View-centric Reasoning about Parallel and Distributed Computation

by

Marc L. Smith
B.S. University of Central Florida, 1986
M.S. University of Central Florida, 1993

A dissertation submitted in partial satisfaction of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2000

Major Professors:
Rebecca J. Parsons
Charles E. Hughes

Abstract

The development of distributed applications has not progressed as rapidly as its enabling technologies. In part, this is due to the difficulty of reasoning about such complex systems. In contrast to sequential systems, parallel systems give rise to parallel events, and the resulting uncertainty of the observed order of these events. Loosely coupled distributed systems complicate this even further by introducing the element of multiple imperfect observers of these parallel events. The goal of this dissertation is to advance parallel and distributed systems development by producing a parameterized model that can be instantiated to reflect the computation and coordination properties of such systems. The result is a model called paraDOS that we show to be general enough to have instantiations of two very distinct distributed computation models, Actors and tuple space. We show how paraDOS allows us to use operational semantics to reason about computation when such reasoning must account for multiple, inconsistent and imperfect views. We then extend the paraDOS model with an abstraction to support composition of communicating computational systems. This extension gives us a tool to reason formally about heterogeneous systems, and about new distributed computing paradigms such as the multiple tuple spaces support seen in Sun's JavaSpaces and IBM's T Spaces.

To Susan &

Matthew

Acknowledgments

Many people, in many different ways, helped me to reach this point. In no particular order of importance, I begin with AT&T, my employer. My management, who supported me to get into the Doctoral Support Program, included Roger Jasko, Terry Burnett, John Pasqua, and Rino Bergonzi. Jim Kingsley, who preceded Roger; and Karl Parks and Rick Fisher, who succeeded Roger, also deserve my thanks. As far as I'm concerned, Roger will never be allowed to pay for a cup of coffee in my presence! A special word of thanks to Jane Price, my DSP coordinator until almost the very end, whose advice and support helped guide me through this long process. I would also like to thank Carol Lee, Donna Ronco, Penny Gilman, Patty O'Dell, and Rick Carpenter for their support, encouragement, and friendship over the years.

I am fortunate to have had Drs. Rebecca Parsons and Charles Hughes as my research committee co-chairs. While preparing for the qualifier exams, Dr. Parsons helped me turn on the first "lightbulb" in my understanding of computability theory, which set into motion the toppling of the rest of the qualifier dominos. As my friend, Dr. Parsons has been a source of strength and inspiration. My history with Dr. Hughes goes back to my sophomore year machine organization class. Over the years, he has been my professor, friend, mentor, and more; the one constant during this time of growth and change in my life. Dr. Hughes helped me map out a study plan to help prepare me for the qualifiers, and also permitted me to audit his class in Formal Languages and Automata

Theory. Together as my co-chairs, Drs. Parsons and Hughes formed the core of a formidable research committee to guide and support me through my research journey.

Several faculty members from UCF deserve my thanks for their contributions to my education. Dr. Mostafa Bassiouni, professor for three of my graduate classes, and a member of my research committee, challenged me to become a much better student than I ever was before taking his classes. Dr. Ronald DeMara, external member (at first) of my research committee, contributed several insights, references, and fine points to the body of my research. Dr. Sheau-Dong Lang permitted me to audit his course in Concrete Math while I was preparing for qualifier exams, and exceeded my expectations for the subject both in his lectures, and in his insistence that I turn my work into him for grading! Dr. Ronald Dutton allowed me to audit his course in Algorithm Design and Analysis, not once but twice, providing inspired lectures and insights. Finally, to Dr. Narsingh Deo, my professor for two courses in parallel architecture, algorithms, and analysis, is a gifted and inspiring professor, whose passion for teaching was evident in every lecture — and every proof from “first principles”!

To Tiffani Williams, who never failed to point out my “options,” to Keith Garfield, whose encouragement the second time around helped me believe I could pass qualifiers, and to the other students in the Evolutionary Computing Lab at UCF, Jaren Johnston and Paulius Micikevicius, thanks for thought provoking conversation, sometimes helpful, sometimes controversial, sometimes distracting, always enjoyable! In the Computer Science office, Connie Weiss, Susan Brooks, Heather Oakes, Laury Anthony, and Sonja Rosignol helped me countless times with logistics, listening, and chocolate.

Scott Nolan has been my good friend and frequent classmate since before we began graduate school together. During my time of need, Scott was my roommate, confidant, and voice of reason. Steve Mackrides and John Baptiste believed in and encouraged me when I almost gave up on this goal, and helped me go for it more than they knew. I would not have passed the qualifier exams were it not for Mark Kilby. Thank you, Mark, for your help as a study partner, friend, and soul-mate.

My most special thanks goes to my wife Susan, whose understanding, support, patience, and love made completing this dissertation possible. Matthew may be too young to remember this time in his life, when his daddy would come home to be with him for dinner, then go back out to school after he went to bed. Often, when I was tired, his picture and smile would give me the motivation to keep working. My brother Ira also played an important role during this time of my life, and for his support I am grateful. I am also happy to be able to share this milestone with my mother, whose love defies the bounds of time and space. Finally, to my brother and sister Michael and Alicia, who are very special to me, and who will never believe I am no longer a student, and to my father and Jan. I love you all.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The development of distributed applications has not progressed as rapidly as its enabling technologies. In part, this is due to the difficulty of reasoning about such complex systems. In contrast to sequential systems, parallel systems give rise to parallel events, and the resulting uncertainty of the observed order of these events. Loosely coupled distributed systems complicate this even further by introducing the element of multiple imperfect observers of these parallel events. Such observers are capable of seeing different views of the same parallel event.

In the opening paragraph, we alluded to three important characteristics that need to be addressed in models of concurrent computation. First, there is the nondeterminism of what events might occur next in a system of concurrent processes. Next, there is the requirement to represent any event simultaneity that does occur. Finally, there is the need to represent the observers' different potentially imperfect views of simultaneously occurring events.

The goal of this dissertation is to advance parallel and distributed systems development by producing a parameterized model that can be instantiated to reflect the computation and coordination properties of such systems, by supporting nondeterminism, parallel events, and views. The result is a model called paraDOS that we show to be general enough to have instantiations of two very distinct distributed computational models, Actors and tuple space. We show how para-

DOS allows us to use operational semantics to reason about computation when such reasoning must account for multiple, inconsistent and imperfect views. We then extend the paraDOS model with an abstraction to support composition of communicating computational systems. This extension gives us a tool to reason formally about heterogeneous systems, and about new distributed computing paradigms such as the multiple tuple spaces support seen in Sun's JavaSpaces and IBM's T Spaces.

1.1 Dissertation Outline

We ground our research in Chapter 2 with important background information, including the role of models and abstraction in the field of Computer Science, paying particular attention to models of computation that play a role in our research. Chapter 3 introduces the concepts that are important to understanding paraDOS, and how these concepts fit together. We present the uninstantiated paraDOS model, with formal definitions, in Chapter 4. Chapters 5 and 6 are the core of our theoretical work, presenting instantiations of paraDOS for Actors and Linda (the canonical example of a tuple space language), respectively. We establish the soundness of these two instantiations of paraDOS by proving two theorems based on equivalences to established operational semantics for Actors and Linda. We reveal paraDOS parameters and the particularly important composition parameter in Chapters 7 and 8, respectively. Chapter 9 gives an extensive treatment of reasoning about properties of computation, and exercises an instance of paraDOS to reason about a Linda definition considered to be ambiguous prior to our research. Finally, we conclude in Chapter 10, presenting a summary of our major contributions and potential future work.

CHAPTER 2

Background

This chapter considers the role of formal models and abstractions in Computer Science, and an approach to describe them. Section 2.1 begins with a discussion of models and abstractions. Sections 2.2 through 2.4 present three diverse computational models: Actors, Linda, and CSP. These three models support different abstractions of concurrency and are the basis and inspiration for much of the research presented in this dissertation. Section 2.6 introduces operational semantics, the tool used to realize paraDOS, the model developed in this dissertation. Finally, Section 2.7 presents related work.

2.1 Models and Abstraction

Scientists rely on models to describe, explain, and predict phenomena. The process that develops such models is one of iterative refinement, involving careful design and verification. Oxford [Oxf97] defines a model as a "simplified description of a system ... to assist calculations and predictions." This simple definition reveals two important aspects of any model, its abstraction and its purpose. Computer Science is not the science of computers; it is the science of models. It could exist independent of the invention of computers and, in fact, models largely motivated the invention of computers.

Typically, the purpose of a new model influences its level of abstraction. One way to verify a new model is to prove its equivalence to an established one. Sometimes, the design of two or more existing, equivalent models suggests the design of a new, more general model. In this sense, the more general a model, the more purposes it serves. Computational models can be predictive, descriptive, or used for reasoning about properties of computation. The purpose of paraDOS is the latter.

Direct observation of a concurrent computer program is problematic, impractical, and not conducive to reasoning about properties of concurrent systems in general. Limitations of human observation include resource availability, endurance, and consistency (both rate and reproducibility). Furthermore, a single correct observation does not exist; multiple views are a consequence of observing systems with multiple concurrent processes. Any model for reasoning about properties of concurrent systems must adequately address the complexities resulting from multiple views of computation.

A concurrent program, in general, requires communication and coordination. Mechanisms to support communication vary from shared memory to message passing to combinations of both of these mechanisms. The design of paraDOS needed to employ a communication abstraction sufficient to support the many varieties of concurrency about which we wish to reason, especially those arising in distributed computation.

A critical designation for paraDOS was the selection of an appropriate level of abstraction for observable events, the primitives we have chosen for formal reasoning. Sequential models of computation often consider the details of internal computational states, transitions, or subexpression evaluation, but this level of granularity is not desirable for reasoning about properties of concurrent systems. Instead, we are inspired by the approach taken in computability theory, and have extended its notion of "input/output behavior" to include interprocess communication.

2.2 Actors

The Actors model of concurrent compuatation is due to Agha [Agh86]. We present the instantiation of paraDOS for Actors, P^{act}, in Chapter 5. Actors is an elegant model of concurrency based on message passing behavior. At the core of this model is the concept of computational agents (actors). The remainder of this section discusses actors and actor computation in sufficient detail to enable the reader to understand the semantics presented in Chapter 5.

Actors compute in response to messages they receive. For each message an actor receives, it can (based on its behavior at the time it receives this message) send messages to other actors, create new actors, and specify its own replacement behavior (not necessarily in this or any other prescribed order). There is still much to say about implementation assumptions and the implications of these requirements (in terms of what is and is not specified). Let's consider each of these requirements in turn, and discuss briefly some of the implications.

An actor can send messages to other actors. This is the only way one actor can affect the behavior of another. While there is a guarantee of delivery for all sent messages, there is no guarantee that the order of receipt will be the same as that of transmission, or even that the order of receipt will be the same for all recipients. Thus, the promise of delivery is the total extent of fairness required. Furthermore, actors communicate asynchronously since synchronous communication would limit parallelism and, in a distributed system, be problematic to implement. Asynchronous messages also increase the level of nondeterminism in the actor model, an important consequence of PDSs.

An actor can create new actors. Any sequence of independent expressions that can be computed in parallel can take advantage of new actors to do so. Subexpression results can be communicated back to other actors waiting for those results. Compilers can perform sub-expression analysis to maximize parallelism, based on hardware and run-time constraints, so as not to burden the programmer. Thus, the Actors model does not unnecessarily constrain otherwise inherent parallelism, or distributivity.

An actor can specify its own replacement behavior. This replacement behavior will govern what that actor does with the next message it receives. In this way, actors can be history sensitive. An actor's actions are a function of its behavior at the time a message is received and the content of the message.

Actors is a seductive model in that it embodies three simple requirements, yet contains all the power and complexity inherent in concurrent computation. Given the proliferation of requirements and specifications for other concurrent models that possess no greater parallel and distributed processing capabilities than actors, the Actors model is the logical choice for the first instantiation of paraDOS.

2.3 Linda

The tuple space model and Linda language are due to Gelernter [Gel85]. We present instantiations of paraDOS for Linda in Chapter 6. Linda is very different from pure message passing-based models (e.g., Actors); therefore they represent an important test of the diversity of paraDOS's instantiation capability. The current popularity of commercial tuple space implementations, such as Sun's JavaSpaces [FHA99] and IBM's T Spaces [WML98], contributes to the relevance of Linda instances of paraDOS.

Linda is not a complete programming language; it is a communication and coordination language. Linda is intended to augment existing computational languages with its coordination primitives to form comprehensive parallel and distributed programming languages. The Linda coordination primitives are `rd`(), `in`(), `out`(), and `eval`(). The idea is that multiple Linda processes share a common space, called a tuple space, through which the processes are able to communicate and coordinate using Linda primitives.

A tuple space may be viewed as a container of tuples, where a tuple is simply a group of values. A tuple is considered active if one or more of its values is currently being computed, and passive if all of its values have been computed. A Linda primitive manipulates tuple space according to the template specified in its argument. Templates represent tuples in a Linda program. A template extends the notion of tuple by distinguishing its passive values as either *formal* or *actual*, where formal values, or *formals*, represent typed wildcards for matching. Primitives `rd`() and `in`() are synchronous, or blocking operations; `out`() and `eval`() are asynchronous.

The `rd()` and `in()` primitives attempt to find a tuple in tuple space that matches their template. If successful, these primitives return a copy of the matching tuple by replacing any formals with actuals in their template. In addition, the `in()` primitive, in the case of a match, removes the matching tuple from tuple space. In the case of multiple matching tuples, a nondeterministic choice determines which tuple the `rd()` or `in()` operation returns. If no match is found, both operations block until such time as a match is found. The `out()` operation places a tuple in tuple space. This tuple is a copy of the operation's template. Primitives `rd()`, `in()`, and `out()` all operate on passive tuples.

All Linda processes reside as value-yielding computations within the active tuples in tuple space. Any Linda process can create new Linda processes through the `eval()` primitive. Execution of the `eval()` operation places an active tuple in tuple space, copied from the template. When a process completes, it replaces itself with a passive value within its respective tuple; when all processes within a tuple replace themselves with values, the formerly active tuple becomes passive. Only passive tuples are visible for matching by the `rd()` and `in()` primitives; thus active tuples are invisible.

In the almost two decades since Gelernter first conceived the Linda language and tuple space, the computer world has evolved dramatically. During most of this time, Linda development and research has primarily been an academic exercise. Only recently has the tuple space approach to building distributed systems gained widespread acceptance. It is instructive to look at Linda's history to understand its current role in distributed computing paradigms.

The Linda language has several desirable properties that seem particularly well-suited for distributed computing. Briefly, since tuples are addressed associatively, through matching, tuple space is a platform independent shared memory.

Unlike message passing systems where a sender must typically specify a message's recipient, tuple space acts as a conduit for the generation, use, and consumption of information between distributed processes. Information generators do not need to know who their consumers will be, nor do information consumers need to know who generated the information they consume. Gelernter calls this property *communication orthogonality*. Additionally, tuples may be generated long before their consumers exist, and tuples may be copied or consumed long after their generators cease to exist. This property is *time independence*.

When distributed computing didn't seem to be making great progress, the focus of Linda research shifted to parallel computing. The difference between distributed and parallel computing is loosely coupled versus tightly coupled processors, respectively. Linda's properties serve parallel computing well, with a natural notion for barrier synchronization and heterogeneity.

In the early nineties, Internet usage began to enter the mainstream of technology with the advent of the world wide web, browsers, Java, and smart devices. What was missing before was network ubiquity, a platform-independent language, and of course, a pervasive motivation. The motivation came when embedded systems migrated from the military to the general public in the form of smart appliances. For the first time, embedded microprocessors, such as those found in telephones, televisions, toaster ovens, and automobiles, had an external interface. The subsequent desire to network and control these devices remotely led to the need for a simple, yet powerful, protocol to enable this technology. Researchers at Sun Microsystems and IBM turned to Gelernter's Linda language and tuple spaces as the basis for developing their new distributed programming tools. Tuple space has returned to its roots, and is now the focus of distributed computing once again.

2.4 CSP

Communicating Sequential Processes (CSP) is due to Hoare [Hoa85]. CSP is a model for reasoning about concurrency; it provides an elegant mathematical notation and set of algebraic laws for this purpose. The inspiration for developing paraDOS based on observable events and the notion of event traces comes from CSP.

CSP views concurrency, as its name implies, in terms of communicating sequential processes. A computational process, in its simplest form, is described by a sequence of observable events. In general, process descriptions also benefit from Hoare's rich process algebra. The CSP process algebra is capable of expressing, among other things, choice, composition, and recursion. The history of a computation is recorded by an observer in the form a sequential trace of events. Events in CSP are said to be offered by the environment of a computation; therefore, they occur when a process accepts an event at the same time the event is offered by the environment.

When two or more processes compute concurrently within an observer's environment, the possibility exists for events to occur simultaneously. CSP has two approaches to express event simultaneity in a trace: synchronization and interleaving. Synchronization occurs when an event e is offered by the environment of a computation, and event e is ready to be accepted by two or more processes in the environment. When the observer records event e in the trace of computation, the interpretation is that all those processes eligible to accept e participate in the event.

The other form of event simultaneity, where two or more distinct events occur simultaneously, is recorded by the observer in the event trace via arbitrary interleaving. For example, if events e_1 and e_2 are offered by the environment, and two respective processes in the environment are ready to accept e_1 and e_2 at the same time, the observer may record either e_1 followed by e_2, or e_2 followed by e_1. In this case, from the trace alone, we can not distinguish whether events e_1 and e_2 occurred in sequence or simultaneously. CSP's contention, since the observer must record e_1 and e_2 in some order, is that this distinction is not important.

CSP's algebraic laws control the permissible interleavings of sequential processes, and support parallel composition, nondeterminism, and event hiding. Important sets within the CSP algebra are the traces, refusals, and failures of a process. The set of traces of a process P represents the set of all sequences of events in which P can participate if required. A refusal of P is an environment — a set of events — within which P can deadlock on its first step. The set of refusals of P represents all environments within which it is possible for P to deadlock. The set of failures of P is a set of trace-refusal pairs, indicating the traces of P that lead to the possibility of P deadlocking.

Reasoning about a system's trace is equivalent to reasoning about its computation. CSP introduces specifications, or predicates, that can be applied to individual traces. To assert a property is true for a system, the associated predicate must be true for all possible traces of that system's computation. Examples of elegant CSP predicates include those that test for properties of nondivergence or deadlock-freedom in a system. Hoare's CSP remains an influential model for reasoning about properties of concurrency. Recent contributions to the field of CSP research include Roscoe [Ros98] and Schneider [Sch00].

2.5 Composition

The conventional notion of composition refers to sequential composition. For example, in imperative programming languages, a common way to compose two or more individual statements involves delimiting with semicolons (e.g., $s_1; s_2$). The semantics of functional programming languages provides for composition through the linking of output values to input values in function application (e.g., $f(g(x))$, where the output value from function $g()$ becomes the input value to function $f()$).

For a non-programming language example, consider the Unix operating system. Unix provides numerous facilities for command composition, including the semicolon (;) and the pipe symbol (|), both of which are forms of composition. The semicolon is an example of sequential composition; for "a; b", command a executes, then command b executes. The pipe is an example of composition that permits concurrency; for "$a \mid b$", a's output becomes b's input, and a and b may run concurrently, subject to b blocking if it needs input from a that has yet to be produced.

Sequential composition is one possible restriction of parallel composition. When we discuss composition within the context of paraDOS, we refer to the more general notion of concurrent, or parallel composition [Mil89, CT90, CT92, FOT92]. Parallel composition provides for the concurrent computation of composed components. One definition of a distributed system is the composition of multiple, loosely coupled *sequential* processes that communicate and coordinate to perform some computation. A more general definition provides for the composition of multiple, loosely coupled distributed systems. Since one of the main goals for paraDOS is to be a general model, we sought to capture the essence of general composition, not only across instances of paraDOS, but with respect to the more general, recursive notion of composition possible in distributed systems.

2.6 Operational Semantics

In this dissertation, we employ operational semantics to develop a general computational model for concurrency. By general we mean a parameterized model capable of instantiation into multiple parallel and distributed systems. Thus, our goal is a model that goes beyond describing the meanings of programs for a particular programming language; we intend paraDOS to be generally applicable to a broad scope of computational systems and paradigms. The success of our research provides further evidence of the utility of operational semantics as an effective means to develop elegant models of computation that support reasoning about the modeled systems.

This section provides a brief introduction to operational semantics. The remainder of this section is organized as follows: Section 2.6.1 defines operational semantics. Sections 2.6.2 and 2.6.3 present two important contributions to the field of operational semantics, Landin's SECD machine and Plotkin's structural operational semantics.

2.6.1 Definition

The field of operational semantics encompasses any formal method used to describe the meaning of a program through the changes its execution makes to the state of some computational model [SK95]. The following definition is from Howe [How93]:

Definition 1 (*operational semantics*) An operational semantics is a set of rules specifying how the state of an actual or hypothetical computer changes while executing a program. The overall state is typically divided into a number of components, e.g. stacks, heaps, registers, etc. Each rule specifies certain preconditions on the contents of some components and their new contents after the application of the rule.

An operational semantics may take many forms, specifying a formal or informal model of computation; it is defined at a level of abstraction appropriate for the model's purpose. Important references for work in operational semantics include Dijkstra [Dij71], Landin [Lan64], Kahn [Kah87], Plotkin [Plo81], Marcotty, *et al* [MLB76], and Hennessy [Hen90]. The remaining subsections present Landin's and Plotkin's respective contributions to the field of operational semantics.

2.6.2 The SECD Machine

The first example is a classic use of operational semantics, the SECD machine by Peter Landin [Lan64]. The purpose of the SECD machine is to evaluate lambda expressions. As a result, the computational techniques employed by the SECD machine have been used in implementations of functional programming languages. SECD's name comes from the names of the four stacks which comprise the machine's configuration, or state:

S for Stack A structure for storing partial results awaiting subsequent use.

E for Environment A collection of bindings of values (actual parameters) to variables (formal parameters).

C for Control A stack of lambda expressions yet to be evaluated plus a special symbol "@" meaning that an application can be performed; the top expression on the stack is the next one to be evaluated.

D for Dump A stack of complete states corresponding to evaluations in progress but suspended while other expressions (inner redexes) are evaluated.

The notation for a state, then, is $cfg(S, E, C, D)$. Finally, SECD has a transition function that maps current states to next states. Formally, we specify this transition function by the mapping *transform* : State $\rightarrow$ State. Figure 2.1 contains the algorithm for the SECD transition function, as specified in Slonneger and Kurtz [SK95]. The SECD machine starts in an initial state with the C stack containing the lambda expression to be evaluated, and the S, E, and D stacks empty. A final state (if one exists for a given lambda expression) is recognized by empty C and D stacks; the result is on top of the S stack. Implementing the SECD machine in Scheme was an important personal milestone in the author's understanding of operational semantics. That implementation is in Appendix A.

2.6.3 Structural Operational Semantics

The SECD machine demonstrates one form of operational semantics, whose purpose is the specification of an abstract machine capable of carrying out the mechanical evaluation of lambda expressions. Another form of operational semantics, developed by Gordon Plotkin [Plo81], is called structural operational semantics. Structural operational semantics presents state transitions in the form of *transform* cfg(S, E, C, D) =

(1) if *head*(C) is a constant
    then cfg([*head*(C) | S], E, *tail*(C), D)

(2) else if *head*(C) is a variable
    then cfg([E(*head*(C)) | S], E, *tail*(C), D)

(3) else if *head*(C) is an application (Rator Rand)
    then cfg(S, E, [Rator,Rand,@ | *tail*(C)], D)

(4) else if *head*(C) is a lambda abstraction λV . B
    then cfg([closure(V,B,E) | S], E, *tail*(C), D)

(5) else if *head*(C) = @ and *head*(*tail*(S)) is a predefined function f
    then cfg([f(*head*(S)) | *tail*(*tail*(S))], E, *tail*(C), D)

(6) else if *head*(C) = @ and *head*(*tail*(S)) = closure(V,B,E_1)
    then cfg([ ], [V$\mapsto$ *head*(S)]E_1, [B], cfg(*tail*(*tail*(S)), E, *tail*(C), D))

(7) else if C = [ ]
    then cfg([*head*(S) | S_1], E_1, C_1, D_1) where D = cfg(S_1,E_1,C_1,D_1)

Figure 2.1: Transition Function for the SECD Machine

inference rules. Thus, the abstract machine becomes a system of inference rules. The classic representation of an inference rule has premises listed above a horizontal line, a conclusion below the line, and any required condition (if necessary), to the right. Formally, here is the general form of an inference rule:

$$\frac{premise_1 \ \ldots \ premise_n}{conclusion} \ condition.$$

Inference rules can be used to specify the abstract syntax of a language, as well as the semantics of expressions and commands. Program meaning derives from the use of inference rules on a program. Inference rules perform syntactic transformations of language elements until no further transformations are possible and normal form values remain. The formal technique of structural induction on transformations provides a powerful mechanism for proving properties about programs.

