o$. The lemma holds with $P'' \stackrel{s'}{\Longrightarrow} (\nu \hat{z})R'$, because $P'' \in P_1[w/y]_i$.

□

We define $\langle \hat{y} \rangle s$ as follows.

$$\langle \hat{y} \rangle s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } y \notin fn(s) \\ s_1.\overline{x}(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\ & s = s_1.\overline{x}y.s_2 \text{ and } y \notin fn(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

Proof of Lemma 2.16:

1. First we prove $P \stackrel{\sqsubset}{\sim}_\rho Q$ implies $(\nu x)P \stackrel{\sqsubset}{\sim}_{\rho - \{x\}} (\nu x)Q$. Suppose for an observer O such that $rcp(O) \cap (\rho - \{x\}) = \emptyset$, we have $(\nu x)P|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. Let z be fresh. Using Lemma A.1, we have $((\nu x)P|O)\{z/x\} \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. Since x is not free in $(\nu x)P$ we have $(\nu x)P|O\{z/x\} \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. Now x is not free in $O\{z/x\}$, and so we have $(\nu x)(P|O\{z/x\}) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. This implies $P|O\{z/x\} \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. But $P \stackrel{\sqsubset}{\sim}_\rho Q$, and $rcp(O) \cap \rho = \emptyset$. Therefore, $Q|O\{z/x\} \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. It follows that $(\nu x)(Q|O\{z/x\}) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. And since x is not free in $O\{z/x\}$, we also have $(\nu x)Q|O\{z/x\} \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. Since z is not free in O, we have $O\{z/x\}\{x/z\} = O$. Therefore, using Lemma A.1 again, we deduce $((\nu x)Q|O\{z/x\})\{x/z\} \stackrel{\overline{\mu}\mu}{\Longrightarrow}$, i.e. $(\nu x)Q|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$ since z is not free in $(\nu x)Q$.

   Now we prove $rcp(R) \cap \rho = \emptyset$ and $P \stackrel{\sqsubset}{\sim}_\rho Q$ imply $P|R \stackrel{\sqsubset}{\sim}_\rho Q|R$. Suppose for an observer O such that $rcp(O) \cap \rho = \emptyset$, we have $(P|R)|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. Then $P|(R|O) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$.

Now, $rcp(R|O) \cap \rho = \emptyset$. Then since $P \stackrel{\sqsubseteq}{\sim}_\rho Q$, we have $Q|(R|O) \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. This, in turn implies that $(Q|R)|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$ and the lemma follows.

2. Let $\overline{x}y|P \stackrel{s}{\Longrightarrow}$ where s is ρ-well-formed, and $t \in T(s,\rho)$. We have $\overline{x}w|P \in (\overline{x}y|P)[w/y]$. Then by Lemma A.4, it follows that there is $P' \in P[w/y]_i$ such that $\overline{x}w|P' \stackrel{s'}{\Longrightarrow}$ for some $s' \preceq t$. Now, $(\nu w)(\overline{x}w|P') \stackrel{\langle\{w\}\rangle s'}{\Longrightarrow}$, and $\langle\{w\}\rangle s' \preceq t$. Then it follows that $(\nu w)(\overline{x}w|\sum_{P' \in P[w/y]_i} P') \stackrel{\langle\{w\}\rangle s'}{\Longrightarrow}$. □

Proof of Lemma 2.19.1: Let $e(s\{z/y\}) \stackrel{r}{\Longrightarrow}$ for a ρ-well-formed r. Without loss of generality we can assume $bn(s) \cap \{z,y\} = \emptyset$, and s is ρ-normal. The proof is by induction on the length of s. The base case $s = \epsilon$ is obvious. Let $s = \alpha.s_1$, then $s\{z/y\} = \alpha\{z/y\}.s_1\{z/y\}$. For the induction step there are two cases depending on α.

1. $\alpha = (\hat{w})\overline{x}w$: We only consider the case $x = y$, which is central to the proof; the case $x \neq y$ is simpler. Then $\alpha\{z/y\} = (\hat{w'})\overline{z}w'$ where $w' = w\{z/y\}$, and $e(s\{z/y\}) = (\nu\hat{w'})(\overline{z}w'|e(s_1\{z/y\}))$. We consider the case $z \in \rho$ which is more interesting; the case $z \notin \rho$ is similar. Since $z \in \rho$ and r is ρ-well-formed, the message $\overline{z}w'$ cannot fire in $e(s\{z/y\}) \stackrel{r}{\Longrightarrow}$. So there are two possibilities.

   - $\overline{z}w'$ is consumed internally. Then $e(s_1\{z/y\}) \stackrel{r_1}{\Longrightarrow}$ for some $r_1 = r_2.zw'.r_3$ such that $r = \langle\hat{w'}\rangle(r_2.r_3)$. Since r is ρ-well-formed, we have r_1 is $(\rho \cup \hat{w'})$-well-formed. By induction hypothesis there is a $(\rho \cup \hat{w'})$-well-formed cotemplate r_1' such that $e(s_1) \stackrel{r_1'}{\Longrightarrow}$ and $e(r_1'\{z/y\}) \stackrel{r_1}{\Longrightarrow}$. Now, $e(s) \stackrel{r'}{\Longrightarrow}$ where $r' = (\hat{w})\overline{y}w.r_1'$. Note that r' is a cotemplate that is ρ-well-formed because $y \notin \rho$. Further, $e(r'\{z/y\}) = (\nu\hat{w'})(\overline{z}w'|e(r_1'\{z/y\}))$. Therefore, $e(r'\{z/y\}) \stackrel{r}{\Longrightarrow}$.
   - $\overline{z}w'$ is not consumed. Then $e(s_1\{z/y\}) \stackrel{r_1}{\Longrightarrow}$ for some r_1 such that $r = \langle\hat{w'}\rangle r_1$. Since r is ρ-well-formed, we have r_1 is $(\rho \cup \hat{w'})$-well-formed. By induction hypothesis there is a $(\rho \cup \hat{w'})$-well-formed cotemplate r_1' such that $e(s_1) \stackrel{r_1'}{\Longrightarrow}$ and $e(r_1'\{z/y\}) \stackrel{r_1}{\Longrightarrow}$. Now, $e(s) \stackrel{r'}{\Longrightarrow}$ where $r' = \langle\hat{w}\rangle r_1'$. Note that r' is a ρ-well-formed cotemplate. Further, $e(r'\{z/y\}) = (\nu\hat{w'})(e(r_1'\{z/y\}))$. Therefore, $e(r'\{z/y\}) \stackrel{r}{\Longrightarrow}$.

2. $\alpha = x(w)$: We only consider the more interesting case where $x = y$. Then $\alpha\{z/y\} = z(w)$ and $e(s\{z/y\}) = z(w).e(s_1\{z/y\}))$. Then $r = (\hat{v})zv.r_1$ for some r_1 such that $e(s_1\{z/y\}\{v/w\}) \stackrel{r_1}{\Longrightarrow}$. Note that r_1 is ρ-well-formed. Since s is cowell-formed, so is $s\{z/y\}$, and therefore w does not occur free as the subject of an input in $s_1\{z/y\}$. Further, since s is ρ-normal, $w \notin \rho$. Then, by induction hypothesis, $e(s_1\{z/y\}) \stackrel{r_2}{\Longrightarrow}$ for some ρ-well-formed cotemplate r_2 such that $e(r_2\{v/w\}) \stackrel{r_1}{\Longrightarrow}$. Now, applying the induction hypothesis again on $s_1\{z/y\}$, we get $e(s_1) \stackrel{r_3}{\Longrightarrow}$ for some ρ-well-formed cotemplate r_3 such that $e(r_3\{z/y\}) \stackrel{r_2}{\Longrightarrow}$. Now, the reader can verify the following claim: for cotemplates t_1, t_2, t_3, if $e(t_1) \stackrel{t_2}{\Longrightarrow}$ and $e(t_2) \stackrel{t_3}{\Longrightarrow}$ then $e(t_1) \stackrel{t_3}{\Longrightarrow}$. Using this claim and Lemma A.1 we conclude $e(r_3\{z/y\}\{v/w\}) \stackrel{r_1}{\Longrightarrow}$. Now, $e(s) \stackrel{r'}{\Longrightarrow}$ where $r' = y(w).r_3$. Since r_3 is ρ-well-formed cotemplate, so is r'. Further, $e(r'\{z/y\}) = z(w).e(r_3\{z/y\}) \stackrel{(\hat{v})zv}{\longrightarrow} e(r_3\{z/y\}\{v/w\}) \stackrel{r_1}{\Longrightarrow}$. Therefore, $e(r'\{z/y\}) \stackrel{r}{\Longrightarrow}$. □