2.7 Related Work

A rich body of work exists proposing process algebraic approaches to model concurrency and distributed computation. However, they each differ from paraDOS in one or two important ways: single-event transitions and assumption of causal relationships between events. For example, CCS and the π-Calculus, by Milner [Mil89, Mil99], employ singular transitions and interleaving to express concurrency. Event structures and causal trees, by Degano, *et al.* [DDM88, DD90b, DD90a], employ graph or tree structures to represent parallel events, whose edges represent causal relationships between individual events. Two important differences here are that causal relationships preclude event structures from being considered parallel events in the sense of paraDOS, and paraDOS does not proceed from the assumption of knowledge of any causal relationships between events; it is strictly observational.

Hoare's Unifying Theories [Hoa94] are not unifying in the same sense of paraDOS as a general model; rather Hoare provides a notation and mechanism for alternatively representing a given model as a denotational, algebraic, or operational semantics. That is, using the proposed notation, a semantics in one form can be mechanically translated to either of the other two semantics.

Joint work in parallel program composition, between the California Institute of Technology and Argonne National Laboratory, by Chandy and Taylor [CT90, CT92] and Foster, *et al.* [FOT92], led to PCN (Program Composition Notation). According to the PCN approach, two types of variables exist: mutable and definitional (single assignment). Mutable variables must be local to some composable element of a program, while definitional variables can be shared across composable elements. At run time, programs are decomposed into pieces, such that single assignment variables may be assigned a value at most once, otherwise they are undefined. Attempts to access an undefined definitional variable are blocked until such time as the variable is assigned a value. This approach results in a run time environment without race conditions.

An informal operational semantics of the C-Linda programming language was developed by Narem [Nar89]. An operational semantics for Actorspaces [AC93, AC94], an extension of the actors model that supports Linda-like tuple spaces, was presented by Callsen [Cal94]. A structured operational semantics for Linda tuple space was developed by Jensen [CJY94, Jen94], as an important part of the development of a computational model for multiple tuple spaces. As one of the points addressed in building a refinement calculus for tuple spaces, Semini and Montangero [SM99] define a reference language and its operational semantics.

While the model presented in this paper is also an operational semantics for Linda and tuple space, our work is distinguished from previous work in several ways. ParaDOS directly supports multiple simultaneous views of a computation. Transition steps in previous models correspond to single event occurrences; transitions in paraDOS correspond to parallel event occurrences. In Narem [Nar89], an informal operational semantics is given for a limited implementation of `eval`(). In Jensen [CJY94, Jen94], `eval`() is treated, but at a different level of abstraction. Support for views and parallel events is a more natural level of abstraction for reasoning about parallel and distributed computation. Finally, paraDOS is a general model for reasoning about parallel and distributed computation that can and has been instantiated for computational paradigms other than tuple space, e.g. see [SPH98]. Previous operational semantics developed by the other researchers mentioned in this section were specific to Linda and tuple space. These ideas will be explored further in the following chapters.

CHAPTER 3

paraDOS Concepts

ParaDOS uses a convergence of tools and techniques for modeling different forms of concurrency, including parallel and distributed systems. It is designed to improve upon existing levels of abstraction for reasoning about properties of concurrent computation. The result is a model of computation with new and useful abstractions for describing concurrency and reasoning about properties of such systems. This chapter discusses important concepts needed to understand paraDOS's features and the motivations for their inclusion.

ParaDOS models concurrency using a parameterized operational semantics. The reasons for choosing operational semantics to develop paraDOS are twofold. First, an operational semantics describes how computation proceeds. Second, an operational semantics permits choosing an appropriate level of abstraction, including the possibility for defining a parameterized model. The motivation for including parameters is to make paraDOS a general model that can be instantiated. Each such instance can be used to study and reason about the properties of some specific parallel or distributed system within a consistent framework.

From CSP we borrow the practice of event-based reasoning and the notion of event traces to represent a computation's history. The first concept to discuss is that of events, or, more precisely, *observable events*. The events of a system are at a level of abstraction meaningful for describing and reasoning about that system's computation. Events are the primitive elements of a CSP environment. CSP events serve a dual purpose; they describe the behavior of a process, and they form an event trace when recorded in sequence by an observer. CSP represents concurrency by interleaving the respective traces of two or more concurrently executing processes. CSP is a process algebra, a system in which algebraic laws provide the mechanism for specifying permissible interleavings, and for expressing predicates to reason about properties of computation.

One of the great challenges of developing a general model concerns the identification of common observable behavior among the variety of possible systems. Interprocess communication is one such common behavior of concurrent systems, even if the specific forms of communication vary greatly. For example, in message passing systems, events could be message transmission and delivery; in shared memory systems, events could be memory reads and writes. Even among these examples, many more possibilities exist for event identification. Since paraDOS is to be a general model of concurrency, event specification is a parameter.

CSP is a model of concurrency that abstracts away event simultaneity by interleaving traces; the CSP algebra addresses issues of concurrency and nondeterminism. This event trace abstraction provides the basis for our work. ParaDOS extends the CSP notion of a trace in several important ways. First, paraDOS introduces the concept of a *parallel event*, an event aggregate, as the building block of a trace. A trace of parallel events is just a list of multisets of events. Traces of event multisets inherently convey levels of parallelism in the computational histories they represent. Another benefit of event multiset traces is the possible occurrence of one or more empty event multisets in a trace. In other words, multisets permit a natural representation of computation proceeding in

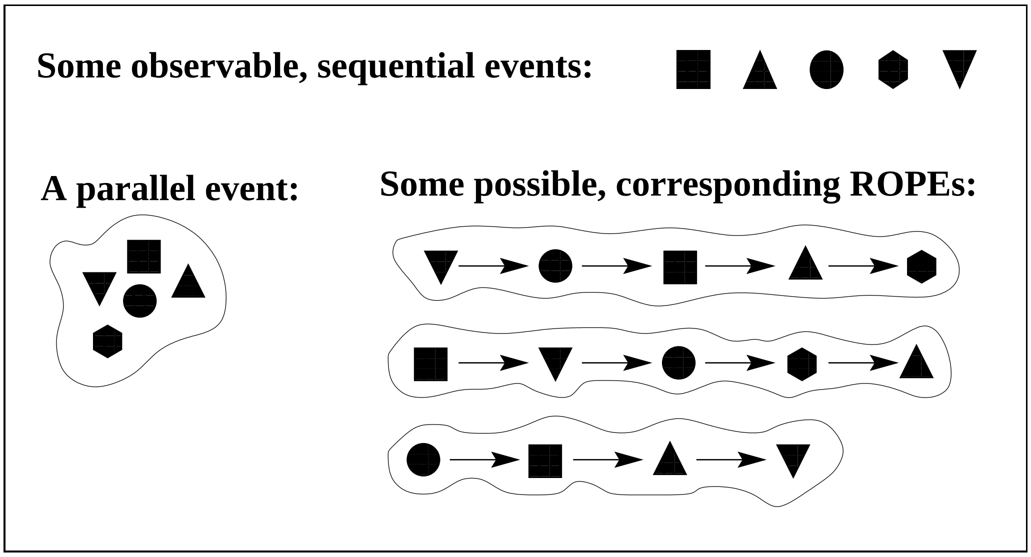

Figure 3.1: paraDOS Concepts: events, parallel event, and ROPEs.

the absence of any observable events. The empty multiset is an alternative to CSP's approach of introducing a special observable event (τ) for this purpose.

In concurrent systems, especially distributed systems, it is possible for more than one observer to exist. Furthermore, it is possible for different observers to perceive computational event sequences differently, or for some observers to miss one or more event occurrences. Reasons for imperfect observation range from network unreliability to relevance filtering in consideration of scalability. ParaDOS extends CSP's notion of a single, idealized observer with multiple, possibly imperfect observers, and the concept of *views*. A view of computation implicitly represents its corresponding observer; explicitly, a view is one observer's perspective of a computation's history, a partial ordering of observable events. Multiple observers, and their corresponding views, provide relevant information about a computation's concurrency, and the many partial orderings that are possible.

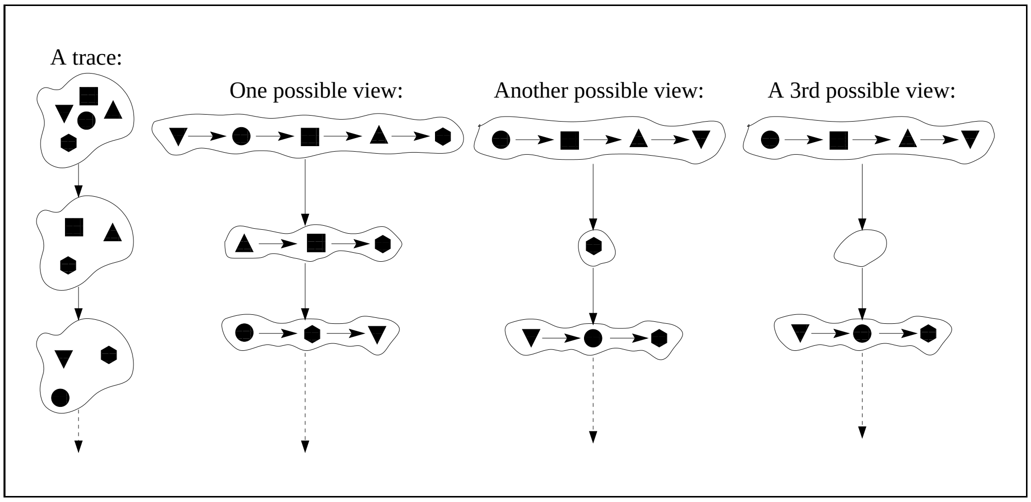

Figure 3.2: paraDOS Concepts: trace and views.

To describe views of computation in paraDOS, we introduce the concept of a ROPE, a randomly ordered parallel event, which is just a list of events from a parallel event. The concepts of observable events, parallel events, and ROPEs are depicted — using shape primitives for events — in Figure 3.1. Because paraDOS supports imperfect observation, the ROPE corresponding to a parallel event multiset need not contain all — or even any — events from that multiset. Indeed, imperfect observation implies some events may be missing from a view of computation.

Another consideration for ROPEs is the possibility of undesirable views. ParaDOS permits designating certain event sequences as not legitimate, and then constraining permissible ROPEs accordingly. Views of a computation are derived from that computation's trace, as depicted in Figure 3.2. While a trace is a list of event multisets, a corresponding view is a list of lists (ROPEs) of events. The structure of a view, like that of a parallel event, preserves concurrency information. An important parameter of paraDOS is the view relation, which permits the possibility of imperfect observation and the designation of undesirable views.

Parallel events, ROPEs, and the distinction of a computation's history from its views are abstractions that permit reasoning about computational histories that cannot, in general, be represented by sequential interleavings. To see this, assume perfect observation, and assume different instances of the same event are indistinguishable. Given these two assumptions, it is not possible to reconstruct the parallel event trace of a computation, even if one is given all possible sequential interleavings of that computation. Thus, while it is easy to generate all possible views from a parallel event trace, the reverse mapping is not. in general, possible. For example, consider the sequential interleaving $\langle A,\ A,\ A,\ A\rangle$, and assume this trace represents all possible interleavings of some system's computational history. It is not possible to determine from this trace alone whether the parallel event trace of the same computation is $\langle\{A, A, A\},\ A\rangle$ or $\langle\{A, A\},\ \{A, A\}\rangle$, or some other possible parallel event trace.

The phenomenon of views is not the only concept that derives from parallel event traces; there is also the concept of *transition density*. Consider a paraDOS trace as a labeled, directed graph, where the parallel events represent nodes, the possible sequences of parallel events in the trace define the directed edges of the graph, and the cardinality of each parallel event multiset serves as a weight with which to label the corresponding node's incoming transition. In other words, we can represent a paraDOS trace as a labeled transition system, where each label measures the number of observable events that occur during that node's corresponding transition. Thus, transition density is a measure of parallelism in each transition of a concurrent system, or, when aggregated over an entire trace, is a measure of overall concurrency. Alternatively, transition density serves as a parameter in paraDOS. A transition density of one models sequential computation; transition densities greater than one specify permissible levels of parallelism.

The concepts described to this point are the primitive elements of trace-based reasoning within paraDOS. What remains are descriptions of the concepts our operational semantics employs to generate parallel events, traces, and views of concurrent computation. To define an operational semantics requires identifying the components of a system's state, and a state transition function to describe how computation proceeds from one state to the next. In the case of an operational semantics for parallel or distributed computation, a transition relation often takes the place of a transition function due to inherent nondeterminism. When multiple independent processes can make simultaneous computational progress in a single transition, many next states are possible; modeling to which state computation proceeds in a transition reduces to a nondeterministic choice from the possible next states.

Several general abstractions emerge concerning the components of a system's state in paraDOS. The first abstraction is to represent processes as continuations. A continuation represents the remainder of a process's computation. The second abstraction is to represent communications as closures. A closure is the binding of an expression and the environment in which it is to be evaluated. The third abstraction is to represent observable behavior from the preceding transition in a parallel event set, discussed earlier in this chapter. The final abstraction concerning components of a paraDOS state is the next (possibly unevaluated) state to which computation proceeds. Thus, the definition of *state* in paraDOS is recursive (and, as the next paragraph explains, lazy). The specifics of processes and communications may differ from one instance of paraDOS to another, but the above abstractions concerning a system's components frame the paraDOS state parameter.

Lazy evaluation — delaying evaluation until the last possible moment — is an important concept needed to understand the specification of a paraDOS transition relation. Lazy evaluation emerges in paraDOS as an effective approach to managing the inherent nondeterminism present in models of concurrency. The computation space of a program modeled by paraDOS is a lazy tree, as depicted in Figure 3.3. Nodes in the tree represent system configurations, or states; branches represent state transitions. A program's initial configuration corresponds to the root node of the tree. Branches drawn with solid lines represent the path of computation, or the tree's traversal. Nodes drawn with solid circles represent the elaborated configurations within the computation space. Dashed lines and circles in the tree represent unselected transitions and unelaborated states, respectively. The transition relation only elaborates the states to which computation proceeds (i.e., lazy evaluation). Without lazy evaluation, the size of our tree (computation space) would distract us from comprehending a system's computation, and attempts to implement an instance of paraDOS without lazy evaluation would be time and space prohibitive, or even impossible in the case of infinite computation spaces.

Each invocation of the transition relation elaborates one additional state within the paraDOS computation space. The result is a traversal down one more level of the lazy tree, from the current system configuration to the next configuration. The abstraction for selecting which state to elaborate amounts to pruning away possible next states, according to policies specified by the transition relation, until only one selection remains. The pruning occurs in stages; each stage corresponds to some amount of computational progress. Two examples of stages of computational progress are the selection of a set of eligible processes and a set of communication closures, where at each stage, all possible sets not chosen represent pruned subtrees of the computation space. Two additional stages in-

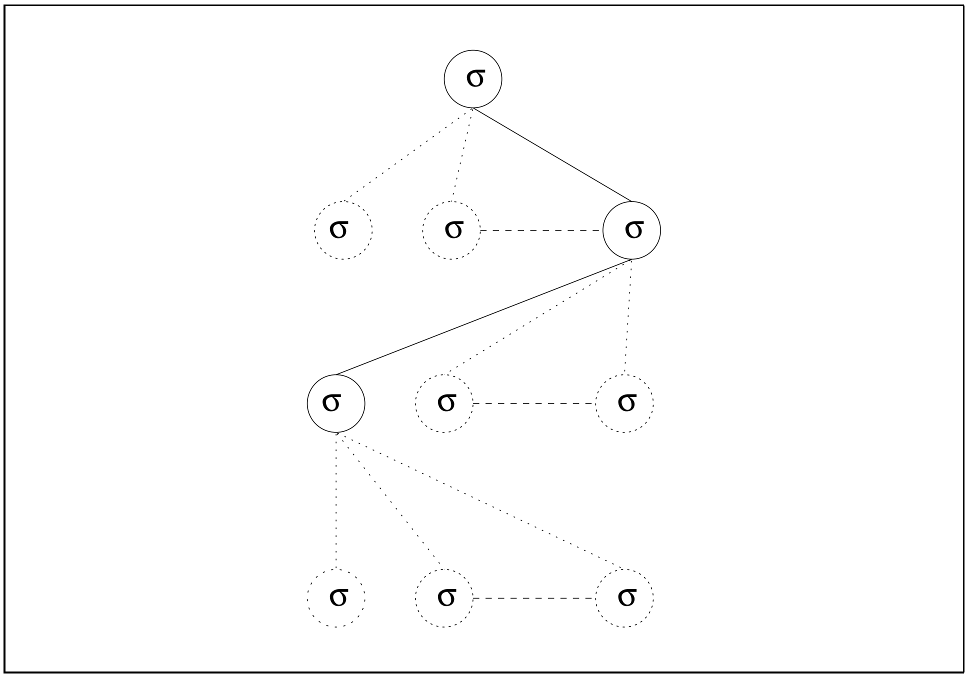

Figure 3.3: ParaDOS computation space: a lazy tree.

volve selecting a sequence to reduce communication closures, and a sequence to evaluate process continuations. Once again, sequences not chosen in each of these two steps represent further pruning of subtrees. The transition relation assumes the existence of a meaning function to abstract away details of the internal computation of process continuations. As well, during the stages of the transition relation, it is possible to generate one or more observable events. The generated events, new or updated process continuations, and new or reduced communication closures contribute to the configuration of the newly elaborated state. Since the number of stages and the semantics of each stage may differ from one instance of paraDOS to another, the specification of the transition relation is a parameter.

One additional paraDOS parameter transcends the previous concepts and parameters discussed in this chapter. This parameter is composition. Implicitly, this chapter presents paraDOS as a framework to model a single concurrent system, whose configuration includes multiple processes, communications, and other infrastructure we use to support reasoning about computational properties. However, especially from a distributed system standpoint, a concurrent system is also the result of composing two or more (possibly concurrent) systems.

For example, consider businesses who have an Internet presence, and wish to integrate their respective systems to take advantage of the benefits of electronic commerce. The result of such integrations (ideally) is concurrent, multiway transactions between the respective business systems. It is more natural to model such systems as the composition of individual business systems than as a single concurrent system.

Since the desire exists to model the composition of concurrent systems, one of paraDOS's parameters is a composition grammar. The degenerate specification of this parameter is a single concurrent system. In general, the composition grammar is a rewriting system capable of generating composition graphs. In these graphs, a node represents a system and an edge connecting two nodes represents the composition of their corresponding systems. Each system has its own computation space, communication closures, and observers. One possible composition grammar — presented in Chapter 8 — generates string representations of a composition tree, where each node is a system, and a parent node represents the composition of its children. Other composition grammars are possible.

CHAPTER 4

paraDOS Uninstantiated

This chapter presents the uninstantiated paraDOS model. First, we introduce helpful notation to understand the subsequent definitions and discussion. Next, we formalize the concepts presented previously in Chapter 3, and lay the foundation for further formal discussion in this dissertation's remaining chapters.

The model presented in this section is denoted $\overline{\mathcal{S}}$, and the components for $\overline{\mathcal{S}}$ are described below. The bar notation is used to denote elements in the model $\overline{\mathcal{S}}$ which correspond to elements in system $\mathcal{S}$.

Formally, $\overline{\mathcal{S}}$ is represented by the 3-tuple $\langle\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\rangle$, where σ represents the computation space of $\overline{\mathcal{S}}$, $\overline{\Lambda}$ represents the set of communication closures within σ, and $\overline{\Upsilon}$ represents the set of views of the computation within σ. The remainder of this section discusses in greater detail the concepts embedded within $\overline{\mathcal{S}}$. In

Table 4.1: paraDOS Notation

Notation	Meaning
$\mathcal{S}$	A concurrent system
$\overline{\mathcal{S}}$	Model of $\mathcal{S}$
σ, σ_i	Computation space (lazy tree) of $\overline{\mathcal{S}}$, or a decorated state within tree σ
$\overline{\Lambda}$	Set of communication closures
λ	A communication closure
$\overline{\Upsilon}$	Set of views
υ	A view
ρ	A ROPE

turn, we cover computation spaces, communication closures, observable events, traces, and views.

The state σ is a lazy tree of state nodes. When we refer to the tree σ, we refer to $\overline{\mathcal{S}}$'s computation space. Each node in the tree represents a potential computational state. Branches in the tree represent state transitions. The root node σ is $\overline{\mathcal{S}}$'s start state, which corresponds to a program's initial configuration in the system being modeled by $\overline{\mathcal{S}}$. State nodes carry additional information to support the operational semantics. The specific elements of σ vary from instance to instance of paraDOS.

Each level of tree σ represents a computational step. Computation proceeds from one state to the next in σ through $\overline{\mathcal{S}}$'s transition function. Given a current state, the transition function randomly chooses a next state from among all possible next states. At each transition, the chosen next state in σ is evaluated, and thus computation proceeds. The logic of the transition function may vary, but must reflect the computational capabilities of the system being modeled by $\overline{\mathcal{S}}$.

Two special conditions exist in which the transition function fails to choose a next state in σ: computational quiescence and computation ends. Computational quiescence implies a temporary condition under which computation cannot proceed; computation ends implies the condition that computation will never proceed. Both conditions indicate that, for a given invocation, the transition function has no possible next states. The manner of detecting, or even the ability to detect, these two special conditions, may vary.

To model the variety of approaches to parallel and distributed computation, paraDOS needs to parameterize communication. The set of communication closures $\overline{\Lambda}$ is the realization of this parameter, where the elements of $\overline{\Lambda}$, the individual closure forms, λ, vary from instance to instance of paraDOS.

These concepts are illustrated in Figures 3.1 and 3.2, and we formally define them next. We define an observable event formally as follows:

Definition 2 (*observable event*) An observable event is an instance of input/output (including message passing) behavior.

In our research, we further distinguish sequential events from parallel events, and define them formally as follows:

Definition 3 (*sequential event*) A sequential event is the occurrence of an individual, observable event.

Definition 4 (*parallel event*) A parallel event is the simultaneous occurrence of multiple sequential events, represented as a set of sequential events.

The traversal of computation space σ represents the actual history of computation of a program within $\overline{\mathcal{S}}$. We borrow the notion of a *trace* from Hoare's CSP [Hoa85], with one significant refinement for distributed systems: it *is* possible for two or more observable events to occur simultaneously. We define sequential and parallel event traces as follows:

Definition 5 (*sequential event trace*) A sequential event trace is an ordered list of sequential events representing the sequential system's computational history.

Definition 6 (*parallel event trace*) A parallel event trace is an ordered list of parallel events representing the parallel system's computational history.

For the remainder of this paper, unless otherwise stated, a trace refers to a parallel event trace in paraDOS. In the case of paraDOS, a parallel event trace is a trace of $\overline{\mathcal{S}}$, constructed from the traversal of σ, and is a representation of $\overline{\mathcal{S}}$'s computational history.

One additional concept proves to be useful for the definition of views. We introduce the notion of a randomly ordered parallel event, or ROPE, as a linearization of events in a parallel event, and define ROPE formally as follows:

Definition 7 (*ROPE*) A randomly ordered parallel event, or ROPE, is a randomly ordered list of sequential events which together comprise a subset of a parallel event.

ParaDOS explicitly represents the multiple, potentially distinct, views of computation within $\overline{\mathcal{S}}$. The notion of a view in paraDOS is separate from the notion of a trace. A view of sequential computation is equivalent to a sequential event trace, and is therefore not distinguished. We define the notion of a view of parallel computation formally as follows:

Definition 8 (*view*) A view, v, of a parallel event trace, tr, is a list of ROPEs where each ROPE, ρ, in v is derived from ρ's corresponding parallel event in a tr.

Thus, views of distributed computation are represented at the sequential event level, with the barriers of ROPEs, in paraDOS; while traces are at the parallel event level.

There are several implications of the definition of ROPE, related to the concept of views, that need to be discussed. First, a subset of a parallel event can be empty, a non-empty proper subset of the parallel event, or the entire set of sequential events that represent the parallel event. The notion of subset represents the possibility that one or more sequential events within a parallel event may not be observed. Explanations for this phenomenon range from imperfect observers to unreliability in the transport layer of the network. Imperfect observers in this context are not necessarily the result of negligence, and are sometimes intentional. Relevance filtering, a necessity for scalability in many distributed applications, is one example of imperfect observation.

The second implication of the definition of ROPE concerns the random ordering of sequential events. A ROPE can be considered to be a sequentialized instance of a parallel event. That is, if an observer witnesses the occurrence of a parallel event, and is asked to record what he saw, the result would be a list in some random order: one sequentialized instance of a parallel event. Additional observers may record the same parallel event differently, and thus ROPEs represent the many possible sequentialized instances of a parallel event.

Element $\overline{\Upsilon}$ of $\overline{\mathcal{S}}$ is a set of views. Each υ in $\overline{\Upsilon}$ is a list of ROPEs that represents a possible view of computation. Let υ_i be a particular view of computation in $\overline{\Upsilon}$. The j^{th} element of υ_i, denoted ρ_j, is a list of sequential events whose order represents observer υ_i's own view of computation. Element ρ_j of υ_i corresponds to the j^{th} element of $\overline{\mathcal{S}}$'s trace, or the j^{th} parallel event. Any ordering of any subset of the j^{th} parallel event of $\overline{\mathcal{S}}$'s trace constitutes a ROPE, or valid view, of the j^{th} parallel event.

We express the view relation with two functions as shown in Figure 4.1. Instances of the view relation differ only by the definitions of their respective states σ. The view relation $\mathcal{F}_\upsilon$ traverses its input view υ and tree σ, until an unelaborated ROPE is encountered in υ. Next, $\mathcal{F}_\upsilon$ calls relation V to continue traversing σ, for some random number of transitions limited so as not to overtake the current state of computation. While V continues to traverse σ, it also constructs a

$$
\begin{array}{l}
\mathcal{F}_\upsilon : \textit{view} \times \textit{state} \longrightarrow \textit{view} \\
\quad \mathcal{F}_\upsilon(\upsilon, \sigma) = \\
\qquad \text{if } \upsilon \text{ empty} \\
\qquad\quad V(\sigma) \\
\qquad \text{else} \\
\qquad\quad \textit{append}((\textit{head}(\upsilon)), \mathcal{F}_\upsilon(\textit{tail}(\upsilon), \textit{nextstate}(\sigma)))
\end{array}
$$

$$
\begin{array}{l}
V : \textit{state} \longrightarrow \textit{view} \\
\quad V(\sigma) = \\
\qquad \text{if } \sigma \textit{ undefined} \\
\qquad\quad () \\
\qquad \text{else} \\
\qquad\quad \text{let } \textit{viewSet} \subseteq \textit{get}\overline{\mathcal{P}}(\sigma) \\
\qquad\quad \text{in let } \rho = \textit{list}(\textit{viewSet}) \\
\qquad\quad \text{in random choice of} \\
\qquad\qquad \begin{cases} \textit{append}((\rho), V(\textit{nextstate}(\sigma)), & \text{or} \\ (\rho) \end{cases}
\end{array}
$$

Figure 4.1: paraDOS View Functions

subsequent view υ' to return to $\mathcal{F}_\upsilon$. For each state traversed, the corresponding ρ_i in υ' is a random linearization of a random subset of $\overline{\mathcal{P}}$. Upon return, $\mathcal{F}_\upsilon$ appends υ' to the end of υ, thus constructing the new view.

Finally, one useful way to characterize the computation space and transition function of $\overline{\mathcal{S}}$ is as a labeled transition system (LTS). An LTS is a labeled, directed graph. We can map the trace of $\overline{\mathcal{S}}$ to an LTS as follows: each state in the trace maps to a node; each transition between states maps to a directed edge between the corresponding nodes; and each label on a state transition denotes a weight. The weight of each edge represents its *transition density*, which we define as:

Definition 9 (*transition density*) Let M represent an LTS, and t represent a transition within M. The transition density of t is the number of observable events that occur when t is chosen within M.

Transition density is an attribute of LTS-based models of computation. For different instances of ParaDOS, transition density may vary. Transition density exists both as a parameter and an attribute, as a specification for and a measure of parallelism. ParaDOS doesn't require the services of an idealized observer to produce a trace precisely because our model supports parallel events, and thus a transition density greater than one.

CHAPTER 5

paraDOS Instantiated for Actors

Section 5.1 presents P^{act}, paraDOS instantiated for the Actors model of computation. Section 5.2 presents AT, Mason and Talcott's Actor Theories [MT97]. Section 5.3 states and proves a theorem concerning the equivalence of a restricted version of the P^{act} semantics and the semantics of AT.

5.1 The P^{act} Instance of ParaDOS

Section 5.1.1 defines the computational elements of the Actors model, and the state of an actor system. Section 5.1.2 gives the domain specification for $\overline{\mathcal{S}}$, and defines P^{act}'s transition and view relations. Section 5.1.3 discusses the functions that help specify P^{act}'s operational semantics. We present the equivalence theorem and proof in Section 5.3. Section 5.1.4 discusses some decidable predicates within P^{act}, and some that are not decidable.

5.1.1 An Actor System

Section 2.2 presented background information regarding the Actors model. In addition, we define the following computational elements of an actor system:

Definition 10 (*actor*) An actor is a computational agent that has a behavior and is uniquely identified by its mail queue address.

Definition 11 (*actor machine*) An actor machine is an instance of an actor and its current behavior, bound to a particular element (address) of that actor's mail queue.

Definition 12 (*task*) A task is the content of a message sent to a designated recipient (an actor) that is uniquely identified by its task id.

Given the definitions of actor, actor machine, and task, we define the state of $\mathcal{S}$ at an instant in time t to be composed of the contents of two sets, active actors and active tasks. The set of active actors, $\mathcal{A}$, contains actor machines still performing computation within $\mathcal{S}$. The set of active tasks, $\mathcal{T}$, consists of undelivered messages within $\mathcal{S}$. Both $\mathcal{A}$ and $\mathcal{T}$ from $\mathcal{S}$ have counterparts $\overline{\mathcal{A}}$ and $\overline{\mathcal{T}}$ in $\overline{\mathcal{S}}$, the equivalent paraDOS system.

5.1.2 The P^{act} Specification

The instance of paraDOS for Actors, P^{act}, is an operational semantics for reasoning about properties of computation in an Actor system, $\mathcal{S}$. To instantiate P^{act}, we must define $\overline{\mathcal{S}} = \langle \sigma, \overline{\Lambda}, \overline{\Upsilon} \rangle$, and P^{act}'s transition and view relations. Table 5.1 contains the domain specification for P^{act}.

Var	Domain	Domain Specification
$\overline{\mathcal{S}}$	*system*	$state \times closureSet \times viewSet$
σ	*state*	$mailqSet \times actmachSet \times taskSet \times$ $parEventSet \times state$ $\mid$ undefined
$\overline{\mathcal{M}}$	*mailqSet*	$P^{(mailq)}$
$\overline{\mathcal{A}}$	*actmachSet*	$P^{(actmach)}$
$\overline{\mathcal{T}}$	*taskSet*	$P^{(task)}$
$\overline{\mathcal{P}}$	*parEventSet*	$P^{(seqEvent)}$
m_i	*mailq*	$list(task)$
$\overline{\alpha}$	*actmach*	$mqloc \times beh \times beh$
$\overline{\tau}$	*task*	$tidtype \times mailq \times msg$
ε	*seqEvent*	$etype \times task$
$m_i[loc]$	*mqloc*	$mailq \times int$
ψ	*beh*	continuation (unspecified)
tid	*tidtype*	task identifier (unspecified)
κ	*msg*	message content (unspecified)
$\mathbf{E}_{type}$	*etype*	$\{\mathbf{E_S}, \mathbf{E_D}\}$
μ	*meaning*	$actmach \times mailqSet \times actmachSet \times$ $taskSet \times actmachSet$
$\overline{\Lambda}$	*closureSet*	$\emptyset$
$\overline{\Upsilon}$	*viewSet*	$P^{(view)}$
υ	*view*	$list(ROPE)$
ρ	*ROPE*	$list(seqEvent)$

Figure 5.1: P^{act} Domain Specification

The remainder of this section discusses the P^{act} domain specification in greater detail, and the P^{act} semantics. Section 5.1.3 contains the helper functions and predicates over the P^{act} domain. These helper functions and predicates support both the P^{act} semantics in this section and the equivalence theorem and proof in Section 5.3.

Before proceeding, we need to comment on the three P^{act} types left unspecified in Table 5.1. Type *msg* represents the data domain of messages and actors, and the semantics is parameterized with respect to this domain. The type for the continuation of an actor, *beh*, is specfic to the particular actor meaning function and is thus unspecified in P^{act}. The only requirement for type *tidtype* is that it be possible to generate unique elements for all tasks within the model. Closure set $\overline{\Lambda}$ remains empty for P^{act}, since the paraDOS abstraction of communication closures did not emerge until after we had defined P^{act}.

The state of $\overline{\mathcal{S}}$, denoted σ, consists of all possible states reachable from the start state of $\mathcal{S}$; for this reason, σ is also the computation space of $\overline{\mathcal{S}}$. Each state at some time t in $\mathcal{S}$ corresponds to at least one state in σ. The root node of σ corresponds to the start state in $\mathcal{S}$. The one-to-many relationship between states in $\mathcal{S}$ and states in σ reflects only the multiple computational paths possible, not additional or different computational power. These multiple paths represent the nondeterminism possible during parallel and distributed computation.

Since $\overline{\mathcal{S}}$ is a model, the states in σ carry additional information to facilitate P^{act}'s operational semantics. A state, σ, is represented by the 5-tuple $\langle\overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next}\rangle$, where $\overline{\mathcal{M}}$ is the set of mail queues for the actors, $\overline{\mathcal{A}}$ is the set of actor machines, $\overline{\mathcal{T}}$ is the set of tasks, $\overline{\mathcal{P}}$ is the set of parallel events, and σ_{next} is either undefined, or the state to which computation next proceeds, as assigned by the transition relation.

Sets $\mathcal{A}$ and $\mathcal{T}$ in $\mathcal{S}$ consist of active actors and tasks, respectively. Their counterparts in $\overline{\mathcal{S}}$, $\overline{\mathcal{A}}$ and $\overline{\mathcal{T}}$, have been introduced as two of the parts comprising the 5-tuple that represents a state, σ, in $\overline{\mathcal{S}}$. $\overline{\mathcal{M}}$ contains the mail queues of the actors whose actor machines are in $\overline{\mathcal{A}}$. In particular, the i^{th} actor's mail queue in $\overline{\mathcal{M}}$, denoted m_i, contains the tasks delivered to actor m_i.

Formally, actor machine $\overline{\alpha}$, an element of $\overline{\mathcal{A}}$, is represented by the 3-tuple $\langle m_i[loc], \psi_{init}, \psi_{cont}\rangle$, where $m_i[loc]$ is the element loc in mail queue m_i within $\overline{\mathcal{M}}$ to which $\overline{\alpha}$ is bound, ψ_{init} is the initial behavior of $\overline{\alpha}$, and ψ_{cont} is the current

$$
\begin{aligned}
&\textit{inboundtasks} : \textit{taskSet} \times \textit{mailqSet} \longrightarrow \textit{taskSet}\\
&\quad \textit{inboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}}) =\\
&\qquad \{\overline{\tau} \mid \overline{\tau} \in \overline{\mathcal{T}} \bigwedge \exists m_i \in \overline{\mathcal{M}} \;\text{ s.t. } \textit{recip}(\overline{\tau}) = m_i\}
\end{aligned}
$$

$$
\begin{aligned}
&\textit{outboundtasks} : \textit{taskSet} \times \textit{mailqSet} \longrightarrow \textit{taskSet}\\
&\quad \textit{outboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}}) =\\
&\qquad \overline{\mathcal{T}} - \textit{inboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}})
\end{aligned}
$$

Figure 5.2: P^{act} Inbound and Outbound Tasks in $\overline{\mathcal{T}}$

continuation behavior of $\overline{\alpha}$. No two actor machines in $\overline{\mathcal{A}}$ are bound to the same mail queue element of any mail queue in $\overline{\mathcal{M}}$.

Formally, model task $\overline{\tau}$, an element of $\overline{\mathcal{T}}$ in σ, is represented by the 3-tuple $\langle tid, m_i, \kappa \rangle$, where tid is the unique task identifier, m_i is the task recipient (an actor's mail queue within $\overline{\mathcal{M}}$), and κ is the communication (message content). Set $\overline{\mathcal{T}}$ can be divided into two subsets: the set of inbound tasks, $\textit{inboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}})$, and the set of outbound tasks, $\textit{outboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}})$. An inbound task is a task $\overline{\tau}$ whose recipient is an actor whose mail queue m_i is in $\overline{\mathcal{M}}$; otherwise $\overline{\tau}$ is an outbound task. Functions *inboundtasks*() and *outboundtasks*() are given in Figure 5.2.

Now that the representations of actor machines and tasks comprising $\overline{\mathcal{A}}$ and $\overline{\mathcal{T}}$ within a state, σ, have been defined, discussion returns to the events in $\overline{\mathcal{P}}$. The formal definition of an observable event is as follows:

Definition 13 (*observable event*) An observable event is a task that has been sent by, or delivered to, an actor.

An event, ε, is represented by the pair $\langle\ \mathbf{E}_{type}, \overline{\tau}\ \rangle$, where values for $\mathbf{E}_{type}$ are either $\mathbf{E_S}$ for a sent task or $\mathbf{E_D}$ for a delivered task. Thus, $\overline{\mathcal{P}}$ in σ is a set of instances of the two types of sequential events.

There is an important derivative relationship between these two elements. In a sense, $\overline{\mathcal{P}}$ is a special derivative of $\overline{\mathcal{T}}$; it represents the changes in $\overline{\mathcal{T}}$ with respect to time. Time in P^{act} is measured discretely by the transitions from one state to the next. Specifically, events in $\overline{\mathcal{P}}$ of type $\mathbf{E_S}$ represent those $\overline{\tau}$'s added to $\overline{\mathcal{T}}$ in the last transition; events in $\overline{\mathcal{P}}$ of type $\mathbf{E_D}$ represent those $\overline{\tau}$'s removed from $inboundtasks(\overline{\mathcal{T}}, \overline{\mathcal{M}})$ in the last transition. Since $\overline{\mathcal{P}}$ is derivable from $\overline{\mathcal{T}}$, it is not necessary to represent $\overline{\mathcal{P}}$ explicitly in σ. However, it is useful to maintain $\overline{\mathcal{P}}$ within each σ to facilitate the construction of views.

The meaning structure, μ, is not part of $\overline{\mathcal{S}}$, but it supports the P^{act} meaning relation shown in Figure 5.3. The meaning relation returns one meaning μ from the set of possible meanings for an actor machine's computation. The formal definition of meaning is as follows:

Definition 14 (*meaning*) The meaning of actor machine $\overline{\alpha}$'s computation from time t to time $t+1$ is information consisting of $\overline{\alpha}$'s remaining computation, the (possibly empty) set of new actors created, the (possibly empty) set of new tasks created, and $\overline{\alpha}$'s (possibly unspecified) replacement behavior.

In P^{act}, we define a meaning structure μ to represent one of the many possible meanings of $\overline{\alpha}$'s computation, and represent μ by the 5-tuple $\langle\, \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \,\rangle$. Element $\overline{\alpha}_{cont}$ represents $\overline{\alpha}$ updated with its new continuation. The sets $\overline{\mathcal{M}}_{new}$ $\overline{\mathcal{A}}_{new}$, and $\overline{\mathcal{T}}_{new}$ are the sets of new mail queues, new actor machines, and new tasks created by $\overline{\alpha}$'s computation. Element $\overline{\mathcal{A}}_{repl}$ is either the empty set, or a singleton set containing $\overline{\alpha}$'s replacement actor machine.

We assume the existence of an actor meaning function Am that abstracts away the details of actor machine execution. This is a reasonable assumption since such semantics are already specified in Agha [Agh86] and more recently in Agha et

$$
\begin{array}{l}
\mathcal{F}_\mu : \mathit{actmach} \longrightarrow \mathit{meaning} \\
\quad \mathcal{F}_\mu(\langle m_i[loc], \psi_{init}, \psi_{cont}\rangle) = \\
\qquad \mu \text{ where} \\
\qquad\quad \mu \in \{\langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl}\rangle \mid \\
\qquad\qquad\quad Am(\psi_{cont}) \text{ yields } \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \ \}
\end{array}
$$

Figure 5.3: P^{act} Meaning Function

al. [AMS97]. Function Am takes as its input argument an actor continuation, ψ_{cont}. Function Am returns a finite set of meanings for a given actor machine. The set of meanings returned by Am represents all the possible continuations of ψ_{cont}. This set must be finite since the language of an individual actor machine, as specified by Agha [Agh86], does not permit infinite execution. From Am, we construct the P^{act} meaning relation $\mathcal{F}_\mu$ shown in Figure 5.3. Relation $\mathcal{F}_\mu$ maps an actor machine $\overline{\alpha}$ to a meaning of $\overline{\alpha}$'s current behavior. Relation $\mathcal{F}_\mu$ randomly chooses one meaning for $\overline{\alpha}$'s computation from the set of all possible meanings of $\overline{\alpha}$'s current continuation, ψ_{cont}.

In P^{act}, computation proceeds by calling the transition relation, $\mathcal{F}_\delta$. The transition relation, shown in Figure 5.4, is itself composed of three functions, the inbound tasks function $\mathcal{F}_{in}$, the outbound tasks function $\mathcal{F}_{out}$, and the generate children function $\mathcal{G}$. Relation $\mathcal{F}_\delta$ traverses σ until it finds a state whose σ_{next} is *undefined*. Such a state is the current state of $\overline{\mathcal{S}}$, denoted σ_{cur}. Relation $\mathcal{F}_\delta$ assigns to $\sigma_{cur}.\sigma_{next}$ the result of applying the composition of $\mathcal{F}_{out}$, $\mathcal{F}_{in}$, and $\mathcal{G}$ to σ_{cur}. Figure 5.5 contains $\mathcal{F}_{out}$ and $\mathcal{F}_{in}$; Figure 5.6 contains $\mathcal{G}$. We consider the cumulative effect of applying each of these functions (relations) to σ_{cur}, in turn, elaborating the next computational state in σ.

The innermost function application $\mathcal{F}_{out}(\sigma_{cur})$ returns a new state $\sigma_{cur}{}'$, in which a random subset of outbound tasks is removed from $\overline{\mathcal{T}}$, $\overline{\mathcal{P}}$ is empty, and the remaining elements are unchanged from σ_{cur}. In the case where no outbound

$$
\begin{array}{l}
\mathcal{F}_\delta : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_\delta(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \\
\qquad \text{if } \sigma_{next}\ \mathit{undefined} \\
\qquad\quad \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \mathcal{G}(\mathcal{F}_{in}(\mathcal{F}_{out}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle))) \rangle \\
\qquad \text{else} \\
\qquad\quad \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \mathcal{F}_\delta(\sigma_{next}) \rangle
\end{array}
$$

Figure 5.4: P^{act} Transition Relation

$$
\begin{array}{l}
\mathcal{F}_{out} : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_{out}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \\
\qquad \text{let } \overline{\mathcal{T}}_{sub} \subseteq \mathit{outboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}}) \\
\qquad \text{in } \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}} - \overline{\mathcal{T}}_{sub},\ \emptyset,\ \mathit{undefined} \rangle \\
\\
\mathcal{F}_{in} : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_{in}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \\
\qquad \text{let } \mathbf{E}_r \subseteq \{ \tau \mid \tau \ \text{ is from the environment } \} \\
\qquad \text{in } \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}} \bigcup \mathbf{E}_r,\ \emptyset,\ \mathit{undefined} \rangle
\end{array}
$$

Figure 5.5: P^{act} Outgoing and Incoming External Tasks

tasks exist in σ_{cur}, $\overline{\mathcal{T}}$remains unchanged in σ_{cur}'. Set $\overline{\mathcal{P}}$ in σ_{cur}' is empty because forwarding outbound tasks beyond $\overline{\mathcal{S}}$ does not constitute an observable event; these tasks were previously observed as sent, and have yet to be delivered.

The middle function application $\mathcal{F}_{in}(\sigma_{cur}')$ returns a new state σ_{cur}'', in which a random subset of inbound tasks is chosen from the environment and added to $\overline{\mathcal{T}}$, $\overline{\mathcal{P}}$ is empty, and the remaining elements are unchanged from σ_{cur}'. In the case where no inbound tasks exist from the environment, state σ_{cur}'' remains unchanged from σ_{cur}'. Notice that $\overline{\mathcal{P}}$ remains empty in σ_{cur}'' because tasks added to $\overline{\mathcal{T}}$ from the environment do not constitute observable events; these tasks were already sent from some other location in the environment, so it is too late to observe such tasks as sent within $\overline{\mathcal{S}}$.

The outermost function application $\mathcal{G}(\sigma_{cur}'')$ returns the elaborated σ_{next} of σ_{cur}'', which represents the random choice of a next state, from among all possible next states in the σ. Function $\mathcal{G}$ constructs its return state based on a random selection of inbound tasks $\overline{\mathcal{T}}_r$ to deliver, the delivery of those tasks $\overline{\mathcal{M}}_{del(\overline{\mathcal{T}}_r)}$, and the subsequent random selection of eligible actor machines $\overline{\mathcal{A}}_r$ to make computational progress. The set $\overline{\mathcal{M}}_{del(\overline{\mathcal{T}}_r)}$ is constructed by removing those mail queues from $\overline{\mathcal{M}}$ to which tasks will be delivered, then adding back those mail queues with their delivered tasks. An eligible actor machine is an $\overline{\alpha}$ whose message state is delivered or consumed, and whose current continuation represents unfinished computation. The set $\mathbf{m}$ of randomly chosen meanings is obtained from applying $\mathcal{F}_\mu$ to each $\overline{\alpha}$ in $\overline{\mathcal{A}}_r$. The specification of $\overline{\mathcal{M}}'$, $\overline{\mathcal{A}}'$, and $\overline{\mathcal{T}}'$ is tedious but straightforward and follows this paragraph's discussion. The specification of $\overline{\mathcal{P}}'$ warrants further attention. Set $\overline{\mathcal{P}}$ will be empty regardless of any inbound or outbound task activity in $\overline{\mathcal{T}}$ that results from the two innermost function applications $\mathcal{F}_{out}$ and $\mathcal{F}_{in}$. Set $\overline{\mathcal{P}}'$ includes events of type $\mathbf{E_S}$ and $\mathbf{E_D}$ that derive from the meanings of actors in $\overline{\mathcal{A}}_r$ that created new tasks $\overline{\tau}$ or had tasks from $\overline{\mathcal{T}}_r$ delivered to them.