Proof of Lemma 2.19.2: Following is a proof sketch. Since we work modulo alpha equivalence on traces, we assume $bn(r) \cap fn(P) = \emptyset$. If s and s' are alpha equivalent then so are $e(s)$ and $e(s')$. Then, since α equivalent processes have the same transitions, we can assume s is normal. The proof is by induction on the length of the computation $e(s) \stackrel{r}{\Longrightarrow}$. The base case is trivial. For the induction step we can write

$$e(s) \stackrel{\alpha}{\longrightarrow} Q \stackrel{r_1}{\Longrightarrow},$$

for some Q and r_1. We have three cases based on α:

- $\alpha = (\hat{y})\overline{x}y$: Then $s = \langle\hat{y}\rangle(s_1.\overline{x}y.s_2)$ for some s_1, s_2 where s_1 does not contain any inputs, and $Q = e(s_1.s_2)$. Further, since $P \stackrel{\langle\hat{y}\rangle(s_1.\overline{x}y.s_2)}{\Longrightarrow}$ we can show $P \stackrel{(\hat{y})\overline{x}y}{\Longrightarrow} P_1 \stackrel{s_1.s_2}{\Longrightarrow}$. Since $e(s_1.s_2) \stackrel{r_1}{\Longrightarrow}$, by induction hypothesis, $P_1 \stackrel{r_1}{\Longrightarrow}$. The lemma follows from $P \stackrel{(\hat{y})\overline{x}y.r_1}{\Longrightarrow}$.

- $\alpha = (\hat{y})xy$: Then $s = s_1.x(u).s_2$ for some s_1, s_2 such that $x \notin bn(s_1)$, and s_1 contains no inputs, and $Q = e(s_1.s_2\{y/u\})$. By normality of s, $u \notin n(s_1)$, and hence we can

write $Q = e((s_1.s_2)\{y/u\}))$. Further, from $P \stackrel{s_1.x(u).s_2}{\Longrightarrow}$, $u \notin n(s_1)$, and $x \notin bn(s_1)$ we also have $P \stackrel{x(u)}{\Longrightarrow} P_1 \stackrel{s_1.s_2}{\Longrightarrow}$. Further, since $\hat{y} \cap fn(P) = \emptyset$, $P \stackrel{(\hat{y})xy}{\Longrightarrow} P_1\{y/u\}$. Then by Lemma A.1 we have $P_1 \stackrel{(s_1.s_2)\{y/u\}}{\Longrightarrow}$. By induction hypothesis $P_1\{y/u\} \stackrel{r_1}{\Longrightarrow}$, and the lemma follows from $P \stackrel{(\hat{y})xy.r_1}{\Longrightarrow}$.