5.1.3 P^{act} Predicates and Helper Functions

Figure 5.7 contains accessor functions for actors and tasks in P^{act}. Functions *actname* and *cont* return an actor machine's name (mail queue identifier) and continuation (behavior), respectively. Functions *content* and *recip* return a task's message content and recipient actor's name, respectively. Function *deliveredtask* returns the task delivered to the specified actor machine.

$$
\begin{array}{l}
\mathcal{G} : state \longrightarrow state \\
\quad \mathcal{G}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \langle \overline{\mathcal{M}}',\ \overline{\mathcal{A}}',\ \overline{\mathcal{T}}',\ \overline{\mathcal{P}}',\ {\sigma_{next}}' \rangle \text{ where} \\
\quad\quad {\sigma_{next}}' \text{ is undefined, and} \\
\quad\quad \text{let } \overline{\mathcal{T}}_r \subseteq inboundtasks(\overline{\mathcal{T}}, \overline{\mathcal{M}}) \\
\quad\quad \text{in let } \overline{\mathcal{M}}_{del(\overline{\mathcal{T}}_r)} = (\overline{\mathcal{M}} - \{mailqnamed(recip(\overline{\tau}), \overline{\mathcal{M}}) \mid \overline{\tau} \in \overline{\mathcal{T}}_r\}) \bigcup \\
\quad\quad\quad\quad\quad\quad \{deliver(\overline{\tau}, mailqnamed(recip(\overline{\tau}), \overline{\mathcal{M}})) \mid \overline{\tau} \in \overline{\mathcal{T}}_r\} \\
\quad\quad \text{in let } \overline{\mathcal{A}}_r \subseteq elig_{acts}(\overline{\mathcal{A}}, \overline{\mathcal{M}}_{del(\overline{\mathcal{T}}_r)}) \\
\quad\quad \text{in let } \mathbf{m} = \{\mathcal{F}_\mu(\overline{\alpha}) \mid \overline{\alpha} \in \overline{\mathcal{A}}_r\} \\
\quad\quad \text{in} \\
\quad\quad\quad \overline{\mathcal{M}}' = \overline{\mathcal{M}}_{del(\overline{\mathcal{T}}_r)} \bigcup \left(\bigcup_{\langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle \in \mathbf{m}} \overline{\mathcal{M}}_{new} \right) \\
\quad\quad\quad \overline{\mathcal{A}}' = (\overline{\mathcal{A}} - \overline{\mathcal{A}}_r) \bigcup \\
\quad\quad\quad\quad\quad \left(\bigcup_{\langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle \in \mathbf{m}} (\{\overline{\alpha}_{cont}\} \bigcup \overline{\mathcal{A}}_{new} \bigcup \overline{\mathcal{A}}_{repl}) \right) \\
\quad\quad\quad \overline{\mathcal{T}}' = (\overline{\mathcal{T}} - \overline{\mathcal{T}}_r) \bigcup \left(\bigcup_{\langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle \in \mathbf{m}} \overline{\mathcal{T}}_{new} \right) \\
\quad\quad\quad \overline{\mathcal{P}}' = \overline{\mathcal{P}} \bigcup \\
\quad\quad\quad\quad\quad \left\{ \langle \mathbf{E_S}, \overline{\tau} \rangle \mid \overline{\tau} \in \left(\bigcup_{\langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle \in \mathbf{m}} \overline{\mathcal{T}}_{new} \right) \right\} \bigcup \\
\quad\quad\quad\quad\quad \{\langle \mathbf{E_D}, \overline{\tau} \rangle \mid \overline{\tau} \in \overline{\mathcal{T}}_r\}
\end{array}
$$

Figure 5.6: P^{act} Generate Children

Figure 5.8 contains modifier functions for actors and tasks in P^{act}. Function *newcont* updates an actor machine's current behavior with the specified continuation. Function *deliver* "delivers" a task to the specified actor's mail queue. For P^{act}, an actor mail queue is a list of tasks, and new tasks are always appended to the end of the list, but this need not be the case, in general. For example, a priority mail queue may have a different delivery strategy, specified by a paraDOS parameter, based on some system policy we wish to model. Function *retrieveMsg* binds an actor machine's behavior to the message contents of the task delivered to the actor machine's mail queue location.

Function $actname : actmach \longrightarrow int$

$actname(\overline{\alpha}) = \; k$

where k is from $\overline{\alpha}.m_k[loc]$.

Function $cont : actmach \longrightarrow beh$

$cont(\overline{\alpha}) = \overline{\alpha}.\psi_{cont}$.

Function $content : task \longrightarrow msg$

$content(\overline{\tau}) = \overline{\tau}.\kappa$.

Function $recip : task \longrightarrow int$

$recip(\overline{\tau}) = \; k$

where k is from $\overline{\tau}.m_k$.

Function $deliveredtask : mailqSet \times actmach \longrightarrow task$

$deliveredtask(\overline{\mathcal{M}}, \overline{\alpha}) = m_k[loc]$

where $m_k \in \overline{\mathcal{M}} \bigwedge$

$\overline{\alpha}$ is bound to $\overline{\alpha}.m_k[loc]$

Figure 5.7: P^{act} Accessor Functions

Figure 5.9 contains predicate functions for actor machines in P^{act}. Predicate *und*? is true if an actor machine's task has not yet been delivered, predicate *del*? is true if an actor machine's task has been delivered but not consumed, and predicate *cons*? is true if an actor machine has consumed its task. In *del*? and *cons*?, the tests for equality and inequality are *syntactic* in the predicates' respective true cases.

5.1.4 P^{act} Decidable Predicates

We now present several decidable predicates in P^{act} that are useful for reasoning about distributed computation. Our first predicate deals with the consumption

Function *newcont* : $actmach \times beh \longrightarrow actmach$

$newcont(\overline{\alpha}, \psi) =$

$\overline{\alpha}'$ where

$\overline{\alpha}'.\psi_{cont} = \psi,$

$\overline{\alpha}'.* = \overline{\alpha}.*\,.$

Function *deliver* : $mailqSet \times task \longrightarrow mailqSet$

$deliver(\overline{\mathcal{M}}, \overline{\tau}) =$

$\overline{\mathcal{M}}'$ where

let $k = recip(\overline{\tau})$ in

$\overline{\mathcal{M}}' = (\overline{\mathcal{M}} - \{m_k\}) \bigcup \{append(m_k, \overline{\tau})\}.$

Function *retrieveMsg* : $actmach \times mailqSet \longrightarrow actmach$

$retrieveMsg(\overline{\alpha}, \overline{\mathcal{M}}) =$

$\overline{\alpha}'$ where

let $C[\;]$ be the evaluation context from $\overline{\alpha}.\psi_{cont}$,

and $k = content(deliveredtask(\overline{\mathcal{M}}, \overline{\alpha}))$ in

$\overline{\alpha}'.\psi_{cont} =$ replace $C[\;]$ with $C[k]$ in $\overline{\alpha}.\psi_{cont}$,

$\overline{\alpha}'.* = \overline{\alpha}.*.$

Figure 5.8: P^{act} Modifier Functions

of tasks. This activity is not observable since it occurs internal to some actor machine. It is, however decidable for an actor machine. We define the *consumed* function formally as:

Definition 15 ($consumed(\overline{\alpha})$) Boolean function *consumed* returns true if actor machine $\overline{\alpha}$ has consumed its task, and returns false otherwise. This is easily decided by comparing elements ψ_{init} and ψ_{cont} of $\overline{\alpha}$. If ψ_{init} and ψ_{cont} are syntactically equal, then $\overline{\alpha}$ has not begun its computation, and thus *consumed* returns false.

We define two states to be equivalent ($\cong$) if their respective $\overline{\mathcal{M}}$, $\overline{\mathcal{A}}$, and $\overline{\mathcal{T}}$ sets are identical. Formally:

Function $und?$: $mailqSet \times actmach \longrightarrow Bool$

$$und?(\overline{\mathcal{M}}, \overline{\alpha}) = \begin{cases} \text{True} & \text{if } \overline{\alpha}.m_k[loc] \text{ in } \overline{\mathcal{M}} \text{ Null}, \\ \text{False} & \text{otherwise.} \end{cases}$$

Function $del?$: $mailqSet \times actmach \longrightarrow Bool$

$$del?(\overline{\mathcal{M}}, \overline{\alpha}) = \begin{cases} \text{True} & \text{if } \overline{\alpha}.m_k[loc] \text{ in } \overline{\mathcal{M}} \text{ not Null } \bigwedge \ \overline{\alpha}.\psi_{init} = \overline{\alpha}.\psi_{cont}, \\ \text{False} & \text{otherwise.} \end{cases}$$

Function $cons?$: $mailqSet \times actmach \longrightarrow Bool$

$$cons?(\overline{\mathcal{M}}, \overline{\alpha}) = \begin{cases} \text{True} & \text{if } \overline{\alpha}.m_k[loc] \text{ in } \overline{\mathcal{M}} \text{ not Null } \bigwedge \ \overline{\alpha}.\psi_{init} \neq \overline{\alpha}.\psi_{cont}, \\ \text{False} & \text{otherwise.} \end{cases}$$

Figure 5.9: P^{act} Predicate Functins

Definition 16 ($\sigma_i \cong \sigma_j$) $\sigma_i \cong \sigma_j \iff$

$$(\sigma_i.\overline{\mathcal{M}} = \sigma_j.\overline{\mathcal{M}}) \bigwedge (\sigma_i.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}}) \bigwedge (\sigma_i.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}})$$

During computation, $\overline{\mathcal{S}}$ may enter one or more states of computational quiescence. Typically, computational quiescence is the result of $\overline{\mathcal{S}}$ waiting for an incoming task (message) from the environment. These periods of time can be detected by following the traversal of σ, searching for consecutive, computationally equivalent states.

The notion of an end of computation for $\overline{\mathcal{S}}$ in P^{act} is not practical. Candidates for this condition include traversals of σ in a current state of computational quiescence. In general, it is not decidable whether $\overline{\mathcal{S}}$, in a current state of computational quiescence, is also in an end of computation state. We cannot know whether $\overline{\mathcal{S}}$ will ever receive another inbound task. Furthermore, even if $\overline{\mathcal{S}}$ is in a current state σ_{cur} that contains no remaining actors performing computation, no tasks, and no actor machines waiting for inbound tasks, σ_{cur} could still be acting as a task conduit for its environment. Inbound tasks that immediately become outbound tasks in $\overline{\mathcal{S}}$ constitute meaningful computation.

5.2 Actor Theories

5.2.1 The AT Model

We briefly describe Actor Theories. For a complete presentation of Actor Theories, see Mason and Talcott [MT97]. An Actor Theory structure is a 3-tuple, defined as follows: $AT = \langle\ \langle \mathbf{A}, \mathbf{S}, \mathbf{M}, \mathbf{L} \rangle,\ \langle acq, \widehat{\_} \rangle, RR\ \rangle$.

The first AT element is a 4-tuple of actor theory primitives: actor names, actor states, message contents, and labels. From these primitives follow the specification of actor entities (actors), messages, and configuration interiors. Thus, $[s]_a$ represents an actor a in state s, $a \triangleleft M$ represents a message intended for recipient actor a with contents M, and I represents a multiset of actors and messages such that no two actor entities have the same name.

The second AT element is a 2-tuple containing the actor theory primitive operations. The acquaintance function acq extracts actor names from an actor state, the contents of a message, or a label. The renaming function $\widehat{\_}$ renames actor names within actor states, message contents, and labels.

The final AT element RR is a set of reaction rules. Reaction rules are triples of the form $l\ :\ I\ \Rightarrow\ I'$, where l is the reaction rule's label, I is the configuration interior prior to transition, and I' is the configuration interior that results from transition.

$$
\begin{array}{lll}
\text{(internal)} & \langle\!\langle\, I_0 \,,\, I \,\rangle\!\rangle^{\rho}_{\chi} \xrightarrow{\;l\;} \langle\!\langle\, I_1 \,,\, I \,\rangle\!\rangle^{\rho}_{\chi} & \text{if } l \;:\; I_0 \;\Rightarrow\; I_1 \;\in RR \\[2ex]
\text{(in)} & \langle\!\langle\, I \,\rangle\!\rangle^{\rho}_{\chi} \xrightarrow{\text{in}(a,M)} \langle\!\langle\, I, a \lhd M \,\rangle\!\rangle^{\rho}_{\chi \cup (acq(M) - \rho)} & \\[1ex]
 & & \text{if } a \in \rho \;\bigwedge\; acq(M) \cap InAct(I) \subseteq \rho \\[3ex]
\text{(out)} & \langle\!\langle\, I, a \lhd M \,\rangle\!\rangle^{\rho}_{\chi} \xrightarrow{\text{out}(a,M)} \langle\!\langle\, I \,\rangle\!\rangle^{\rho \cup (acq(M) - \chi)}_{\chi} & \text{if } a \notin InAct(I) \\[2ex]
\text{(idle)} & \langle\!\langle\, I \,\rangle\!\rangle^{\rho}_{\chi} \xrightarrow{\text{idle}} \langle\!\langle\, I \,\rangle\!\rangle^{\rho}_{\chi} &
\end{array}
$$

Figure 5.10: AT Transition Rules

An actor configuration consists of a configuration interior I, along with I's corresponding set of receptionists ρ and set of externals χ. The receptionists of I are members of a subset of the actors within I whose names have been communicated externally. The set of I's external actors contains those actor names referenced within I but not found in I.

The set of actor configurations is defined as follows:

$\mathbf{K} = \{\langle\!\langle\, I \,\rangle\!\rangle^{\rho}_{\chi} \;\mid\; \rho \subseteq InAct(I) \;\bigwedge\; ExtAct(I) \subseteq \chi\}$. K ranges over $\mathbf{K}$.

Finally, the AT transition rules are specified by a labeled transition relation of the form $K \xrightarrow{\;l\;} K'$, where the range of l includes not just the labels of rules within RR, but the three new label forms *in*, *out*, and *idle*. The heart of the AT semantics are the four transition rules found in Figure 5.10. An in transition reflects a message coming in from the environment; an out transition reflects a message transmitted to the environment. An idle transition does not change the actor configuration.

5.2.2 AT Predicates and Helper Functions

We assume the existence of two predicates on an actor state s, $ready?(s)$ and $busy?(s)$, that return true if an actor in state s is not busy computing and ready to receive a new message, or busy computing in its current state, respectively.

Since actor replacement behavior differs between AT and P^{act}, a one-to-many relationship exists between the actor machines in P^{act} and actors in AT. For comparison purposes between respective AT states and P^{act} states, we assume the existence of a helper function, $ICfg(I)$, to filter out those actors in I eligible for garbage collection [AMS97].

5.3 Equivalence Proof for actors

The equivalence proof presented in this section also appears in Smith, et al. [SPH98]. We prove the equivalence of a restricted version of P^{act}, denoted $\widehat{P^{act}}$, with AT, the Actor Theory semantics presented by Mason and Talcott [MT97].

5.3.1 P^{act} Restricted

Figure 5.11 contains restricted versions of the transition relation, and the inbound and outbound tasks functions. The restriction permits only singular transition density in $\widehat{P^{act}}$. The restricted transition relation elaborates only next states that reflect the computational progress of at most one outbound or inbound task, or the meaning of a single actor machine's computational progress.

$$
\begin{array}{l}
\mathcal{F}_{\hat{\delta}} : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_{\hat{\delta}}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \\
\quad\quad \text{if } \sigma_{next}\ \mathit{undefined} \\
\quad\quad\quad \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \mathcal{F}_{\widehat{in}}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) \rangle, \text{ or} \\
\quad\quad\quad \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \mathcal{F}_{\widehat{out}}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) \rangle, \text{ or} \\
\quad\quad\quad \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma' \rangle, \text{ where} \\
\quad\quad\quad\quad \sigma' \ \text{ is derived from } \mathcal{F}_{\mu}(\overline{\alpha}) \ \text{ where } \overline{\alpha}\ \in\ \overline{\mathcal{A}} \\
\quad\quad \text{else} \\
\quad\quad\quad \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \mathcal{F}_{\hat{\delta}}(\sigma_{next}) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}_{\widehat{out}} : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_{\widehat{out}}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \\
\quad\quad \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}} - \{\overline{\tau}_r\}, \emptyset, \mathit{undefined} \rangle \text{ where} \\
\quad\quad\quad \overline{\tau}_r\ \in\ \mathit{outboundtasks}(\overline{\mathcal{T}}, \overline{\mathcal{M}})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}_{\widehat{in}} : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_{\widehat{in}}(\langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle) = \\
\quad\quad \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}} \bigcup \{\overline{\tau}_r\}, \emptyset, \mathit{undefined} \rangle \text{ where} \\
\quad\quad\quad \overline{\tau}_r \ \text{ is from the environment}
\end{array}
$$

Figure 5.11: $\widehat{P^{act}}$ Restricted Transition Function

5.3.2 Theorem and Proof

The theorem and proof presented in this section rely on three equivalence relations that specify the conditions under which a state from $\widehat{P^{act}}$ and a state from AT are considered equivalent. We define the configuration equivalence relation $\underset{cfg}{\longmapsto}$ using actor equivalence relation $\underset{act}{\longmapsto}$ and message equivalence relation $\underset{msg}{\longmapsto}$. On a high level, $\underset{cfg}{\longmapsto}$ ensures that each active actor machine from P^{act} is equivalent to some actor in AT not eligible for garbage collection (i.e., active), and conversely, that each active actor from AT is equivalent to some active actor machine in $\widehat{P^{act}}$. Similarly, each undelivered tasks from $\widehat{P^{act}}$ must be equivalent to some undelivered message in AT, and vice versa.

The $\underset{act}{\longmapsto}$ relation accommodates two differences between actors in $\widehat{P^{act}}$ and AT. The first difference concerns actor machines in $\widehat{P^{act}}$ and anonymous actors in AT. The actor machine approach is consistent with that of Agha's original work [Agh86]. An actor machine's computation is a function of the task it consumes; an actor machine consumes only one task during its lifetime. The anonymous actors approach of AT is different, but equivalent; anonymous actors have already consumed their message, and by definition, no new messages can be sent to them. The use of function $anon()$ in $\underset{act}{\longmapsto}$ is consistent with renaming from AT. Both cases of $\underset{act}{\longmapsto}$ identify equivalent actor names by taking into account the possibility of renaming.

The second difference between actors in $\widehat{P^{act}}$ and AT is one of granularity. $\widehat{P^{act}}$ distinguishes message delivery from message consumption. For AT, message delivery and consumption are a single, atomic instance of computational progress. Among two candidate instances of actor behavior, $\underset{act}{\longmapsto}$ must determine the proper equivalence test based on the state of message delivery and consumption for the $\widehat{P^{act}}$ actor machine. The continuation of an actor machine whose task has been delivered but not consumed, for $\underset{act}{\longmapsto}$'s purposes, is a function of the unconsumed task.

The $\underset{msg}{\longmapsto}$ relation returns true for two messages (tasks) that have identical content, and whose recipients are the same actor. For both $\underset{msg}{\longmapsto}$ and $\underset{act}{\longmapsto}$, syntactic equality implies semantic equality. The three equivalence relations are defined as follows:

Definition 17 ($\underset{cfg}{\longmapsto}$) Let $K_i^{AT} = \left\langle\!\!\left\langle\; I \;\right\rangle\!\!\right\rangle_{\chi}^{\rho}$ and $K_j^{\widehat{P^{act}}} = \sigma_j$, where $\sigma_j = \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next}\rangle$. Then $K_i^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}$ iff

$$\Big(\forall [s]_a \in \mathit{ICfg}(I)\ \exists \overline{\alpha}_p \in \overline{\mathcal{A}} \text{ s.t. } [s]_a \underset{act}{\longmapsto} \overline{\alpha}_p\Big) \bigwedge$$
$$\Big(\forall \overline{\alpha}_p \in \overline{\mathcal{A}}\ \exists [s]_a \in \mathit{ICfg}(I) \text{ s.t. } [s]_a \underset{act}{\longmapsto} \overline{\alpha}_p\Big) \bigwedge$$
$$\Big(\forall a \triangleleft M \in I\ \exists \overline{\tau}_r \in \overline{\mathcal{T}} \text{ s.t. } a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r\Big) \bigwedge$$
$$\Big(\forall \overline{\tau}_r \in \overline{\mathcal{T}}\ \exists a \triangleleft M \in I \text{ s.t. } a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r\Big) \bigwedge$$
$$\Big(\forall a \in \rho\ \exists \overline{\alpha}_p \in \overline{\mathcal{A}} \text{ s.t. } \mathit{actname}(\overline{\alpha}_p) = a\Big) \bigwedge$$
$$\Big(\forall a \in \chi . \forall \overline{\alpha}_p \in \overline{\mathcal{A}} . \mathit{actname}(\overline{\alpha}_p) \neq a\Big).$$

Definition 18 ($\underset{act}{\longmapsto}$) $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$ iff

if $\mathit{del?}(\overline{\mathcal{M}}, \overline{\alpha}_p)$

$$\Big(\big(a = \mathit{actname}(\overline{\alpha}_p)\big) \bigvee \big(a \in \{b \mid b = \mathit{anon}(\mathit{actname}(\overline{\alpha}_p))\}\big)\Big) \bigwedge$$
$$\Big(s = \mathit{cont}(\mathit{retrieveMsg}(\overline{\alpha}_p, \overline{\mathcal{M}}))\Big).$$

else

$$\Big(\big(a = \mathit{actname}(\overline{\alpha}_p)\big) \bigvee \big(a \in \{b \mid b = \mathit{anon}(\mathit{actname}(\overline{\alpha}_p))\}\big)\Big) \bigwedge$$
$$\Big(s = \mathit{cont}(\overline{\alpha}_p)\Big).$$

Definition 19 ($\underset{msg}{\longmapsto}$) $a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$ iff

$$\Big(a = \mathit{recip}(\overline{\tau}_r)\Big) \bigwedge \Big(M = \mathit{content}(\overline{\tau}_r)\Big).$$

Theorem 1 states that we can model the computations of all Actor programs equivalently in both AT and P^{act}. Specifically, if the initial configuration of an Actor program pgm reaches a certain configuration under AT, it can reach an equivalent configuration under P^{act}. Similarly, if the initial configuration of an Actor program pgm reaches a certain configuration under P^{act}, it can reach an equivalent configuration under AT. The proof is by induction on the number of transitions, in both directions. The proof from AT to P^{act} must consider all the cases corresponding to possible transitions program pgm can make under AT. The proof from P^{act} to AT must consider all the cases corresponding to possible transitions program pgm can make under P^{act}.

Theorem 1 *For all actor programs,* pgm*, let* K_0^{AT} *be the initial configuration of* pgm *from* $\mathbf{K}^{AT^{big}}$(pgm)*, and* $K_0^{\widehat{P^{act}}}$ *be the initial configuration of* pgm *from* $\mathbf{K}^{\widehat{P^{act}}}$(pgm)*.*

$K_0^{AT} \overset{AT}{\Longrightarrow}{}^{*}\ K^{AT}$ *iff* $K_0^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow}{}^{*}\ K^{\widehat{P^{act}}}$

where

$K^{AT} \underset{cfg}{\longmapsto} K^{\widehat{P^{act}}}$.

Proof: $(\Longrightarrow)$

$\forall i \geq 0 . \exists j \geq 0 .\ K_0^{AT} \overset{AT}{\Longrightarrow}{}^{i}\ K_i^{AT}$

$\Longrightarrow\ K_0^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow}{}^{j}\ K_j^{\widehat{P^{act}}} \bigwedge K_i^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}$.

By induction on i.

Base: $(i = 0)$

$i = 0 \implies K_i^{AT} = K_0^{AT}$.

So, it suffices to prove $\exists j \geq 0$ s.t. $K_0^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow}{}^{j}\ K_j^{\widehat{P^{act}}} \bigwedge K_0^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}$.

But K_0^{AT} is the initial configuration of pgm in AT^{big}.

$\Longrightarrow\ K_0^{AT} = \left\langle\!\!\left\langle\ I\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$, where I contains one actor, $[s]_a$, and no messages. Thus $I = \{[s]_a\}$.

$\Longrightarrow\ \rho = \{a\}$, K_0^{AT}'s only receptionist; and $\chi = \emptyset$, since initially, pgm has no knowledge of external actors.

$K_0^{\widehat{P^{act}}}$ is the initial configuration of pgm in P^{act}.

$\Longrightarrow\ K_0^{\widehat{P^{act}}} = \sigma_0 = \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle$.

$\Longrightarrow\ \overline{\mathcal{P}} = \emptyset$, and initially, σ_{next} is undefined; and

$\overline{\mathcal{M}} = \{a\}$,

$\overline{\mathcal{A}} = \{\overline{\alpha}_p\}$, where $(actname(\overline{\alpha}_p) = a) \bigwedge (cont(\overline{\alpha}_p) = s)$, and

$\overline{\mathcal{T}} = \emptyset$.

From the definition of $\underset{cfg}{\longmapsto}$, w.r.t. K_0^{AT} and $K_0^{\widehat{P^{act}}}$, notice

$[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$ satisfies the condition for actor correspondence,
message correspondence is true vacuously,
the condition for receptionists is true, since $a \in \rho$ and $actname(\overline{\alpha}_p) = a$,
and the condition for external actors is true vacuously.

$\therefore$ By definition of $\underset{cfg}{\longmapsto}$, for $i = 0, j = 0$,

$$K_0^{AT} \underset{cfg}{\longmapsto} K_0^{\widehat{P^{act}}} \text{ is true.}$$

I.H.: Assume for some $i \geq 0$ steps,

$K_0^{AT} \overset{AT}{\Longrightarrow}^i K_i^{AT}$ implies $\exists j \geq 0$, s.t.

$$K_0^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow}^j K_j^{\widehat{P^{act}}} \bigwedge K_i^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}.$$

I.S.: (Prove true for $i+1$ steps)

$K_0^{AT} \overset{AT}{\Longrightarrow}^{i+1} K_{i+1}^{AT} =$

$$K_0^{AT} \overset{AT}{\Longrightarrow}^i K_i^{AT} \overset{AT}{\Longrightarrow}^1 K_{i+1}^{AT}, \text{ by definition of } \overset{AT}{\Longrightarrow}.$$

By I.H. we know $\exists j$ s.t.

$$K_0^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow}^j K_j^{\widehat{P^{act}}} \bigwedge K_i^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}.$$

Consider cases for $K_i^{AT} \overset{AT}{\Longrightarrow}^1 K_{i+1}^{AT}$. Prove $\exists n \geq 0$ s.t.

$$K_j^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow}^n K_{j+n}^{\widehat{P^{act}}} \bigwedge K_{i+1}^{AT} \underset{cfg}{\longmapsto} K_{j+n}^{\widehat{P^{act}}}.$$

Case 1a: create actor

- Let $K_i^{AT} = \left\langle\!\!\left\langle\, I_i, I \,\right\rangle\!\!\right\rangle_\chi^\rho$, $K_{i+1}^{AT} = \left\langle\!\!\left\langle\, I_{i+1}, I \,\right\rangle\!\!\right\rangle_\chi^\rho$, and $K_i^{AT} \overset{AT}{\Longrightarrow} K_{i+1}^{AT}$, where $I_i = \{[s]_a\}$ and $I_{i+1} = \{[s]_a, [s']_a, [t]_b\}$.
- By I.H. and $\underset{cfg}{\longmapsto}$,

  $\exists \overline{\alpha}_p \in \sigma_j.\overline{\mathcal{A}}$, s.t. $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$.

- By definition of $\overset{AT}{\Longrightarrow}$, $Am([s]_a)$ yields $\{[s']_a, [t]_b\}$

  $\therefore$ By definition of $\mathcal{F}_\mu$, $\exists \mu \in \mathcal{F}_\mu(\overline{\alpha}_p)$, where

  $\mu = \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$,

  $cont(\overline{\alpha}_{cont}) = s'$ (syntactic equality implies semantic equality),

  $\overline{\mathcal{A}}_{new} = \{\overline{\alpha}_r\}$,

  $actname(\overline{\alpha}_r) = b$, and

  $\overline{\mathcal{M}}_{new} = \{b\}$.

  $\therefore\ [t]_b \underset{act}{\longmapsto} \overline{\alpha}_r$.

- By definition of $\mathcal{G}$, for $n = 1$,

  $\exists \sigma_{j+1}$ s.t. $\sigma_{j+1} = \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

  $\overline{\mathcal{M}} = \sigma_j.\overline{\mathcal{M}} \bigcup \overline{\mathcal{M}}_{new}$,

  $\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}_{cont}\} \bigcup \overline{\mathcal{A}}_{new}$,

  $\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,

  $\overline{\mathcal{P}} = \emptyset$, and

  σ_{next} is initially undefined,

  where

  $[s']_a \underset{act}{\longmapsto} \overline{\alpha}_{cont}$, since $cont(\overline{\alpha}_{cont}) = s'$, $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$,

  and by definition of $\overline{\alpha}_{cont}$.

  $\therefore$ By definition of $\underset{cfg}{\longmapsto}$, $K^{AT}_{i+1} \underset{cfg}{\longmapsto} K^{\widehat{P^{act}}}_{j+1}$.

- By definition of $\mathcal{F}_{\hat{\delta}}$ (limited transition function), $\sigma_j.\sigma_{next} = \sigma_{j+1}$ is one legal transition.

  $\therefore\ K^{\widehat{P^{act}}}_{j} \overset{\widehat{P^{act}}}{\Longrightarrow} K^{\widehat{P^{act}}}_{j+1}$.

- $\therefore$ Proved I.S. for case 1a.

Case 1b: send message

- Let $K^{AT}_i = \left\langle\!\!\left\langle\ I_i, I\ \right\rangle\!\!\right\rangle^{\rho}_{\chi}$, $K^{AT}_{i+1} = \left\langle\!\!\left\langle\ I_{i+1}, I\ \right\rangle\!\!\right\rangle^{\rho}_{\chi}$, and $K^{AT}_i \overset{AT}{\Longrightarrow} K^{AT}_{i+1}$,

  where $I_i = \{[s]_a\}$ and $I_{i+1} = \{[s]_a, [s']_a, b \lhd M\}$.

- By I.H. and $\underset{cfg}{\longmapsto}$,

  $\exists \overline{\alpha}_p \in \sigma_j.\overline{\mathcal{A}}$, s.t. $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$.

- By definition of $\overset{AT}{\Longrightarrow}$, $Am([s]_a)$ yields $\{[s']_a, b \triangleleft M\}$

  $\therefore$ By definition of $\mathcal{F}_\mu$, $\exists \mu \in \mathcal{F}_\mu(\overline{\alpha}_p)$, where

  $\mu = \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$,

  $cont(\overline{\alpha}_{cont}) = s'$ (syntactic equality implies semantic equality),

  $\overline{\mathcal{T}}_{new} = \{\overline{\tau}_r\}$,

  $\therefore\ b \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$.

- By definition of $\mathcal{G}$, for $n = 1$,

  $\exists \sigma_{j+1}$ s.t. $\sigma_{j+1} = \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

  $\overline{\mathcal{M}} = \sigma_j.\overline{\mathcal{M}}$,

  $\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}_{cont}\}$,

  $\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}} \bigcup \overline{\mathcal{T}}_{new}$,

  $\overline{\mathcal{P}} = \{\varepsilon\}$, where $\varepsilon = \langle \mathbf{E_S}, \overline{\tau}_r \rangle$ (a send event), and

  σ_{next} is initially undefined,

  where

  $[s']_a \underset{act}{\longmapsto} \overline{\alpha}_{cont}$, since $cont(\overline{\alpha}_{cont}) = s'$, $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$,

  and by definition of $\overline{\alpha}_{cont}$.

  $\therefore$ By definition of $\underset{cfg}{\longmapsto}$, $K^{AT}_{i+1} \underset{cfg}{\longmapsto} K^{\widehat{P^{act}}}_{j+1}$.

- By definition of $\mathcal{F}_{\hat{\delta}}$ (limited transition function), $\sigma_j.\sigma_{next} = \sigma_{j+1}$ is one legal transition.

  $\therefore\ K^{\widehat{P^{act}}}_j \overset{\widehat{P^{act}}}{\Longrightarrow} K^{\widehat{P^{act}}}_{j+1}$.

- $\therefore$ Proved I.S. for case 1b.

Case 1c: actor/message synchronization

- Let $K^{AT}_i = \left\langle\!\!\left\langle\ I_i, I\ \right\rangle\!\!\right\rangle^{\rho}_{\chi}$, $K^{AT}_{i+1} = \left\langle\!\!\left\langle\ I_{i+1}, I\ \right\rangle\!\!\right\rangle^{\rho}_{\chi}$, and $K^{AT}_i \overset{AT}{\Longrightarrow} K^{AT}_{i+1}$,

  where $I_i = \{[s]_a, a \triangleleft M\}$ and $I_{i+1} = \{[s]_a, [s']_a\}$,

  s.t. *ready*?$([s]_a)$ is true $\bigwedge$ *busy*?$([s']_a)$ is true.

- By I.H. and $\underset{cfg}{\longmapsto}$,

  $\exists \overline{\alpha}_p \in \sigma_j.\overline{\mathcal{A}}$, s.t. $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p \bigwedge$

  $\exists \overline{\tau}_r \in \sigma_j.\overline{\mathcal{T}}$, s.t. $a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$.

- By definition of $\stackrel{AT}{\Longrightarrow}$, $Am([s]_a)$ yields $\{[s']_a\}$.

  $\therefore$ By definition of $\mathcal{F}_\mu$, $\exists \mu \in \mathcal{F}_\mu(\overline{\alpha}_p)$, where

  $\mu = \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$,

  $cont(\overline{\alpha}_{cont}) = s'$ (syntactic equality implies semantic equality).

- By definition of $\mathcal{G}$, for $n = 1$,

  $\exists \sigma_{j+1}$ s.t. $\sigma_{j+1} = \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

  $\overline{\mathcal{M}} = deliver(\sigma_j.\overline{\mathcal{M}}, \overline{\tau}_r)$,

  $\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}_{cont}\}$,

  $\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}} - \{\overline{\tau}_r\}$,

  $\overline{\mathcal{P}} = \{\varepsilon\}$, where $\varepsilon = \langle \mathbf{E_D}, \overline{\tau}_r \rangle$ (a deliver event), and

  σ_{next} is initially undefined,

  where

  $[s']_a \underset{act}{\longmapsto} \overline{\alpha}_{cont}$, since $cont(\overline{\alpha}_{cont}) = s'$, $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$,

  and by definition of $\overline{\alpha}_{cont}$.

  $\therefore$ By definition of $\underset{cfg}{\longmapsto}$, $K_{i+1}^{AT} \underset{cfg}{\longmapsto} K_{j+1}^{\widehat{P^{act}}}$.

- By definition of $\mathcal{F}_{\hat{\delta}}$ (limited transition function), $\sigma_j.\sigma_{next} = \sigma_{j+1}$ is one legal transition.

  $\therefore K_j^{\widehat{P^{act}}} \stackrel{\widehat{P^{act}}}{\Longrightarrow} K_{j+1}^{\widehat{P^{act}}}$.

- $\therefore$ Proved I.S. for case 1c.

Case 1d: replacement specification

- Let $K_i^{AT} = \left\langle\!\!\left\langle\, I_i, I \,\right\rangle\!\!\right\rangle_\chi^\rho$, $K_{i+1}^{AT} = \left\langle\!\!\left\langle\, I_{i+1}, I \,\right\rangle\!\!\right\rangle_\chi^\rho$, and $K_i^{AT} \stackrel{AT}{\Longrightarrow} K_{i+1}^{AT}$,

  where $I_i = \{[s]_a\}$ and $I_{i+1} = \{[s]_a, [s']_{(a)}, [t]_a\}$.

- By I.H. and $\underset{cfg}{\longmapsto}$,

  $\exists \overline{\alpha}_p \in \sigma_j.\overline{\mathcal{A}}$, s.t. $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$

- By definition of $\overset{AT}{\Longrightarrow}$, $Am([s]_a)$ yields $\{[t]_a, [s']_{(a)}\}$, where t is the replacement behavior continuation of actor a, and s' is the continuation of actor a's initial behavior, s, carried out by an anonymous actor (a).

  $\therefore$ By definition of $\mathcal{F}_\mu$, $\exists \mu \in \mathcal{F}_\mu(\overline{\alpha}_p)$, where

  $\mu = \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$,

  $\overline{\mathcal{A}}_{repl} = \{\overline{\alpha}_r\}$,

  $cont(\overline{\alpha}_r) = t$ (syntactic equality implies semantic equality),

  $cont(\overline{\alpha}_{cont}) = s'$ (syntactic equality implies semantic equality).

- By definition of $\mathcal{G}$, for $n = 1$,

  $\exists \sigma_{j+1}$ s.t. $\sigma_{j+1} = \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

  $\overline{\mathcal{M}} = \sigma_j.\overline{\mathcal{M}} \bigcup \{(a)\}$,

  $\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}_{cont}\} \bigcup \overline{\mathcal{A}}_{repl}$,

  $\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,

  $\overline{\mathcal{P}} = \emptyset$, and

  σ_{next} is initially undefined,

  where

  $[s']_{(a)} \underset{act}{\longmapsto} \overline{\alpha}_{cont}$, since $cont(\overline{\alpha}_{cont}) = s'$,

  and by definition of (a), $\underset{act}{\longmapsto}$, and $\overline{\alpha}_{cont}$

  and

  $[t]_a \underset{act}{\longmapsto} \overline{\alpha}_r$ since $cont(\overline{\alpha}_r) = t$, $[s]_a \underset{act}{\longmapsto} \overline{\alpha}_p$,

  and by definition of $\overline{\alpha}_r$.

  $\therefore$ By definition of $\underset{cfg}{\longmapsto}$, $K^{AT}_{i+1} \underset{cfg}{\longmapsto} K^{\widehat{P^{act}}}_{j+1}$.

- By definition of $\mathcal{F}_{\hat{\delta}}$ (limited transition function), $\sigma_j.\sigma_{next} = \sigma_{j+1}$ is one legal transition.

  $\therefore K^{\widehat{P^{act}}}_j \overset{\widehat{P^{act}}}{\Longrightarrow} K^{\widehat{P^{act}}}_{j+1}$.

- $\therefore$ Proved I.S. for case 1d.

Case 2: in

- Let $K_i^{AT} = \left\langle\!\!\left\langle\ I\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$, $K_{i+1}^{AT} = \left\langle\!\!\left\langle\ I, a \triangleleft M\ \right\rangle\!\!\right\rangle_{\chi \bigcup (acq(M)-\rho)}^{\rho}$, and $K_i^{AT} \overset{AT}{\Longrightarrow} K_{i+1}^{AT}$,

  where $(a \in \rho)\ \bigwedge\ (acq(M) \bigcap\ InAct(I) \subseteq\ \rho)$.

- By I.H. and $\underset{cfg}{\longmapsto}$,

  $\exists \overline{\alpha}_p \in \sigma_j.\overline{\mathcal{A}}$, s.t. $actname(\overline{\alpha}_p) = a$

- By definition of $\overset{AT}{\Longrightarrow}$, message $a \triangleleft M$ is added to K_{i+1}^{AT}'s internal configuration by an external entity.

  $\therefore \exists \overline{\tau}_r$ from the same external entity, s.t. $a \triangleleft M\ \underset{msg}{\longmapsto}\ \overline{\tau}_r$

  ($\Longrightarrow recip(\overline{\tau}_r) = actname(\overline{\alpha}_p)$, by definition of $\underset{msg}{\longmapsto}$)

- By definition of $\mathcal{G}$ and $\mathcal{F}_{\widehat{in}}$, for $n = 1$,

  $\exists \sigma_{j+1}$ s.t. $\sigma_{j+1} = \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

  $\overline{\mathcal{M}} = \sigma_j.\overline{\mathcal{M}}$,

  $\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}}$,

  $\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}\ \bigcup\ \{\overline{\tau}_r\}$,

  $\overline{\mathcal{P}} = \{\varepsilon\}$, where $\varepsilon = \langle \mathbf{E_S}, \overline{\tau}_r \rangle$ (a send event), and

  σ_{next} is initially undefined,

  $\therefore K_{i+1}^{AT}\ \underset{cfg}{\longmapsto}\ K_{j+1}^{\widehat{P_{act}}}$, since by definition of $\underset{cfg}{\longmapsto}$, I.H., and $\overset{AT}{\Longrightarrow}$, only $a \triangleleft M$ is added to K_i^{AT}'s internal configuration, and $a \triangleleft M\ \underset{msg}{\longmapsto}\ \overline{\tau}_r$. Conditions for actors and receptionists in $\underset{cfg}{\longmapsto}$ are unchanged. $(acq(M) - \rho)$ adds at most external actor names to χ, thus preserving the condition for external actors.

- By definition of $\mathcal{F}_{\hat{\delta}}$ (limited transition function), $\sigma_j.\sigma_{next} = \sigma_{j+1}$ is one legal transition.

  $\therefore\ K_j^{\widehat{P_{act}}} \overset{\widehat{P_{act}}}{\Longrightarrow} K_{j+1}^{\widehat{P_{act}}}$.

- $\therefore$ Proved I.S. for case 2.

Case 3: out

- Let $K_i^{AT} = \left\langle\!\!\left\langle\ I, a \triangleleft M\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$, $K_{i+1}^{AT} = \left\langle\!\!\left\langle\ I\ \right\rangle\!\!\right\rangle_{\chi}^{\rho \bigcup (acq(M)-\chi)}$, and $K_i^{AT} \overset{AT}{\Longrightarrow} K_{i+1}^{AT}$,

  where $a \notin InAct(I)$
- By I.H. and $\underset{cfg}{\longmapsto}$,

  $\exists \overline{\tau}_r \in \sigma_j.\overline{\mathcal{T}}$, s.t. $(a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r) \bigwedge (recip(\overline{\tau}_r) \notin \sigma_j.\overline{\mathcal{M}})$.
- By definition of $\overset{AT}{\Longrightarrow}$, message $a \triangleleft M$ is removed from internal configuration of K_{i+1}^{AT}.
- By definition of $\mathcal{G}$ and $\mathcal{F}_{\widehat{out}}$, for $n = 1$,

  $\exists \sigma_{j+1}$ s.t. $\sigma_{j+1} = \langle \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

  $\overline{\mathcal{M}} = \sigma_j.\overline{\mathcal{M}}$,

  $\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}}$,

  $\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}} - \{\overline{\tau}_r\}$,

  $\overline{\mathcal{P}} = \emptyset$, and

  σ_{next} is initially undefined,

  $\therefore K_{i+1}^{AT} \underset{cfg}{\longmapsto} K_{j+1}^{\widehat{P^{act}}}$, since by I.H., definition of $\underset{cfg}{\longmapsto}$, and $\overset{AT}{\Longrightarrow}$,

  only $a \triangleleft M$ is removed from K_i^{AT}'s internal configuration, and $a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$. Conditions for actors and external actor names are unchanged. $(acq(M) - \chi)$ adds at most internally defined actor names to ρ, thus preserving the condition for receptionists.
- By definition of $\mathcal{F}_{\hat{\delta}}$ (limited transition function), $\sigma_j.\sigma_{next} = \sigma_{j+1}$ is one legal transition.

  $\therefore\ K_j^{\widehat{P^{act}}} \overset{\widehat{P^{act}}}{\Longrightarrow} K_{j+1}^{\widehat{P^{act}}}$.
- $\therefore$ Proved I.S. for case 3.

Case 4: idle

- Let $K_i^{AT} = \left\langle\!\!\left\langle\ I\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$, $K_{i+1}^{AT} = \left\langle\!\!\left\langle\ I\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$, and $K_i^{AT} \overset{AT}{\Longrightarrow} K_{i+1}^{AT}$,

  $\therefore K_i^{AT} = K_{i+1}^{AT}$

- By I.H.,

  $K_i^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}$, and since $K_i^{AT} = K_{i+1}^{AT}$,

  $\therefore K_{i+1}^{AT} \underset{cfg}{\longmapsto} K_j^{\widehat{P^{act}}}$.

- By definition of $\xRightarrow{AT}$, after zero transitions ($n = 0$),

  $\therefore K_j^{\widehat{P^{act}}} \xRightarrow{\widehat{P^{act}}}{}^{0} K_j^{\widehat{P^{act}}}$.

- $\therefore$ Proved I.S. for case 4.

Proof: ($\Longleftarrow$)

$\forall i \geq 0. \exists j \geq 0.\ K_0^{\widehat{P^{act}}} \xRightarrow{\widehat{P^{act}}}{}^{i} K_i^{\widehat{P^{act}}}$

$\quad \Longrightarrow\ K_0^{AT} \xRightarrow{AT}{}^{j} K_j^{AT} \bigwedge K_i^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_j^{AT}$.

By induction on i, where i is the height of tree with root σ_0.

Base: ($i = 0$)

$i = 0 \implies K_i^{\widehat{P^{act}}} = K_0^{\widehat{P^{act}}}$.

So, it suffices to prove $\exists j \geq 0$ s.t. $K_0^{AT} \xRightarrow{AT}{}^{j} K_j^{AT} \bigwedge K_0^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_j^{AT}$.

But $K_0^{\widehat{P^{act}}} = \sigma_0$, its root, the initial configuration of pgm in $\widehat{P^{act}}$.

$\Longrightarrow \sigma_0 = \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle$, where σ_0 contains one actor machine, $\overline{\alpha}_p$, whose continuation corresponds to pgm, initially denoted s; and no tasks.

Thus,

$\overline{\mathcal{M}} = \{a\}$,

$\overline{\mathcal{A}} = \{\overline{\alpha}_p\}$, where $(actname(\overline{\alpha}_p) = a) \bigwedge (cont(\overline{\alpha}_p) = s)$,

$\overline{\mathcal{T}} = \emptyset$,

$\overline{\mathcal{P}} = \emptyset$, and

σ_{next} is initially undefined.

K_0^{AT} is the initial configuration of pgm in AT^{big}.

$\Longrightarrow K_0^{AT} = \left\langle\!\left\langle\ I\ \right\rangle\!\right\rangle_{\chi}^{\rho}$, where I contains one actor, $[s]_a$ and no messages.

Thus $I = \{[s]_a\}$, $\rho = \{a\}$, and $\chi = \emptyset$.

By definition of $\underset{cfg}{\longmapsto}$, $K_0^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_0^{AT}$, since

$\overline{\alpha}_p \underset{act}{\longmapsto} [s]_a$,
message correspondence is true vacuously,
the condition for receptionists is true, since $a \in \rho$ and $actname(\overline{\alpha}_p) = a$,
and the condition for external actors is true, vacuously.

$\therefore$ Base case holds with $j = 0$.

I.H.: Assume for tree σ_0 of height $i \geq 0$,

$\mathcal{F}_{\hat{\delta}}^i(K_0^{\widehat{P^{act}}}) = K_i^{\widehat{P^{act}}}$ implies $\exists j \geq 0$, s.t.

$$K_0^{AT} \overset{AT}{\Longrightarrow}{}^j K_j^{AT} \bigwedge K_i^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_j^{AT}.$$

I.S.: Prove true for tree σ_0 of height i+1:

That is, prove $\mathcal{F}_{\hat{\delta}}^{i+1}(K_0^{\widehat{P^{act}}}) = K_{i+1}^{\widehat{P^{act}}}$ implies

$$\exists k \geq 0, \text{ s.t. } K_0^{AT} \overset{AT}{\Longrightarrow}{}^k K_k^{AT} \bigwedge K_{i+1}^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_k^{AT}.$$

By definition of $\mathcal{F}_{\hat{\delta}}$, $\mathcal{F}_{\hat{\delta}}^{i+1}(K_0^{\widehat{P^{act}}}) = \mathcal{F}_{\hat{\delta}}(\mathcal{F}_{\hat{\delta}}^i(K_0^{\widehat{P^{act}}}))$.

By I.H., $\mathcal{F}_{\hat{\delta}}^i(K_0^{\widehat{P^{act}}}) = K_i^{\widehat{P^{act}}}$ implies

$$\exists j \text{ s.t. } K_0^{AT} \overset{AT}{\Longrightarrow}{}^j K_j^{AT} \bigwedge K_i^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_j^{AT}.$$

Consider cases for $\mathcal{F}_{\hat{\delta}}(K_i^{\widehat{P^{act}}}) = K_{i+1}^{\widehat{P^{act}}}$.

Prove $\exists n \geq 0$, where $k = j + n$, s.t.

$$K_j^{AT} \overset{AT}{\Longrightarrow}{}^n K_{j+n}^{AT} \bigwedge K_{i+1}^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_{j+n}^{AT}.$$

Let $K_i^{\widehat{P^{act}}} = \langle \overline{\mathcal{M}},\ \overline{\mathcal{A}},\ \overline{\mathcal{T}},\ \overline{\mathcal{P}},\ \sigma_{next} \rangle$.