- $\alpha = \tau$: Then we have $e(s) \stackrel{(\hat{y})\overline{x}y.xy}{\longrightarrow} Q'$, $s = \langle\hat{y}\rangle(s_1.\overline{x}y.s_2.x(u).s_3)$, s_1 and s_2 contain only outputs, $Q' = e(s_1.s_2.s_3\{y/u\})$, and $Q = (\nu\hat{y})Q' = e(\langle\hat{y}\rangle(s_1.s_2.s_3\{y/u\}))$. Since $P \stackrel{\langle\hat{y}\rangle(s_1.\overline{x}y.s_2.x(u).s_3)}{\Longrightarrow}$, we have $P \stackrel{\langle\hat{y}\rangle(s_1.\overline{x}y.s_2.xy.s_3\{y/u\})}{\Longrightarrow}$ by Lemma A.1. The complementary input and output actions can be preponed so that $P \Longrightarrow P_1 \stackrel{\langle\hat{y}\rangle(s_1.s_2.s_3\{y/u\})}{\Longrightarrow}$. By induction hypothesis $P_1 \stackrel{r_1}{\Longrightarrow}$, and the lemma follows. □

Proof of Lemma 2.19.3: That $(\nu\hat{y})e(r) \stackrel{r'}{\Longrightarrow}$ is immediate. The proof of $(\nu\hat{y})e(r) = e(r')$ is by a straightforward induction on the length of r. The idea is to push $(\nu\hat{y})$ inwards as far as possible. To push across a restriction, we can use *I1, A3, A8* and *A19*, and the fact that $(\nu x)0 = 0$ which can be derived using *A2, A14, A19*. To push across a message we can use *A8*, and to push across an input we can use *A11*. If at any point, $(\nu\hat{y})$ cannot be pushed further, either case 2 of the definition of r' applies, or *A11* can be used and case 1 applies. If $(\nu\hat{y})$ can be pushed all the way in, we can use *A3* and $(\nu x)0 = 0$, and case 3 applies. □

Proof of Lemma 2.19.4: The proof is by induction on the length of s. Without loss of generality, we may assume $bn(s) \cap \{z, y\} = \emptyset$. The base case $s = \epsilon$ is obvious. For the induction step, there are three cases:

1. $s = (\hat{v})uv.s_1$: For $t' \in T(s\{z/y\}, \rho)$, we have $t' = (\hat{v'})u'v'.t'_1$, where $u' = u\{z/y\}$ $v' = v\{z/y\}$ and $t'_1 \in T(s_1\{z/y\}, \rho)$. By induction hypothesis, there is $t_1 \in T(s_1, \rho)$ such that $t_1\{z/y\} \preceq t'_1$ using only *L4*. But we have $(\hat{v})uv.t_1 \in T((\hat{v})uv.s_1, \rho)$, and $((\hat{v})uv.t_1)\{z/y\} = (\hat{v'})u'v'.t_1\{z/y\} \preceq (\hat{v'})u'v'.t'_1 = t'$, using only *L4*.

2. $s = \overline{u}(v).s_1$: For $t' \in T(s\{z/y\}, \rho)$ we have $t' = \overline{u'}(v).t'_1$, where $u' = u\{z/y\}$ and $t'_1 \in T(s_1\{z/y\}, \rho \cup \{v\})$. By induction hypothesis, there is $t_1 \in T(s_1, \rho \cup \{v\})$

such that $t_1\{z/y\} \preceq t'_1$ using only *L4*. But we have $\overline{u}(v).t_1 \in T(\overline{u}(v).s_1, \rho)$, and $(\overline{u}(v).t_1)\{z/y\} = \overline{u'}(v).t_1\{z/y\} \preceq \overline{u'}(v).t'_1 = t'$, using only *L4*.

3. $s = \overline{u}v.s_1$: There are two subcases.