Case 1a: create actor

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}',\ \overline{\mathcal{A}}',\ \overline{\mathcal{T}}',\ \overline{\mathcal{P}}',\ \sigma_{next}' \rangle$, where σ'_{next} is initially undefined, and by definition of create actor:

  $\exists b \in \overline{\mathcal{M}}'$ s.t. $b \notin \overline{\mathcal{M}}$,

  $\exists \overline{\alpha}_r \in \overline{\mathcal{A}}'$ s.t. $\overline{\alpha}_r \notin \overline{\mathcal{A}} \bigwedge actname(\overline{\alpha}_r) = b$,

$\exists\overline{\alpha}'_p \in \overline{\mathcal{A}}'$, $\overline{\alpha}_p \in \overline{\mathcal{A}}$ s.t. $actname(\overline{\alpha}'_p) = a \ \bigwedge \ actname(\overline{\alpha}_p) = a \ \bigwedge$
$cont(\overline{\alpha}_p) \neq cont(\overline{\alpha}'_p)$,

s.t. $\overline{\mathcal{M}}' = \overline{\mathcal{M}} \bigcup \{b\}$,

$\overline{\mathcal{A}}' = (\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}'_p, \overline{\alpha}_r\}$,

$\overline{\mathcal{T}}' = \overline{\mathcal{T}}$, and

$\overline{\mathcal{P}}' = \emptyset$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle \ I_j \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists[s]_a \in I_j$ s.t. $\overline{\alpha}_p \ \underset{act}{\longmapsto} \ [s]_a$

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $cont(\overline{\alpha}_p) = s$.

- By definition of $\mathcal{F}_\mu$, $\mathcal{F}_\mu(\overline{\alpha}_p) \ = \mu \ = \ \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$,

  where $\overline{\mathcal{M}}_{new} = \{b\}$, $\overline{\mathcal{A}}_{new} = \{\overline{\alpha}_r\}$.

  $\therefore$ By definition of $\mathcal{F}_\mu$ and $\mathcal{F}_{\hat{\delta}}$, $\overline{\alpha}_{cont} = \overline{\alpha}'_p$.

- By definition of Am, $Am([s]_a)$ yields $\{[s']_a, [t]_b\}$, where $s' = cont(\overline{\alpha}'_p)$ and $t = cont(\overline{\alpha}_r)$. Let $I_{j+1} = I_j \bigcup \{[s']_a, [t]_b\}$.

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $\overline{\alpha}'_p \ \underset{act}{\longmapsto} \ [s']_a$ and $\overline{\alpha}_r \ \underset{act}{\longmapsto} \ [t]_b$.

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 1$,

  $\exists K_{j+1}^{AT}$ s.t. $K_j^{AT} \overset{AT}{\Longrightarrow}{}^1 \ K_{j+1}^{AT}$, where $K_{j+1}^{AT} = \left\langle\!\!\left\langle \ I_{j+1} \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$.

- By definition of $\underset{cfg}{\longmapsto}$,

  $\therefore K_{i+1}^{\widehat{P^{act}}} \ \underset{cfg}{\longmapsto} \ K_{j+1}^{AT}$.

- Proved I.S. for case 1a.

Case 1b: create task

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}',\ \overline{\mathcal{A}}',\ \overline{\mathcal{T}}',\ \overline{\mathcal{P}}',\ \sigma_{next}{}' \rangle$, where σ'_{next} is initially undefined, and by definition of create task:

  $\exists b \in \overline{\mathcal{M}}'$ s.t. $b \notin \overline{\mathcal{M}}$,

  $\exists \overline{\tau}_r \in \overline{\mathcal{T}}'$ s.t. $\overline{\tau}_r \notin \overline{\mathcal{T}}$, and

  $\exists \overline{\alpha}'_p \in \overline{\mathcal{A}}'$, $\overline{\alpha}_p \in \overline{\mathcal{A}}$ s.t. $actname(\overline{\alpha}'_p) = a \ \bigwedge \ actname(\overline{\alpha}_p) = a \ \bigwedge$

  $$cont(\overline{\alpha}_p) \neq cont(\overline{\alpha}'_p),$$

  s.t. $\overline{\mathcal{M}}' = \overline{\mathcal{M}}$,

  $\overline{\mathcal{A}}' = (\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}'_p\}$,

  $\overline{\mathcal{T}}' = \overline{\mathcal{T}} \bigcup \{\overline{\tau}_r\}$, and

  $\overline{\mathcal{P}}' = \{\varepsilon\}$, where $\varepsilon = \langle \mathbf{E_S}, \overline{\tau}_r \rangle$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle \ I_j \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists [s]_a \in I_j$ s.t. $\overline{\alpha}_p \ \underset{act}{\longmapsto} \ [s]_a$

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $cont(\overline{\alpha}_p) = s$.

- By definition of $\mathcal{F}_\mu$, $\mathcal{F}_\mu(\overline{\alpha}_p) \ = \mu \ = \ \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$, where $\overline{\mathcal{T}}_{new} = \{\overline{\tau}_r\}$.

  $\therefore$ By definition of $\mathcal{F}_\mu$ and $\mathcal{F}_{\hat{\delta}}$, $\overline{\alpha}_{cont} = \overline{\alpha}'_p$.

- By definition of Am, $Am([s]_a)$ yields $\{[s']_a, b \lhd M\}$, where $s' = cont(\overline{\alpha}'_p) \ \bigwedge \ b = recip(\overline{\tau}_r) \ \bigwedge \ M = content(\overline{\tau}_r)$.

  Let $I_{j+1} = I_j \bigcup \{[s']_a, b \lhd M\}$.

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $\overline{\alpha}'_p \ \underset{act}{\longmapsto} \ [s']_a$; by definition of $\underset{msg}{\longmapsto}$, $\overline{\tau}_r \ \underset{msg}{\longmapsto} \ b \lhd M$.

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 1$,

  $\exists K_{j+1}^{AT}$ s.t. $K_j^{AT} \overset{AT}{\Longrightarrow}{}^1 K_{j+1}^{AT}$, where $K_{j+1}^{AT} = \left\langle\!\!\left\langle \ I_{j+1} \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$.

- By definition of $\underset{cfg}{\longmapsto}$,

  $\therefore K_{i+1}^{\widehat{P^{act}}} \ \underset{cfg}{\longmapsto} \ K_{j+1}^{AT}$.

- Proved I.S. for case 1b.

Case 1c: deliver task

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}',\ \overline{\mathcal{A}}',\ \overline{\mathcal{T}}',\ \overline{\mathcal{P}}',\ \sigma_{next}{}' \rangle$,

  where σ'_{next} is initially undefined, and by definition of deliver task:

  $\exists \overline{\tau}_r \in \overline{\mathcal{T}}$ s.t. $\overline{\tau}_r \notin \overline{\mathcal{T}}'$,

  and $\exists a \in \overline{\mathcal{M}}'$ s.t. $recip(\overline{\tau}_r) = a$,

  and $\exists \overline{\alpha}_p \in \overline{\mathcal{A}}'$, $\overline{\alpha}_p \in \overline{\mathcal{A}}$ s.t. $actname(\overline{\alpha}_p) = a \ \bigwedge$

  $del?(\overline{\mathcal{M}}', \overline{\alpha}_p)$ True $\bigwedge$ $und?(\overline{\mathcal{M}}, \overline{\alpha}_p)$ True,

  s.t. $\overline{\mathcal{M}}' = deliver(\overline{\mathcal{M}}, \overline{\tau}_r)$,

  $\overline{\mathcal{A}}' = \overline{\mathcal{A}}$,

  $\overline{\mathcal{T}}' = \overline{\mathcal{T}} - \{\overline{\tau}_r\}$, and

  $\overline{\mathcal{P}}' = \{\varepsilon\}$, where $\varepsilon = \langle \mathbf{E_D}, \overline{\tau}_r \rangle$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle\ I_j\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists [s]_a \in I_j$ s.t. $\overline{\alpha}_p \underset{act}{\longmapsto} [s]_a \ \bigwedge\ \exists a \triangleleft M \in I_j$ s.t. $\overline{\tau}_r \underset{msg}{\longmapsto} a \triangleleft M$.

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $cont(\overline{\alpha}_p) = s$

  (syntactic equality implies semantic equality)

  and by definition of $\underset{msg}{\longmapsto}$, $content(\overline{\tau}_r) = M$

  (syntactic equality implies semantic equality)

- By definition of actor/message synchronization,

  Let $I_{j+1} = (I_j - \{a \triangleleft M\}) \bigcup \{[s']_a\}$, where $Am([s]_a)$ yields $\{[s']_a\}$, s.t. $ready?([s]_a)$ True $\bigwedge$ $busy?([s']_a)$ True

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, for σ_{i+1}, $\overline{\alpha}_p \underset{act}{\longmapsto} [s']_a$

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 1$,

  $\exists K_{j+1}^{AT}$ s.t. $K_j^{AT} \overset{AT}{\Longrightarrow}{}^1 K_{j+1}^{AT}$, where $K_{j+1}^{AT} = \left\langle\!\!\left\langle\ I_{j+1}\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$.

- By definition of $\underset{cfg}{\longmapsto}$,

  $\therefore K_{i+1}^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_{j+1}^{AT}$.

- Proved I.S. for case 1c.

Case 1d: consume task

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}', \ \overline{\mathcal{A}}', \ \overline{\mathcal{T}}', \ \overline{\mathcal{P}}', \ \sigma_{next}{}' \rangle$, where σ'_{next} is initially undefined, and by definition of consume task:

  $\exists \overline{\alpha}'_p \in \overline{\mathcal{A}}', \ \overline{\alpha}_p \in \overline{\mathcal{A}}$ s.t.

  $actname(\overline{\alpha}'_p) = a \ \bigwedge \ actname(\overline{\alpha}_p) = a \ \bigwedge$

  $cons?(\overline{\mathcal{M}}', \overline{\alpha}'_p)$ True $\bigwedge$ $del?(\overline{\mathcal{M}}, \overline{\alpha}_p)$ True,

  s.t. $\overline{\mathcal{M}}' = \overline{\mathcal{M}}$,

  $\overline{\mathcal{A}}' = (\overline{\mathcal{A}} - \{\overline{\alpha}_p\} \bigcup \{\overline{\alpha}'_p\}$,

  $\overline{\mathcal{T}}' = \overline{\mathcal{T}}$, and

  $\overline{\mathcal{P}}' = \emptyset$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle \ I_j \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists [s]_a \in I_j$ s.t. $\overline{\alpha}_p \ \underset{act}{\longmapsto} \ [s]_a$.

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $cont(retrieveMsg(\overline{\alpha}_p, \overline{\mathcal{M}})) = s$

  (syntactic equality implies semantic equality)

- By definition of $\mathcal{F}_\mu$, $\mathcal{F}_\mu(\overline{\alpha}_p) \ = \mu \ = \ \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$,

  $\therefore$ By definition of $\mathcal{F}_\mu$ and $\mathcal{F}_{\hat{\delta}}$, $\overline{\alpha}_{cont} = \overline{\alpha}'_p$.

- By definition of actor/message synchronization,

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $\overline{\alpha}'_p \ \underset{act}{\longmapsto} \ [s]_a$

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 0$,

  $\therefore K_j^{AT} \overset{AT}{\Longrightarrow}{}^0 \ K_j^{AT}$.

- By definition of $\underset{cfg}{\longmapsto}$,

  $\therefore K_{i+1}^{\widehat{P^{act}}} \ \underset{cfg}{\longmapsto} \ K_j^{AT}$.

- Proved I.S. for case 1d.

Case 1e: replacement specification

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}', \overline{\mathcal{A}}', \overline{\mathcal{T}}', \overline{\mathcal{P}}', \sigma_{next}' \rangle$, where σ'_{next} is initially undefined, and by definition of replacement specification:

  $\exists \overline{\alpha}_r \in \overline{\mathcal{A}}'$ s.t. $\overline{\alpha}_r \notin \overline{\mathcal{A}}$,
  $\exists \overline{\alpha}'_p \in \overline{\mathcal{A}}'$, $\overline{\alpha}_p \in \overline{\mathcal{A}}$, and
  $actname(\overline{\alpha}_r) = a \ \bigwedge \ actname(\overline{\alpha}'_p) = a \ \bigwedge \ actname(\overline{\alpha}_p) = a \ \bigwedge$
  $cont(\overline{\alpha}_r) \neq cont(\overline{\alpha}'_p) \ \bigwedge \ cont(\overline{\alpha}_r) \neq cont(\overline{\alpha}_p) \ \bigwedge$
  $cont(\overline{\alpha}'_p) \neq cont(\overline{\alpha}_p)$,

  s.t. $\overline{\mathcal{M}}' = \overline{\mathcal{M}}$,
  $\overline{\mathcal{A}}' = (\overline{\mathcal{A}} - \{\overline{\alpha}_p\}) \bigcup \{\overline{\alpha}'_p, \overline{\alpha}_r\}$,
  $\overline{\mathcal{T}}' = \overline{\mathcal{T}}$, and
  $\overline{\mathcal{P}}' = \emptyset$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle \ I_j \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists [s]_a \in I_j$ s.t. $\overline{\alpha}_p \ \underset{act}{\longmapsto} \ [s]_a$
  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $cont(\overline{\alpha}_p) = s$.

- By definition of $\mathcal{F}_\mu$, $\mathcal{F}_\mu(\overline{\alpha}_p) \ = \mu \ = \ \langle \overline{\alpha}_{cont}, \overline{\mathcal{M}}_{new}, \overline{\mathcal{A}}_{new}, \overline{\mathcal{T}}_{new}, \overline{\mathcal{A}}_{repl} \rangle$.

  $\therefore$ By definition of $\mathcal{F}_\mu$ and $\mathcal{F}_{\hat{\delta}}$, $\overline{\alpha}_{cont} = \overline{\alpha}'_p$, and $\overline{\mathcal{A}}_{repl} = \{\overline{\alpha}_r\}$.

- By definition of Am, $Am([s]_a)$ yields $\{[s']_{(a)}, [t]_a\}$,

  where $s' = cont(\overline{\alpha}'_p)$ and $t = cont(\overline{\alpha}_r)$.
  Let $I_{j+1} = I_j \bigcup \{[s']_{(a)}, [t]_a\}$.
  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $\overline{\alpha}'_p \ \underset{act}{\longmapsto} \ [s']_{(a)}$ and $\overline{\alpha}_r \ \underset{act}{\longmapsto} \ [t]_a$.

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 1$,

  $\exists K_{j+1}^{AT}$ s.t. $K_j^{AT} \overset{AT}{\Longrightarrow}{}^1 \ K_{j+1}^{AT}$, where $K_{j+1}^{AT} = \left\langle\!\!\left\langle \ I_{j+1} \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$.

- By definition of $\underset{cfg}{\longmapsto}$,

  $\therefore K_{i+1}^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_{j+1}^{AT}$.

- Proved I.S. for case 1e.

Case 2: incoming external task

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}',\ \overline{\mathcal{A}}',\ \overline{\mathcal{T}}',\ \overline{\mathcal{P}}',\ \sigma_{next}{}' \rangle$, where σ'_{next} is initially undefined, and by definition of incoming external task:

  $\exists \overline{\tau}_r \in \overline{\mathcal{T}}'$ s.t. $\overline{\tau}_r \notin \overline{\mathcal{T}}$,

  $\bigwedge\ recip(\overline{\tau}_r) = a\ \bigwedge\ content(\overline{\tau}_r) = M$

  $\bigwedge\ \exists \overline{\alpha}_p \in \overline{\mathcal{A}}$, $\overline{\alpha}_p \in \overline{\mathcal{A}}'$ s.t. $actname(\overline{\alpha}_p) = a$,

  s.t. $\overline{\mathcal{M}}' = \overline{\mathcal{M}}$,

  $\overline{\mathcal{A}}' = \overline{\mathcal{A}}$,

  $\overline{\mathcal{T}}' = \overline{\mathcal{T}} \bigcup \{\overline{\tau}_r\}$, and

  $\overline{\mathcal{P}}' = \{\varepsilon\}$, where $\varepsilon = \langle \mathbf{E_S}, \overline{\tau}_r \rangle$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle\ I_j\ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists [s]_a \in I_j$ s.t. $\overline{\alpha}_p \underset{act}{\longmapsto} [s]_a$

  $\therefore$ By definition of $\underset{act}{\longmapsto}$, $cont(\overline{\alpha}_p) = s$.

- By definition of $\mathcal{F}_{\widehat{in}}$, $\mathcal{F}_{\hat{\delta}}$,

  task $\overline{\tau}_r$ is added to $K_{i+1}^{\widehat{P^{act}}}$ by an external entity.

  $\therefore \exists a \triangleleft M$ from the same external entity, s.t. $\overline{\tau}_r \underset{msg}{\longmapsto} a \triangleleft M$.

- Let $I_{j+1} = I_j \bigcup \{a \triangleleft M\}$, and let $\chi' = \chi \bigcup (acq(M) - \rho)$.

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 1$,

  $\exists K_{j+1}^{AT}$ s.t. $K_j^{AT} \overset{AT}{\Longrightarrow}{}^1 K_{j+1}^{AT}$, where $K_{j+1}^{AT} = \left\langle\!\!\left\langle\ I_{j+1}\ \right\rangle\!\!\right\rangle_{\chi'}^{\rho}$.

- By definition of $\underset{cfg}{\longmapsto}$, and I.H.

  $\therefore K_{i+1}^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_{j+1}^{AT}$, since only $\overline{\tau}_r$ is added to $K_{i+1}^{\widehat{P^{act}}}$, and $a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$.

  Conditions for actors and receptionists in $\underset{cfg}{\longmapsto}$ are unchanged, and $(acq(M) - \rho)$ adds only external actor names to χ, thus preserving the condition for external actors.

- Proved I.S. for case 2.

Case 3: outgoing external task

- Let $K_{i+1}^{\widehat{P^{act}}} = \sigma_{i+1}$ s.t. $\sigma_{next} = \sigma_{i+1}$ and $\sigma_{i+1} = \langle \overline{\mathcal{M}}', \overline{\mathcal{A}}', \overline{\mathcal{T}}', \overline{\mathcal{P}}', {\sigma_{next}}' \rangle$,

  where σ'_{next} is initially undefined, and by definition of outgoing external task:

  $$\exists \overline{\tau}_r \in \overline{\mathcal{T}} \text{ s.t. } \overline{\tau}_r \notin \overline{\mathcal{T}}',$$
  $$\bigwedge \ recip(\overline{\tau}_r) = a \ \bigwedge \ content(\overline{\tau}_r) = M$$
  $$\bigwedge \ \forall \overline{\alpha}_p \in \overline{\mathcal{A}}, \overline{\alpha}_p \in \overline{\mathcal{A}}'. \ actname(\overline{\alpha}_p) \neq a,$$

  s.t. $\overline{\mathcal{M}}' = \overline{\mathcal{M}}$,

  $\overline{\mathcal{A}}' = \overline{\mathcal{A}}$,

  $\overline{\mathcal{T}}' = \overline{\mathcal{T}} - \{\overline{\tau}_r\}$, and

  $\overline{\mathcal{P}}' = \emptyset$.

- By I.H. and $\underset{cfg}{\longmapsto}$, for $K_j^{AT} = \left\langle\!\!\left\langle \ I_j \ \right\rangle\!\!\right\rangle_{\chi}^{\rho}$,

  $\exists a \triangleleft M \in I_j$ s.t. $a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$.

- By definition of $\mathcal{F}_{\widehat{out}}$, $\mathcal{F}_{\hat{\delta}}$,

  task $\overline{\tau}_r$ is removed from $K_{i+1}^{\widehat{P^{act}}}$.

- Let $I_{j+1} = I_j - \{a \triangleleft M\}$, and let $\rho' = \rho \bigcup (acq(M) - \chi)$.

- By definition of $\overset{AT}{\Longrightarrow}$, for $n = 1$,

  $\exists K_{j+1}^{AT}$ s.t. $K_j^{AT} \overset{AT}{\Longrightarrow}{}^1 K_{j+1}^{AT}$, where $K_{j+1}^{AT} = \left\langle\!\!\left\langle \ I_{j+1} \ \right\rangle\!\!\right\rangle_{\chi}^{\rho'}$.

- By definition of $\underset{cfg}{\longmapsto}$, and I.H.

  $\therefore K_{i+1}^{\widehat{P^{act}}} \underset{cfg}{\longmapsto} K_{j+1}^{AT}$, since only $\overline{\tau}_r$ is removed from $K_{i+1}^{\widehat{P^{act}}}$,

  and $a \triangleleft M \underset{msg}{\longmapsto} \overline{\tau}_r$. Conditions for actors and external actor names in $\underset{cfg}{\longmapsto}$ are unchanged, and $(acq(M) - \chi)$ adds only internally defined actor names to ρ, thus preserving the condition for receptionists.

- Proved I.S. for case 3.

Since $\Longrightarrow$ and $\Longleftarrow$ hold for all respective cases, we conclude Theorem 1 is true.

CHAPTER 6

paraDOS Instantiated for Linda, Tuple Space

This chapter presents two operational semantics for Linda, and an equivalence proof between our semantics and the work by Jensen [Jen94]. Section 6.1 discusses the evolution of the two semantic versions of paraDOS for Linda, and definitions and notation that apply to both semantics. The first semantics describes how computation proceeds using functions defined in set-theoretic notation, similar to our approach in P^{act}. We describe the second operational semantics for Linda using the programming language Scheme. We present these two semantics in Sections 6.2 and 6.3, respectively. Section 6.4 contains the theorem and proof.

6.1 Instance Evolution and Definitions

Section 6.2 presents our original efforts instantiating paraDOS for Linda, and is self-contained with function descriptions and formal definitions. We conceived this set-theoretic semantics prior to distilling parameters for paraDOS, prior to considering how to represent composition in paraDOS, and thus prior to abstracting a set of message closures for $\overline{\mathcal{S}}$. The set-theoretic semantics lends itself to a more direct comparison with P^{act}. It is also instructive to compare the two operational semantics for Linda, since we derived the Scheme-based implementation from the set-theoretic description. The major difference between the two semantics, other than message closures, is that the Scheme-based semantics permitted us to discard the meaning structure used to accumulate multiple Linda processes' computational progress. Later, we augmented the Scheme semantics with message closures in consideration of composition. The equivalence proof in Section 6.4 refers to the Scheme-based semantics. We defer further discussion of composition until Chapter 8.

Let $\overline{\mathcal{S}}$ denote tuple space $\mathcal{S}$'s corresponding $\mathcal{P}^{\mathbf{TS}}$ model. It remains to define the structure of states σ within $\overline{\mathcal{S}}$, the transition function $\mathcal{F}_\delta$ of $\overline{\mathcal{S}}$, and what constitutes an observable event in $\overline{\mathcal{S}}$. We begin our discussion with the structure of σ. A state σ is represented by the 4-tuple $\langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where $\overline{\mathcal{A}}$ represents the multiset of active tuples, $\overline{\mathcal{T}}$ represents the multiset of passive tuples, $\overline{\mathcal{P}}$ represents the parallel event multiset, and σ_{next} is either *undefined*, or the state to which computation proceeds, as assigned by the transition function.

We introduce a mechanism to refer to specific tuples in a multiset of a state. To access members of the i^{th} state's multiset of active tuples, consider $\sigma_i = \langle \overline{\mathcal{A}}_i, \overline{\mathcal{T}}_i, \overline{\mathcal{P}}_i, \sigma_{i+1} \rangle$. Elements of $\overline{\mathcal{A}}_i$ can be ordered $1, 2, \ldots, |\overline{\mathcal{A}}_i|$; let $t_1, t_2, \ldots, t_{|\overline{\mathcal{A}}_i|}$ represent the corresponding tuples. The fields of a tuple t_j, for $1 \leq j \leq |\overline{\mathcal{A}}_i|$, can be projected as $t_j[k]$, for $1 \leq k \leq |t_j|$. See Figures 6.1 and 6.6 for the respective set-theoretic and Scheme-based domain specification of states, tuples, and fields.

The Scheme-based $\mathcal{P}^{\mathbf{TS}}$ semantics classifies the type of a tuple field as either active, pending, or passive. The set-theoretic semantics distinguishes only active and passive tuple field types. An active field is one that contains a Linda process making computational progress. A pending field contains a Linda process executing a synchronous primitive, but still waiting for a match. A passive field is one whose final value is already computed. Tuple `t` is active if it contains at least one active or pending field, otherwise `t` is passive. An active tuple becomes passive, and thus visible for matching in tuple space, when all of its originally active or pending fields become passive.

Multiple possible meanings of an individual Linda process's computation exist, when considered in the context of the multiple Linda processes that together comprise tuple space computation. Each state transition in $\mathcal{P}^{\mathbf{TS}}$ represents one of the possible cumulative meanings of the active or pending tuple fields making computational progress in that transition. We address these many possible individual and cumulative meanings when we describe the $\mathcal{P}^{\mathbf{TS}}$ transition function.

6.2 Set-theoretic Semantics for Linda

This section discusses the set-theoretic semantic functions that comprise $\mathcal{P}^{\mathbf{TS}}$. Figures 6.2 through 6.5 contain the corresponding algorithmic descriptions, not all of which are presented at the same level of detail. Specifically, we focus on the functions $\mathcal{G}$ (generate children), Lm (Linda meaning), and $\mathcal{F}_v$ (the view function), as they perform the interesting work for tuple space instantiation. Figure 6.1 contains the original domain specifications for the set-theoretic $\mathcal{P}^{\mathbf{TS}}$. For domains *tupleSet* and *parEventSet*, $S^{(tuple)}$ and $S^{(seqEvent)}$ are, respectively, multiset powersets of tuples and sequential events. As was the case for P^{act}, the set of communication closures $\overline{\Lambda}$ remains empty for the set-theoretic $\mathcal{P}^{\mathbf{TS}}$ specification.

Computation proceeds in $\mathcal{P}^{\mathbf{TS}}$ through invocation of transition function $\mathcal{F}_\delta$, shown in Figure 6.2, along with the generate meaning function *genMeaning* and the $\mathcal{P}^{\mathbf{TS}}$ meaning function $\mathcal{F}_\mu$. Function $\mathcal{F}_\delta$ traverses σ until it finds a state whose

Var	Domain	Domain Specification
$\overline{\mathcal{S}}$	*system*	$state \times closureSet \times viewSet$
σ	*state*	$tupleSet \times tupleSet \times parEventSet \times state$ | undefined
$\overline{\mathcal{A}}$, $\overline{\mathcal{T}}$	*tupleSet*	$S^{(tuple)}$
$\overline{\mathcal{P}}$	*parEventSet*	$S^{(seqEvent)}$
Lprocs	*LprocSet*	$P^{(int \times int)}$
μ	*meaning*	$tupleSet^7$
$\overline{\Lambda}$	*closureSet*	$\emptyset$
$\overline{\Upsilon}$	*viewSet*	$P^{(view)}$
υ	*view*	$list(ROPE)$
ρ	*ROPE*	$list(seqEvent)$
t, t_j, `template`	*tuple*	$list(field)$
$t_j[k]$	*field*	$fieldtype \times data$
ε	*seqEvent*	$etype \times tuple$
$\mathbf{E}_{type}$	*etype*	{`'Ecreated`, `'Ecopied`, `'Econsumed`, `'Egenerating`, `'Egenerated`}
$t_j[k]$.`type`	*fieldtype*	{'A', 'P'}
$t_j[k]$.`contents`	*data*	$beh \bigcup Base$
ψ	*beh*	continuation (unspecified)
	Base	base types (unspecified)
$f()$	*function*	functions (unspecified)

Figure 6.1: Set-theoretic $\mathcal{P}^{\mathbf{TS}}$ Domain Specification.

σ_{next} is *undefined*. Such a state is the current state of $\overline{\mathcal{S}}$, denoted σ_{cur}. Function $\mathcal{F}_\delta$ assigns to $\sigma_{cur}.\sigma_{next}$ the result of applying the generate children function $\mathcal{G}$ to σ_{cur}. Function $\mathcal{G}$ is shown separately in Figure 6.3. Applying $\mathcal{F}_\delta$ to σ elaborates the next computational state in the trace of $\overline{\mathcal{S}}$.

The function *genMeaning* constructs one possible composite meaning that results from multiple Linda processes making simultaneous computational progress within a shared tuple space. Function *genMeaning* utilizes the $\mathcal{P}^{\mathbf{TS}}$ meaning function $\mathcal{F}_\mu$, which in turn calls the Linda meaning function Lm. Function Lm is shown separately in Figure 6.4.

$$
\begin{array}{l}
\mathcal{F}_\delta : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{F}_\delta(\langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle) = \\
\quad\quad \text{if } \sigma_{next} \text{ undefined} \\
\quad\quad\quad \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \mathcal{G}(\langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle) \rangle \\
\quad\quad \text{else} \\
\quad\quad\quad \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \mathcal{F}_\delta(\sigma_{next}) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}_\mu : \mathit{tuple} \times \mathit{int} \times \mathit{meaning} \longrightarrow \mathit{meaning} \\
\quad \mathcal{F}_\mu(t_j, k, \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle) = \\
\quad\quad \mu \text{ where} \\
\quad\quad\quad \mu \in \{ \langle \overline{\mathcal{A}}', \overline{\mathcal{T}}', \overline{\mathcal{T}}'_{rd}, \overline{\mathcal{T}}'_{in}, \overline{\mathcal{T}}'_{out}, \overline{\mathcal{A}}'_{eval}, \overline{\mathcal{T}}'_{pass} \rangle \mid \\
\quad\quad\quad\quad\quad Lm(t_j, k, \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle) \\
\quad\quad\quad\quad\quad \text{yields } \langle \overline{\mathcal{A}}', \overline{\mathcal{T}}', \overline{\mathcal{T}}'_{rd}, \overline{\mathcal{T}}'_{in}, \overline{\mathcal{T}}'_{out}, \overline{\mathcal{A}}'_{eval}, \overline{\mathcal{T}}'_{pass} \rangle \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{genMeaning} : \mathit{LprocSet} \times \mathit{meaning} \longrightarrow \mathit{meaning} \\
\quad \mathit{genMeaning}(\text{Lprocs}, \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle) = \\
\quad\quad \text{if Lprocs empty} \\
\quad\quad\quad \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\quad\quad \text{else} \\
\quad\quad\quad \text{let } \langle j, k \rangle \in \text{ Lprocs} \\
\quad\quad\quad\quad \text{in } \mathit{genMeaning}((\text{Lprocs} - \langle j, k \rangle), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathcal{F}_\mu(t_j, k, \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle))
\end{array}
$$

Figure 6.2: Transition and meaning functions.

The generate children function $\mathcal{G}$ deserves closer attention, since it specifies the next state in σ elaborated by transition function $\mathcal{F}_\delta$. The behavior of $\mathcal{G}$ describes event generation for $\mathcal{P}^{\mathbf{TS}}$. First, $\mathcal{G}$ selects a random subset of active Linda processes to make computational progress. Next, $\mathcal{G}$ passes those Linda processes to *genMeaning*, which returns a composite meaning, in turn assigned to μ. Finally, $\mathcal{G}$ uses the elements of μ to construct its own return state tuple. Specifically, $\mathcal{G}$ adds to the $\overline{\mathcal{A}}'$ multiset the updated multiset of active tuples, and any new active tuples generated as a result of *eval* primitives. It adds to the $\overline{\mathcal{T}}'$ multiset the updated multiset of passive tuples, and any new passive tuples generated as a result of *out* primitives. $\mathcal{G}$ builds the $\overline{\mathcal{P}}'$ multiset from the events

$$
\begin{array}{l}
\mathcal{G} : \mathit{state} \longrightarrow \mathit{state} \\
\quad \mathcal{G}(\langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle) = \\
\qquad \langle \overline{\mathcal{A}}', \overline{\mathcal{T}}', \overline{\mathcal{P}}', \sigma_{next}' \rangle \text{ where} \\
\qquad\quad \text{let Lprocs} = \{\langle j, k \rangle \mid (1 \leq j \leq |\overline{\mathcal{A}}|) \bigwedge \ t_j \in \overline{\mathcal{A}} \ \bigwedge (1 \leq k \leq |t_j|) \ \bigwedge \\
\qquad\qquad\qquad\qquad\qquad t_j[k].\mathtt{type} = \text{'A'}\} \\
\qquad\quad \text{in let randsub} \subseteq \text{Lprocs} \\
\qquad\quad \text{in let } \mu = \mathit{genMeaning}(\text{randsub}, \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle) \\
\qquad\qquad \text{where } \mu = \langle \overline{\mathcal{A}}_{\mu}, \overline{\mathcal{T}}_{\mu}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\qquad \text{in} \\
\qquad\qquad\quad \overline{\mathcal{A}}' = \overline{\mathcal{A}}_{\mu} \bigcup \overline{\mathcal{A}}_{eval} \\
\qquad\qquad\quad \overline{\mathcal{T}}' = \overline{\mathcal{T}}_{\mu} \bigcup \overline{\mathcal{T}}_{out} \\
\qquad\qquad\quad \overline{\mathcal{P}}' = \{\langle \mathtt{'Ecreated}, t \rangle \mid t \in \overline{\mathcal{T}}_{out}\} \bigcup \\
\qquad\qquad\qquad\quad \{\langle \mathtt{'Ecopied}, t \rangle \mid t \in \overline{\mathcal{T}}_{rd}\} \bigcup \{\langle \mathtt{'Econsumed}, t \rangle \mid t \in \overline{\mathcal{T}}_{in}\} \bigcup \\
\qquad\qquad\qquad\quad \{\langle \mathtt{'Egenerating}, t \rangle \mid t \in \overline{\mathcal{A}}_{eval}\} \bigcup \\
\qquad\qquad\qquad\quad \{\langle \mathtt{'Egenerated}, t \rangle \mid t \in \overline{\mathcal{T}}_{pass}\} \\
\qquad\qquad\quad \sigma_{next}' \text{ is undefined}
\end{array}
$$

Figure 6.3: The generate children function.

it discerns from the contents of meaning structure μ. $\overline{\mathcal{P}}'$ is a 5-way union of multisets; one for each event type abstraction in $\mathcal{P}^{\mathbf{TS}}$, and not coincidentally, each event type abstracted for $\mathcal{P}^{\mathbf{TS}}$ corresponds to its own element of μ. The return state's σ_{cur} is undefined at return time, indicating it has not yet been elaborated. Next, discussion of the Lm function reveals how the multisets of μ are assigned their member tuples.

The Linda meaning function Lm, shown in Figure 6.4, handles three general cases. Either process $t_j[k]$ makes computational progress involving no Linda primitives, but still has remaining computation; process $t_j[k]$ makes computational progress involving no Linda primitives, and replaces itself with a typed return value; or process $t_j[k]$ makes computational progress, the last part of which is a Linda primitive.

$$
\begin{array}{l}
Lm : tuple \times int \times meaning \longrightarrow meaning \\
\quad Lm(t_j, k, \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle) = \\
\qquad \text{let } t'_j = tupleUpdate(t_j, k, rand \circ Lm_{comp}) \\
\qquad \text{in} \\
\qquad\quad \text{let } \overline{\mathcal{A}}' = \begin{cases} (\overline{\mathcal{A}} - \{t_j\}) \bigcup \{t_j{}'\} & \text{if } \exists \ell, 1 \leq \ell \leq |t_j|, \text{ s.t. } t_j{}'[\ell].\texttt{type} = '\text{A}', \\ \overline{\mathcal{A}} - \{t_j\} & \text{otherwise.} \end{cases} \\
\qquad\quad\quad \overline{\mathcal{T}}'_{pass} = \begin{cases} \overline{\mathcal{T}}_{pass} & \text{if } \exists \ell, 1 \leq \ell \leq |t_j|, \text{ s.t. } t_j{}'[\ell].\texttt{type} = '\text{A}', \\ \overline{\mathcal{T}}_{pass} \bigcup \{t_j{}'\} & \text{otherwise.} \end{cases} \\
\qquad\quad \text{in} \\
\qquad\quad\quad \text{if } (t'_j[k].\texttt{type} = '\text{A}') \bigwedge (nextcomp(t'_j[k]) \text{ is a Linda primitive}) \\
\qquad\quad\quad \bigwedge \text{ (randomly choose to proceed)} \\
\qquad\quad\quad\quad \text{let } t''_j = tupleUpdate(t'_j, k, nextcomp()), \\
\qquad\quad\quad\quad\quad \text{and } LindaPrim, \texttt{template} \text{ be from } nextcomp(t_j{}'[k]) \\
\qquad\quad\quad\quad \text{in} \\
\qquad\quad\quad\quad\quad \text{let } \overline{\mathcal{A}}'' = (\overline{\mathcal{A}}' - \{t'_j\}) \bigcup \{t''_j\}, \\
\qquad\quad\quad\quad\quad\quad t = \begin{cases} tupleMatch(\texttt{template}, \overline{\mathcal{T}}) & \text{if } LindaPrim = rd \bigvee in, \\ tcopy(\texttt{template}) & \text{otherwise.} \end{cases} \\
\qquad\quad\quad\quad\quad \text{in Cases for } LindaPrim = \\
\qquad\quad\quad\quad\quad\quad rd: \ \text{if } (t = \text{Fail}) \ \langle \overline{\mathcal{A}}', \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\quad\quad\quad\quad\quad\quad\ \ \text{else } \langle \overline{\mathcal{A}}'', \overline{\mathcal{T}}, (\overline{\mathcal{T}}_{rd} \bigcup \{t\}), \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\quad\quad\quad\quad\quad in: \ \text{if } (t = \text{Fail}) \ \langle \overline{\mathcal{A}}', \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\quad\quad\quad\quad\quad\quad\ \ \text{else } \langle \overline{\mathcal{A}}'', (\overline{\mathcal{T}} - \{t\}), \overline{\mathcal{T}}_{rd}, (\overline{\mathcal{T}}_{in} \bigcup \{t\}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\quad\quad\quad\quad\quad out: \langle \overline{\mathcal{A}}'', \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, (\overline{\mathcal{T}}_{out} \bigcup \{t\}), \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\quad\quad\quad\quad\quad eval: \langle \overline{\mathcal{A}}'', \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, (\overline{\mathcal{A}}_{eval} \bigcup \{t\}), \overline{\mathcal{T}}_{pass} \rangle \\
\qquad\quad\quad \text{else} \\
\qquad\quad\quad\quad \langle \overline{\mathcal{A}}', \overline{\mathcal{T}}, \overline{\mathcal{T}}_{rd}, \overline{\mathcal{T}}_{in}, \overline{\mathcal{T}}_{out}, \overline{\mathcal{A}}_{eval}, \overline{\mathcal{T}}'_{pass} \rangle
\end{array}
$$

Figure 6.4: The Linda meaning function.

Lm receives input parameters indicating $t_j[k]$ as the tuple and field containing the Linda process to make computational progress, and a cumulative meaning structure reflecting the computations of Linda processes previously passed to Lm in the current $\mathcal{P}^{\mathbf{TS}}$ transition. First, tuple t'_j reflects the update of t_j, with $t'_j[k]$

assigned its new field value, randomly chosen from the set of all possible new field values.

Next, if t'_j is passive, t_j is removed from the multiset of active tuples and t'_j is added to the multiset of newly passive tuples. Otherwise t'_j replaces t_j in the multiset of active tuples, and the multiset of newly passive tuples is unchanged. To this point, only internal computational progress is possible. Lm might choose at random to return a meaning structure reflecting the changes thus far, or be forced to do so if either t'_j is passive or t'_j's next computation is not a Linda primitive.

If Lm proceeds, tuple t''_j reflects the update of t'_j, with $t''_j[k]$ assigned its new field value, which must be the result of the new continuation of its Linda process, since, by definition it computed no additional internal computation and precisely one Linda primitive since t'_j was produced. Then Lm replaces t'_j with t''_j in the multiset of active tuples. If the Linda primitive is either *rd* or *in*, Lm attempts to find a matching tuple t for the operation; otherwise Lm copies the *out* or *eval* template to t.

Finally, Lm considers the cases for the Linda primitive to determine the return meaning structure. If a *rd* or *in* was attempted and the tuple match failed, a meaning structure is returned that reflects only t'_j's impact on the multiset of active tasks (i.e. the next time this Linda process is chosen to make computational progress, it will retry the same *rd* or *in* operation). If the match was successful, in the case of a *rd*, t is added to the $\overline{\mathcal{T}}_{rd}$ multiset; and in the case of *in*, t is both removed from the state's tuple space, $\overline{\mathcal{T}}$, and added to $\overline{\mathcal{T}}_{in}$. If the Linda primitive was an *out*, t is added to the $\overline{\mathcal{T}}_{out}$ multiset. Otherwise, the Linda primitive must have been an *eval*, in which case t is added to the $\overline{\mathcal{A}}_{eval}$ multiset.

$Lm_{comp} : \mathit{field} \longrightarrow P^{(\mathit{field})}$

$$
\begin{array}{l}
\mathit{tupleMatch} : \mathit{tuple} \times \mathit{tupleSet} \longrightarrow \mathit{tuple} \bigcup \text{Fail} \\
\quad \mathit{tupleMatch}(\texttt{template}, \overline{\mathcal{T}}) = \\
\quad\quad \text{if } \exists t_m \in \overline{\mathcal{T}} \ \text{ s.t. } \mathit{match}(\texttt{template}, t_m) \\
\quad\quad\quad t_m \\
\quad\quad \text{else} \\
\quad\quad\quad \text{Fail // (i.e. blocked...)}
\end{array}
$$

$$
\begin{array}{l}
\mathit{tupleUpdate} : \mathit{tuple} \times \mathit{int} \times \mathit{function} \longrightarrow \mathit{tuple} \\
\quad \mathit{tupleUpdate}(t_j, k, f()) = \\
\quad\quad t'_j, \text{ where} \\
\quad\quad\quad \forall \ell, 1 \leq \ell \leq |t_j| \\
\quad\quad\quad\quad t'_j[\ell] = \begin{cases} f(t_j[\ell]) & \text{if } \ell = k, \\ t_j[\ell] & \text{otherwise.} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathit{tcopy} : \mathit{tuple} \longrightarrow \mathit{tuple} \\
\quad \mathit{tcopy}(\texttt{template}) = \\
\quad\quad t, \text{ where} \\
\quad\quad\quad \forall k, 1 \leq k \leq |\texttt{template}| \\
\quad\quad\quad\quad t[k].\texttt{contents} = \texttt{template}[k], \\
\quad\quad\quad\quad t[k].\texttt{type} = \begin{cases} '\text{P}' & \text{if } \texttt{template}[k] \text{ passive,} \\ '\text{A}' & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 6.5: Functions used by Lm

We show the functions invoked by Lm in Figure 6.5. Briefly, we assume the existence of function Lm_{comp} to handle the details of individual process computation. This is a reasonable assumption since the intent of Linda is to augment existing programming languages with primitive tuple space operations. The meaning of an individual Linda process's computation, as conveyed by Lm_{comp}, derives from the well-understood semantics of its underlying programming language. Additionally, we define a tuple matching function, *tupleMatch*, to describe the behavior of the synchronous Linda coordination primitives. Furthermore, *tupleMatch* assumes the existence of a $match()$ predicate. Finally, we define two additional helper functions, *tupleUpdate* and *tcopy*, to handle the details of updating and copying tuples.

Function Lm_{comp} may make computational progress on its input parameter, a tuple field. The computational progress may be up to, but not including, a Linda primitive function. Function Lm_{comp} returns an updated tuple field containing a possibly updated continuation, and a possibly updated `type` indicator if the tuple field changed from active to passive as a result of its computational progress.

6.3 Scheme-based Semantics for Linda

The Scheme-based $\mathcal{P}^{\mathbf{TS}}$ model extends the syntax of the Linda primitives with a tuple space handle prefix. This handle can refer to the tuple space in which the issuing Linda process resides (i.e. "self"), or it can be a tuple space handle acquired by the issuing Linda process during the course of computation. The use of a tuple space handle is consistent with commercial implementations of tuple space. The existence of this handle is explained when we discuss tuple space composition in Section 8.3. Tuple space handles are nothing more than values, and may thus reside as fields within tuples in tuple space. In the absense of composition, acquiring a tuple space handle h reduces to matching and copying a tuple that contains h as one of its values.

We present the Scheme-based semantics of $\mathcal{P}^{\mathbf{TS}}$ in detail in this section. Not all functions are discussed at the same level of detail. We give an overview of the transition function and the view function, focusing on important aspects of tuple space computation and view generation. Figure 6.6 contains the domain specification for the version of $\mathcal{P}^{\mathbf{TS}}$ described in this section.

Var	Domain	Domain Specification
$\overline{\mathcal{S}}$	*system*	$state \times closureSet \times viewSet$
`sigma` (σ)	*state*	$tupleSet \times tupleSet \times parEventSet \times state$ | undefined
`LBar` ($\overline{\Lambda}$)	*closureSet*	$S^{(closure)}$
$\overline{\Upsilon}$	*viewSet*	$P^{(view)}$
`state-LBar`, $\langle\sigma, \overline{\Lambda}\rangle$	*SCSPair*	$state \times closureSet$
`ABar` ($\overline{\mathcal{A}}$), `TBar` ($\overline{\mathcal{T}}$)	*tupleSet*	$S^{(tuple)}$
`PBar` ($\overline{\mathcal{P}}$)	*parEventSet*	$S^{(seqEvent)}$
`LProcs`	*LprocSet*	$P^{(int \times int)}$
`t`, `tsubj`, `template`	*tuple*	*list*(*field*)
	field	$fieldtype \times data$
	seqEvent	$etype \times tuple$
	etype	{`'Ecreated`, `'Ecopied`, `'Econsumed`, `'Egenerating`, `'Egenerated`}
field.`type`	*fieldtype*	{'Active, 'Pending, 'Passive}
field.`contents`	*data*	$beh \bigcup Base \bigcup Formal$
	beh	continuation (unspecified)
	Base	base types (unspecified)
	Formal	?*Base*
`closure`, `lambda`, λ	*closure*	$asynchCl \bigcup synchCl \bigcup sendCl \bigcup matchCl \bigcup reactCl \bigcup asynchLPrim$
	asynchCl	{"`send(handle, delay(lambda))`" | `handle` denotes tuple space $\bigwedge$ `lambda` $\in$ *asynchLPrim* }
	synchCl	{"`send(handle, delay(lambda))`" | `handle` denotes tuple space $\bigwedge$ `lambda` $\in$ *sendCl* }
	sendCl	{"`send(self, force(lambda))`" | `self` denotes tuple space $\bigwedge$ `lambda` $\in$ *matchCl* }
	matchCl	{"`(let t = force(lambda) in delay(lambda2))`" | `lambda` $\in$ *synchLPrim* $\bigwedge$ `lambda2` $\in$ *reactCl* }
	reactCl	{ "`react(j,k,t)`" }
	asynchLPrim	{`eval(template)`, `out(template)`}
	synchLPrim	{`rd(template)`, `in(template)`}
`upsilon`, υ	*view*	*list*(*ROPE*)
`rho`, ρ	*ROPE*	*list*(*seqEvent*)

Figure 6.6: $\mathcal{P}^{\mathbf{TS}}$ Domain Specification.

Computation proceeds in $\mathcal{P}^{\mathbf{TS}}$ through invocation of the transition function `F-delta`. `F-delta` takes a pair of arguments, tree σ and the set of communication closures $\overline{\Lambda}$, and elaborates the next state in the trace of σ. There are two phases in a $\mathcal{P}^{\mathbf{TS}}$ transition: the inter-process phase and the intra-process phase. The inter-process phase, or communication phase, specified by `F-LambdaBar`, concerns the computational progress of the Linda primitives in $\overline{\Lambda}$. The intra-process phase, specified by `G`, concerns the computational progress of active Linda processes within σ_{cur}. `F-delta` returns the pair containing the elaborated tree σ_{new} and the resulting new set of communication closures $\overline{\Lambda}_{new}$.

During the first phase of a $\mathcal{P}^{\mathbf{TS}}$ transition, function `F-LambdaBar` chooses a random subset of communication closures from $\overline{\Lambda}$ to attempt to reduce. In $\mathcal{P}^{\mathbf{TS}}$, each communication closure represents the computational progress of an issued Linda primitive. The domain specification for the different closure forms is included in Figure 6.6. From the perspective external to `F-LambdaBar`, these closures make computational progress in parallel. Linda primitives are scheduled via a randomly ordered list to model the nondeterminism of race conditions and the satisfaction of tuple matching operations among competing synchronous requests. `F-LambdaBar` returns a σ-$\overline{\Lambda}$ pair representing one possible result of reducing the communication closures.