   - $v = y$: Then $s\{z/y\} = \overline{u'}z.s_1\{z/y\}$ where $u' = u\{z/y\}$. There are two more subcases:

     – $z \in \rho$: For $t' \in T(s\{z/y\}, \rho)$ we have $t' = \overline{u'}(w).t'_1$, where w fresh, and $t'_1 \in T(s_2, \rho \cup \{w\})$ for some $s_2 \in s_1\{z/y\}[w/z]$. Using the fact that y cannot occur free in the input actions of s, we can show show $s_2 = s_3\{z/y\}$ for some $s_3 \in s_1[w/z]$. Clearly, s_3 does not contain free occurrences of y in input actions. Then by induction hypothesis, there is $t_1 \in T(s_3, \rho \cup \{w\})$ such that $t_1\{z/y\} \preceq t'_1$ using only *L4*. It is easy see that, since w is fresh, $t_1\{z/w\} \in T(s_1, \rho)$. Then we have $\overline{u}y.t_1\{z/w\} \in T(s, \rho)$. Then $(\overline{u}y.t_1\{z/w\})\{z/y\} = \overline{u'}z.t_1\{z/w\}\{z/y\} = \overline{u'}z.t_1\{z/y\}\{z/w\} \prec \overline{u'}(w).t_1\{z/y\} \preceq \overline{u'}(w).t'_1 = t'$, where the relation $\prec$ is by *L4*.

     – $z \notin \rho$: For $t' \in T(s\{z/y\}, \rho)$ we have $t' = \overline{u'}z.t'_1$, where $t'_1 \in T(s_1\{z/y\}, \rho)$. By induction hypothesis, there is $t_1 \in T(s_1, \rho)$ such that $t_1\{z/y\} \preceq t'_1$ using only *L4*. But we have $\overline{u}y.t_1 \in T(\overline{u}y.s_1, \rho)$, and $(\overline{u}y.t_1)\{z/y\} = \overline{u'}z.t_1\{z/y\} \preceq \overline{u'}z.t'_1 = t'$ using only *L4*.

   - $v \neq y$: Then $s\{z/y\} = \overline{u'}v.s_1\{z/y\}$ where $u' = u\{z/y\}$. There are two subcases.

     – $v \notin \rho$: Then for $t' \in T(s\{z/y\}, \rho)$ we have $t' = \overline{u'}v.t'_1$, where $t'_1 \in T(s_1\{z/y\}, \rho)$. By induction hypothesis, there is $t_1 \in T(s_1, \rho)$ such that $t_1\{z/y\} \preceq t'_1$ using only *L4*. But we have $\overline{u}v.t_1 \in T(s, \rho)$, and $(\overline{u}v.t_1)\{z/y\} = \overline{u'}v.t_1\{z/y\} \preceq \overline{u'}v.t'_1 = t'$ using only *L4*.

     – $v \in \rho$: Then for $t' \in T(s\{z/y\}, \rho)$ we have $t' = \overline{u'}(w).t'_1$, where w fresh, $t'_1 \in T(s_2, \rho \cup \{w\})$ for some $s_2 \in s_1\{z/y\}[w/v]$. Using the fact that y occurs free only in output actions of s_1, we can show $s_2 = s_3\{z/y\}$ for some

$s_3 \in s_1[w/v]$. Clearly, y does not occur free in input actions of s_3. Then by induction hypothesis, there is $t_1 \in T(s_3, \rho \cup \{w\})$ such that $t_1\{z/y\} \preceq t_1'$ using only *L4*. But we have $\overline{u}(w).t_1 \in T(s, \rho)$, $(\overline{u}(w).t_1)\{z/y\} = \overline{u'}(w).t_1\{z/y\} \preceq \overline{u'}(w).t_1' = t'$, using only *L4*. □

APPENDIX B

Executable Specification in Maude

B.1 Specification of Asynchronous π-Calculus

Following is the module `APISEMANTICS` that contains the rewrite rules for the operational semantics of asynchronous π-calculus (Table 2.1). The function `genQid` used in the condition of the last `Res` rule generates an identifier that is fresh, i.e. an identifier not used to construct channel names in the set passed as the argument to the function.

```
mod PISEMANTICS is
   inc PISYNTAX .
   inc CHANSET .
   inc TRACE .
   sorts EnvTrm TraceTrm .
   subsort EnvTrm < TraceTrm .

   op  [_]_ : Chanset Trm -> EnvTrm [frozen] .
   op  {_}_ : Action TraceTrm -> TraceTrm [frozen] .
   op  notinfn : Qid Trm -> Prop .

   vars N : Nat .          vars X Y Z : Qid .
   vars CX CY : Chan .     var  CS CS1 CS2 : Chanset .
   vars A : Action .       vars P1 Q1 P Q : Trm .
   var  SUM : SumTrm .     var IO : ActionType .
```

```
eq  notinfn(X,P) = not X{0} in freenames(P) .

rl [Inp] : [CY CS] (CX(X) . P)  =>
               {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

rl [Inp] : [CY CS] ((CX(X) . P) + SUM) =>
               {f(i,CX,CY)} ([CY CS] ([X := CY] P)) .

rl [Tau] : [CS] (tau . P) => { tauAct } ([CS] P) .

rl [Tau] : [CS] ((tau . P) + SUM) => { tauAct } ([CS] P) .

crl [BInp] : [CS] P => {b(i,CX,'u)} ['u{0} [shiftup 'u] CS] P1
             if (not flag in CS) /\
                CS1 := flag 'u{0} [shiftup 'u] CS  /\
                [CS1] [shiftup 'u] P => {f(i,CX,'u{0})} [CS1] P1 .

rl  [Out] : [CS] CX < CY > => { f(o,CX,CY) } ([CS] nil) .

crl [Par] : [CS] (P | Q) => {f(IO,CX,CY)} ([CS] (P1 | Q))
            if [CS] P => {f(IO,CX,CY)} ([CS] P1) .

crl [Par] : [CS] (P | Q) =>
               {b(IO,CX,Y)} [Y{0} ([shiftup Y] CS)] (P1 | [shiftup Y] Q)
            if [CS] P => {b(IO,CX,Y)} ([CS1] P1) .

crl [Com] : [CS] (P | Q) => {tauAct} ([CS] (P1 | Q1))
            if [CS] P => {f(o,CX,CY)} ([CS] P1)  /\
                  [CY CS] Q => {f(i,CX,CY)} ([CY CS] Q1) .

crl [Close] : [CS] (P | Q) => {tauAct} [CS] new [Y] (P1 | Q1)
              if [CS] P => {b(o,CX,Y)} [CS1] P1 /\
                 [Y{0} [shiftup Y] CS] [shiftup Y] Q =>
                                    {f(i,CX,Y{0})} [CS2] Q1 .
```

```
  crl [Res] : [CS] (new [X] P) =>
                 {[shiftdown X] f(IO,CX,CY)} [CS] (new [X] P1)
              if CS1 := [shiftup X] CS  /\
                 [CS1] P => {f(IO,CX,CY)} [CS1] P1  /\
                 (not X{0} in (CX CY)) .

  crl [Res] : [CS] (new [X] P) => {tauAct} [CS] (new [X] P1)
              if [CS] P => {tauAct} [CS] P1 .

  crl [Res] : [CS] (new [X] P) =>
              {[shiftdown X] b(o,CX,Z)} [Z{0} CS] new[X]([ Y := Z{0} ] P1)
              if Z := genQid(X{0} CS freenames(P)) /\
                 [[shiftup X] CS] P => {b(o,CX,Y)} [CS1] P1 /\
                 X{0} =/= CX .

  crl [Open] : [CS] (new[X] P) => {[shiftdown X] b(o,CY,X)} [X{0} CS1] P1
               if CS1 := [shiftup X] CS /\
                  [CS1] P => {f(o,CY,X{0})} [CS1] P1 /\ X{0} =/= CY .

  crl [If] : [CS1] (if CX = CX then P else Q fi) => {A} [CS2] P1
             if [CS1] P => {A} [CS2] P1 .

  crl [Else] : [CS1] (if CX = CY then P else Q fi) => {A} [CS2] Q1
               if CX =/= CY /\ [CS1] Q => {A} [CS2] Q1 .

  crl [Rep] : [CS1] (! P) => {A} [CS2] P1
              if [CS1] (P | (! P)) => {A} [CS2] P1 .

endm
```

B.2 Specification of A$\pi_{\neq}$

Following is the module that specifies behavior definitions and operations on them. For a tuple of identifiers `QT` and natural `n`, `distinct?(QT)` checks if each component of `QT` is distinct, `pick(QT,n)` returns in the n^{th} component of `QT`, and `tochanset(QT,n)` returns a set of channels that contains the channel `X{n}` for each identifier `X` in `QT`.

```
fmod APICONTEXT is
  including APISYNTAX .
  protecting QIDTUPLE .

  sorts  Defn Context .
  subsort  Defn < Context .

  ************************************
  *** Constructors for definitions
  ************************************

  op  _:=(_;_)_  :  Qid QidTuple QidTuple Term -> Defn [prec 8] .
  op  emptycontext  : -> Context .
  op  _,_  :  Context Context -> Context [assoc id: emptycontext prec 9] .
  op  context : -> Context .

  ***********************************
  *** Checking for well-definedness
  ***********************************
  var X :  Qid .
  var CX  :  Chan .
  var P  :  Term .
  vars B B' :  Qid .
  vars QT QT1 QT2  :  QidTuple .

  op  defined?  :  Qid Context -> Bool .
  op  unique?  :  Context -> Bool .
  op  wellformed?  :  Context -> Bool .
```

```
ops arity1 arity2 :  Qid Context -> Nat .

**********************************
*** check if a constant is defined
**********************************
eq  defined?(B, emptyContext) = false .
eq  defined?(B, (B' := (QT1 ; QT2) P , D)) = B == B' or defined?(B, D) .

******************************************************
*** check if all constants have at most one definition
******************************************************
eq  unique?(emptyContext) = true .
eq  unique?((B := (QT1 ; QT2) P , D)) = (not defined?(B, D)) and unique?(D) .

*************************************************************************
*** check for well-formedness of definitions, arities, freenames and such ...
*************************************************************************
eq  wellformed?(emptyContext) = true .
eq  wellformed?((B := (QT1 ; QT2) CX(X) . P , D')) =
                        length(QT1) > 0 and s(s(s(0))) > length(QT1) and
                        distinct?(QT1 QT2) and
                        pick(QT1 , s(0)){0} == CX and
                        freenames(CX(X) . P) subset
                           (tochanset(QT1 , 0) cup tochanset(QT2 , 0)) .

eq  arity1(B , (B' := (QT1 ; QT2) P, D)) = if B == B' then length(QT1)
                                           else arity1(B , D) fi .
eq  arity2(B , (B' := (QT1 ; QT2) P, D)) = if B == B' then length(QT2)
                                           else arity2(B , D) fi .


**************************************
*** instantiating behavior definitions
**************************************
op  defn : Qid Context -> Defn .
```

```
  op  inst : Defn ChanTuple -> Term .
  op  subst : Term QidTuple ChanTuple -> Term .

  var CT : ChanTuple .

  eq  defn(B, (B' := (QT1 ; QT2) P , D)) = if B == B' then B' := (QT1 ; QT2) P
                                           else defn(B, D) fi .

  eq  inst(B := (QT1 ; QT2) P, CT) = subst(P , QT1 QT2 , CT) .

  eq  subst(P , emptyQidTuple , emptyChanTuple) = P .
  eq  subst(P , X QT , CX CT) = [X := CX] subst(P , QT , CT) .

endfm
```

Vita

Prasannaa Thati was born in Rayadurg, India, on September 15, 1976. He graduated from the Indian Institute of Technology, Kanpur in 1997 with a BTech in Computer Science and Engineering and a minor in Discrete Mathematics. He then joined the University of Illinois at Urbana Champaign for graduate studies in Computer Science under the supervision of Prof. Gul Agha. He obtained a MS in Computer Science in August 2000, an MS in Mathematics in December 2002, and a PhD in Computer Science in October 2003. Prasannaa Thati's primary research interest is in formal specification and verification of computer systems.