To better understand the functions that reduce closures in $\overline{\Lambda}$, we take a moment to examine more closely the *closure* domain from Figure 6.6. The closure domains that form *closure* characterize the stages through which communication activity proceeds in tuple space. The form of closure domains *asynchCl*, *synchCl*, and *sendCl* specifies that a lambda expression λ be sent to a designated $\overline{\Lambda}$ set. Closures from domains *asynchCl* and *synchCl* explicitly delay the evaluation of λ; domain *sendCl* explicitly forces the evaluation of λ. The designation of the

$\overline{\Lambda}$ set is through a tuple space handle. The notion of sending a closure, and the notion of tuple space handles, both derive from our ongoing research in tuple space composition. The processing of the `send` closure results in the set union of the $\overline{\Lambda}$ designated by `handle` and the singleton set containing element λ.

Functions `reduce-out` and `reduce-eval` both take an asynchronous communication closure and a σ-$\overline{\Lambda}$ pair as arguments, and return a σ-$\overline{\Lambda}$ pair. The `reduce-out` function adds a passive tuple to tuple space, and generates event `'Ecreated`. Similarly, `reduce-eval` adds an active tuple to tuple space, and generates event `'Egenerating`.

Function `reduce-send` returns an updated σ-$\overline{\Lambda}$ pair. In the case of delayed evaluation, `reduce-send` adds the send argument of λ to $\overline{\Lambda}$. Otherwise, evaluation of the send argument of λ is forced, and `reduce-send` attempts to reduce the let expression containing a synchronous Linda primitive. The let expression fails to reduce if there is no match in tuple space for the underlying `rd( )`or `in( )`operation's template. If the let expression can't be evaluated, `reduce-send` adds λ back to $\overline{\Lambda}$. Adding λ back to $\overline{\Lambda}$ permits future reduction attempts. Otherwise, the let expression reduces, `reduce-send` adds the new closure to $\overline{\Lambda}$, and σ, upon return, reflects the reduced let expression (for example, a tuple might have been removed from tuple space).

Functions `reduce-rd` and `reduce-in` both take a synchronous communication closure and a σ-$\overline{\Lambda}$ pair as arguments, and return either a tuple-state pair, or null. Both functions attempt to find a matching tuple in tuple space, and if unsuccessful, return null. If a match exists, `reduce-rd` returns a copy of the matching tuple, and generates event `'Ecopied`. Similarly, `reduce-in` returns a copy of matching tuple `t`, but also removes `t` from tuple space, while generating event `'Econsumed`.

The reactivate form of a communication closure specifies which field of which tuple contains a pending Linda process that is to be reactivated. Specifically, the `reduce-react` function updates `tsubj[k]` to make it an active Linda process, and fills its evaluation context with redex `t`. `reduce-react` is applied to a closure and a σ-$\overline{\Lambda}$ pair, where the closure contains `j`, `k`, and `t`. The σ-$\overline{\Lambda}$ pair returned by `reduce-react` contains the updated tuple.

During the second phase of a $\mathcal{P}^{\mathbf{TS}}$ transition, function `G` chooses a random subset of active Linda processes to make computational progress. From the perspective external to `F-LambdaBar`, these processes make computational progress in parallel. Internal to `G`, Linda processes are scheduled via the `genMeaning` function. The sequence doesn't matter, since during this intra-process phase of transition, no tuple space interactions occur. `G` returns a σ-$\overline{\Lambda}$ pair representing one possible cumulative meaning of the random subset of active Linda processes making computational progress.

A closer look at `genMeaning` is in order. Within a PDS, in general, it is possible for individual processes to make simultaneous computational progress at independent, variable rates. Thus, for $\mathcal{P}^{\mathbf{TS}}$, it is incumbent upon `genMeaning` to be capable of reflecting all possible combinations of computational progress among a list of Linda processes in the σ-$\overline{\Lambda}$ pair it returns. With the help of `F-mu`, `genMeaning` satisfies this requirement. For each Linda process, `F-mu` randomly chooses a meaning from the set of all possible meanings `Lm` could return; *i.e.* each process proceeds for some random amount of its total potential computational progress.

Function `Lm` is the high-level Linda meaning function for a process $t_j[k]$ in σ-$\overline{\Lambda}$. `Lm` handles three general cases. Either process $t_j[k]$ makes computational progress involving no Linda primitives, but still has remaining computation; process $t_j[k]$

makes computational progress involving no Linda primitives, and replaces itself with a typed return value; or process $t_j[k]$ makes computational progress, the last part of which is a Linda primitive. `Lm` assumes the existence of helper function `Lm-comp` to return all possible meanings of internal Linda process computation (that is, up to, but not including, a Linda primitive function). A random choice determines how $t_j[k]$ gets updated. In the case of the final active process within t_j becoming passive, `Lm` moves t_j from the set of active tuples to the set of passive tuples, and generates event `'Egenerated`.

In the case where $t_j[k]$'s computational progress includes a Linda primitive, function `Lm-prim` finishes the work `Lm` started. The two main cases of Linda primitives are asynchronous and synchronous. In either case, `Lm-prim` constructs the appropriate closure forms and adds the closure containing the primitive request to $\overline{\Lambda}$. In the case of the synchronous primitive, `Lm-prim` also changes $t_j[k]$ from active to pending.

The careful reader may question the need for a double choice of meanings among `Lm` and `F-mu`, for a given Linda process $t_j[k]$. Briefly, `Lm` selects a random meaning for $t_j[k]$; `F-mu` constructs the set of all possible meanings that `Lm` could return for $t_j[k]$, only to select from *this* set a random meaning for $t_j[k]$. Clearly, we could have structured a single random choice; but not doing so permits us to isolate and investigate different scheduling policies and protocols. For each transition, the number of possible next states is combinatorially large. Recall that `Lm` and `F-mu` are part of the function that generates children, one of which the transition function chooses to elaborate, in lazy tree σ. Each random choice the transition function makes prunes subsets of possible next states, until one remaining state is finally elaborated. Since `Lm-comp` is a helper function, the double choice of meanings emphasizes the possibilities for a single Linda process, and is consistent with the other random choices made during transition.

This concludes our description of the Scheme functions associated with transition in $\mathcal{P}^{\mathbf{TS}}$. The functional nature of Scheme gives a precise and elegant description of the operational semantics for Linda and tuple space. Equally precise and elegant is the Scheme implementation of the $\mathcal{P}^{\mathbf{TS}}$ view relation. Functions `F-view` and `more-ropes`, are equivalent instantiations of the the view relation defined in Chapter 4. The transition and view relations together allow us to reason about all possible behaviors of a distributed system's computation, and all possible views of each of those behaviors. Thus we have a powerful tool for identifying and reasoning about properties of distributed computation.

```
; Scheme function description of paraDOS instantiated for Linda
; (with support for tuple space composition)
;
; Lambda closure forms:
; --------------------
;
; Linda primitive cases:
;     1.  handle.rd(template)
;     2.  handle.in(template)
;     3.  handle.out(template)
;     4.  handle.eval(template)
; where
;     "handle" can be "self", "parent", or an acquired TS handle;
;     unqualified Linda primitives imply the handle "self"; and
;     handles other than "self" imply composition
;
; Cases 1 and 2:  (synchronous primitives)
;     lambda1 = send(handle, delay(lambda2))
;     lambda2 = send(self, force(lambda3))
;     lambda3 = (let t = force(lambda4) in delay(react(j,k,t)))
;     lambda4 = rd(template), or //case 1
;               in(template)     //case 2
```

```
;
; Cases 3 and 4:  (asynchronous primitives)
;     lambda1 = send(handle, delay(lambda2))
;     lambda2 = out(template), or //case 3
;               eval(template)    //case 4
;
; where
;     send(handle, lambda) is defined as the set union of
;     TS handle's LambdaBar set with the singleton set
;     containing lambda.


; Transition Function
; Summary:  Returns a state-LBar pair (list).
; This is how computation proceeds.
(define F-delta
   (lambda (state-LBar)
      (let ((sigma (get-state state-LBar))
            (LBar (get-LBar state-LBar)))
         (let ((sigmaCur (get-cur-state sigma)))
            (let ((new-state-LBar (G (F-LambdaBar
                        (list sigmaCur LBar)))))
               (let ((newsigma (get-state new-state-LBar))
                     (newLBar (get-LBar new-state-LBar)))
                  (list (elaborate-sigma sigma newsigma)
                        (newLBar))))))))


; get current state
; Summary:  helper function called by F-delta; traverses
; computational history to last elaborated state.
(define get-cur-state
   (lambda (sigma)
      (let ((next-sigma (get-next-state sigma)))
         (if (null?  next-sigma)
            sigma
            (get-cur-state next-sigma)))))
```

```
; elaborate sigma
; Summary:  helper function called by F-delta; elaborates
; next state in computational history with newsigma.
(define elaborate-sigma
   (lambda (sigma newsigma)
      (let ((Abar (get-Abar sigma))
            (Tbar (get-Tbar sigma))
            (Pbar (get-Pbar sigma))
            (next-sigma (get-next-state sigma)))
         (if (null?  next-sigma)
            (make-state Abar Tbar Pbar newsigma)
            (make-state Abar Tbar Pbar
               (elaborate-sigma
                  next-sigma newsigma))))))
```

```
; F-LambdaBar
; Summary:  Returns a state-LBar pair.  Selects random subset
; of closures from LambdaBar set.  Invokes reduce-all to do the
; work, passing in a randomly-ordered list of closures from
; subset selected, and an initialized state-LBar pair.  The
; state from state-LBar consists of the multisets of active
; and passive tuples from input state sigma.  The LBar element
; of state-LBar is the set difference of itself and the random
; subset of closures selected.
(define F-LambdaBar
   (lambda (state-LBar)
      (let ((sigma (get-state state-LBar))
            (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
               (Tbar (get-Tbar sigma))
               (randclosures
                  (get-rand-subset LBar)))
            (reduce-all
               (as-list randclosures)
               (list (make-state Abar Tbar '() '())
                  (set-diff LambdaBar randclosures)))))))
```

```
; reduce-all
; Summary:  Returns a state-LBar pair.  Accumulates the effects
; of applying the closures to the state in state-LBar.  Closures
; that couldn't reduce are added back to LBar in state-LBar.
; This function farms out the work, one closure at a time, to
; function reduce-1.
(define reduce-all
   (lambda (closures state-LBar)
      (if (null?  closures)
         (state-LBar)
         (reduce-all (cdr closures)
               (reduce-1 (car closures) state-LBar)))))
```

```
; reduce-1
; Summary:  Returns a state-LBar pair.  The outer-most function
; of closures in an LBar set are one of send(), reactivate(),
; or one of the asynchronous Linda primitives, out() and
; eval().  This function farms out the work accordingly.
(define reduce-1
   (lambda (closure state-LBar)
      (cond
         ((out?  closure)
            (reduce-out closure state-LBar))
         ((eval?  closure)
            (reduce-eval closure state-LBar))
         ((send?  closure)
            (reduce-send closure state-LBar))
         ((react?  closure)
            (reduce-react closure state-LBar)))))
```

```
; reduce-out
; Summary:  returns a new state-LBar pair.  The state element
; of state-LBar results from applying the Linda primitive
; out(template) to the input state.  Specifically, a new
; tuple is added to the new state's Tbar set.  Also, the
; new event Ecreated for the new tuple is added to the PBar
; set of the new state.  LBar is unchanged.
(define reduce-out
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
               (t (get-template closure)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((newTbar (union Tbar (singleton t)))
                     (newPbar (union Pbar (singleton
                              (make-event 'Ecreated t)))))
               (list (make-state
                              Abar newTbar newPbar '())
                        (get-LBar state-LBar)))))))


; reduce-eval
; Summary:  returns a new state-LBar pair.  Similar to reduce-out,
; a new active tuple is added to the new state's Abar set.  The
; corresponding event Egenerating for the new tuple is added to
; the PBar set of the new state.  LBar is unchanged.
(define reduce-eval
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
               (t (get-template closure)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((newAbar (union Abar (singleton t)))
                     (newPbar (union Pbar (singleton
                              (make-event 'Egenerating t)))))
               (list (make-state
                              newAbar Tbar newPbar '())
                        (get-LBar state-LBar)))))))
```

```
; reduce-react
; Summary:  returns a new state-LBar pair.  The reactivate closure
; specifies that within the ABar set of the state contained
; in the state-LBar pair, the k-th field of the j-th tuple
; is the process to be made active.  Part of the activation
; of this process includes the binding of tuple t to the "rd"
; or "in" call in the continuation:  the point which the process
; was originally suspended!
(define reduce-react
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
            (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
               (Tbar (get-Tbar sigma))
               (Pbar (get-Pbar sigma)))
            (let ((tuple-j (get-tuple Abar
                  (get-j closure))))
               (let ((field-k (get-field tuple-j
                     (get-k closure))))
                  (let ((new-field-k
                        (set-field-type
                        (bind field-k (get-t closure))
                        'Active)))
                     (let ((new-tuple-j
                           (add-field (remove-field
                           tuple-j field-k) new-field-k)))
                        (let ((newAbar (union
                              (set-diff Abar
                              (singleton tuple-j))
                              (singleton new-tuple-j))))
                           (list (make-state
                                       newAbar Tbar Pbar '())
                                    LBar))))))))))
```

```
; reduce-send (without TS composition)
; Summary:  returns a new state-LBar pair.  If the closure
; expression to be sent is delayed, strip the delay() and "send"
; by adding to LBar set.  Otherwise, closure is a forced "let"
; expression.  Farm off to reduce-let function.  If reduce-let
; fails, then reduce-send fails, and the original closure is
; added to returned state-LBar's set of closures (where
; state-LBar's state is unchanged).  If reduce-let was
; successful, the let expression bound a tuple into it's
; delayed subexpression (reactivate).  reduce-send then returns
; the new state-LBar pair, consisting of the subsequent new state
; and the reduced closure in LBar.
(define reduce-send
   (lambda (closure state-LBar)
      (let ((send-arg (get-send-arg closure)))
         (if (delayed?  send-arg)
            (let ((LBar1 (union (cadr state-LBar)
                     (singleton (strip-delay send-arg)))))
               (list (car state-LBar) LBar1))
            ;else forced
            (let ((closure-state
               (reduce-let (strip-force send-arg)
                     state-LBar)))
                  (if (null?  closure-state)
                     ;reduce failed
                     (list (car state-LBar)
                        (union (cadr state-LBar)
                           (singleton closure)))
                     ;else it reduced!
                     (let ((LBar1 (union (cadr state-LBar)
                           (car closure-state))))
                        (list (cadr closure-state)
                           LBar1))))))))
```

```
; reduce-let
; Summary:  returns a closure-state pair.  The closure part is a
; possibly reduced let expression, and a possibly modified state.
; Reduction depends on the success or failure of the forced
; Linda primitives rd or in.  A reduced closure consists of
; binding the result of the rd or in to the delayed part of
; the let closure.  The work of reducing the rd or in is farmed
; out to corresponding functions.
(define reduce-let
   (lambda (closure state-LBar)
      (let ((Lprim (get-forced closure))
               (react (get-delayed closure)))
         (let ((tuple-state
                  (if (rd?  Lprim)
                        (reduce-rd closure state-LBar)
                        (reduce-in closure state-LBar))))
            (if (null?  tuple-state)
               '() ;prim failed
               (let ((bound-closure
                        (bind (car tuple-state) react))
                     (newstate (cadr tuple-state)))
                  (list bound-closure newstate)))))))
```

```
; reduce-rd
; Summary:  returns a tuple-state pair.  Farms out matching work
; to exists?  function.  If successful, tuple part of tuple-state
; contains matching tuple t, and state part of tuple-state
; contains new event 'Ecopied in its Pbar set.
(define reduce-rd
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
               (template (get-template closure)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((f ((lambda t)
                     (match?  template t))))
               (let ((t (exists?  Tbar f)))
                  (if (null?  t)
                     ('())
                     (let ((newPbar (union Pbar
                                 (make-event 'Ecopied t))))
                        (let ((newsigma (make-state
                                    Abar Tbar newPbar '())))
                           (list t newsigma)))))))))))
```

```
; reduce-in
; Summary:  returns a tuple-state pair.  Similar to reduce-rd,
; except if successful, also removes matching tuple t from
; new state's Tbar set in tuple-state pair.
(define reduce-in
   (lambda (closure state-LBar)
      (let ((sigma (get-state state-LBar))
               (template (get-template closure)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((f ((lambda t)
                        (match?  template t))))
               (let ((t (exists?  Tbar f)))
                  (if (null?  t)
                     ('())
                     (let ((newTbar (set-diff
                                 Tbar (singleton t)))
                              (newPbar (union Pbar
                                 (make-event 'Econsumed t))))
                        (let ((newsigma (make-state
                                 Abar newTbar newPbar '())))
                           (list t newsigma))))))))))


; exists?
; Summary:  returns a matching tuple from TBar if one is found
; that satisfies the f function.  The f function is bound by the
; caller to check for a match with a particular template.
(define exists?
   (lambda (TBar f)
      (if (null?  TBar)
         ('())
         (let ((tuple (car TBar)))
            (if (f tuple)
               (tuple)
               (exists?
                  (set-diff TBar (singleton tuple))
                  f))))))
```

```
; Generate Children
; Summary:  Returns a state-LBar pair.
(define G
   (lambda (state-LBar)
      (let ((sigma (get-state state-LBar))
            (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma)))
            (let ((Lprocs (get-active-procs Abar)))
               (let ((randsub (get-rand-subset Lprocs)))
                  (genMeaning (as-list randsub)
                        (list (make-state
                                    Abar Tbar Pbar '())
                                 LBar)))))))))


; Generate Meaning
; Summary:  Returns a state-LBar pair.  Applies meaning function
; F-mu to all members of Lprocs, accumulating the effects of each
; Linda process' computation in the state-LBar pair returned.
(define genMeaning
   (lambda (Lprocs state-LBar)
         (if (null?  Lprocs)
            state-LBar
            (let ((jk-pair (car Lprocs))
                  (sigma (get-state state-LBar)))
               (let ((j (get-j jk-pair))
                     (k (get-k jk-pair))
                     (Abar (get-Abar sigma)))
                  (let ((tsubj (get-tuple j Abar)))
                     (genMeaning (cdr Lprocs)
                        (F-mu tsubj k state-LBar))))))))
```

```
; F-mu
; Summary:  Returns a state-LBar pair.  The meaning of the
; computation of single Linda process residing in tuple j,
; field k, is reflected in the return value.  The meaning is
; a random selection from the set of possible meanings.
(define F-mu
   (lambda (tsubj k state-LBar)
      (let ((meanings-of-tsubj-k
                (gen-set Lm tsubj k state-LBar)))
         (car (as-list meanings-of-tsubj-k)))))
```

```
; Lm
; Summary:  returns a state-LBar pair.  High level Linda meaning
; function.  Computational progress of a Linda process, in
; location k of tuple tsubj, is reflected in the state
; returned by this function.  Progress consists of internal
; and/or external computation.  In the case of the final
; active process within tsubj going passive, in addition to
; removing tsubj from Abar and adding to Tbar, an 'Egenerated
; event is added to Pbar.  If after making internal progress,
; a Linda primitive immediately follows, Lm enlists Lm-prim
; to do the rest.
(define Lm
   (lambda (tsubj k state-LBar)
      (let ((sigma (get-state state-LBar))
                (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
                   (Tbar (get-Tbar sigma))
                   (Pbar (get-Pbar sigma)))
            (let ((tsubj1 (tupleUpdate tsubj k
                         (composition rand Lm-comp))))
               (if (exists-active-field?  tsubj1)
                  (let ((Abar1 (union
                            (set-diff Abar (singleton tsubj))
                            (singleton tsubj1))))
                     (process-redex tsubj1 k
                              Abar1 Tbar Pbar LBar))
                  (let ((Abar1 (set-diff Abar
                              (singleton tsubj)))
                           (Tbar1 (union Tbar
                              (singleton tsubj1)))
                           (Pbar1 (union Pbar
                              (singleton (make-event
                                 'Egenerated tsubj1)))))
                     (process-redex tsubj1 k
                              Abar1 Tbar1 Pbar1 LBar)))))))
```

```
; Process redex
; Summary:  returns a state-LBar pair.  Helper function to
; complete the work of Lm.
(define process-redex
   (lambda (tsubj k Abar Tbar Pbar LBar)
      (let ((redex (get-redex tsubj k)))
         (if (linda-prim?  redex)
            (Lm-prim tsubj k
               (list (make-state Abar Tbar Pbar '())
                  LBar))
            (list (make-state Abar Tbar Pbar '())
               LBar)))))
```

```
; Lm-prim
; Summary:  returns a state-LBar pair.  High level Linda meaning
; function for external computation.  External computation
; consists of a process issuing one of the Linda primitives.
; Depending on whether the Linda primitive is synchronous or
; asynchronous, the process will suspend, 'Pending completion of
; the operation, or reduce the asynchronous primitive,
; respectively.
(define Lm-prim
   (lambda (tsubj k state-LBar)
      (let ((sigma (get-state state-LBar))
               (LBar (get-LBar state-LBar)))
         (let ((Abar (get-Abar sigma))
                  (Tbar (get-Tbar sigma))
                  (Pbar (get-Pbar sigma))
                  (redex (get-redex tsubj k)))
            (let ((handle (get-handle redex))
                     (lprim (get-Linda-prim redex))
                     (template (get-template redex)))
               (if (asynch-prim?  lprim)
                  ;asynchronous primitive
                  (let ((lambda3 (list lprim template)))
                     (let ((lambda2
                              (list 'force lambda3)))
                        (let ((lambda1 (list
                              ('send handle
                                 (list 'delay lambda2)))))
```

```
          (let ((LBar1 (union LBar
                        (singleton lambda1)))
                     (tsubj1 (tupleUpdate
                        tsubj k reduce-asynch)))
             (let ((Abar1 (union
                          (set-diff Abar
                           (singleton tsubj))
                          (singleton tsubj1))))
                (list (make-state
                        Abar1 Tbar Pbar '())
                   LBar1))))))
;synchronous primitive
(let ((lambda4 (list lprim template)))
   (let ((lambda3 (list 'let t
                (list 'force lambda4)
                'in (list 'delay (list
                   'react tsubj k t)))))
      (let ((lambda2 (list 'send
                (get-self-handle state-LBar)
                   (list 'force lambda3))))
         (let ((lambda1
                      (list 'send handle
                      (list 'delay lambda2))))
            (let ((LBar1 (union LBar
                          (singleton lambda1)))
                        (tsubj1 (tupleUpdate
                               tsubj k
                               make-pending)))
               (let ((Abar1 (union
                        (set-diff
                           Abar
                           (singleton tsubj))
                        (singleton tsubj1))))
                  (list (make-state
                        Abar1 Tbar Pbar '())
                     LBar1))))))))))))
```

```
; View function
; Summary:  creates a new view, if upsilon is an empty list of
; ROPEs; otherwise appends zero or more ROPEs to an existing
; view of computation (from sigma).
(define F-view
   (lambda (upsilon sigma)
      (if (null?  upsilon)
            (more-ropes sigma)
            (append (list (car upsilon))
               (F-view (cdr upsilon)
                  (get-next-state sigma))))))


; more ropes
; Summary:  helper function called by F-view; returns a list of
; zero or more ROPEs generated from the corresponding
; parallel event sets of sigma's traversal.
(define more-ropes
   (lambda (sigma)
      (if (null?  sigma)
         '()
         (let ((Pbar (get-Pbar sigma))
               (next-sigma (get-next-state sigma)))
            (let ((v-randsub (get-rand-subset Pbar)))
               (let ((rho (as-list v-randsub)))
                  (random-choice (list rho)
                     (append (list rho)
                           (more-ropes next-sigma)))))))))
```

6.4 Equivalence Proof

This section discusses the operational semantics of previous work with which we will be comparing paraDOS for Linda. We present our plan of attack for the equivalence proof, based on the assumptions of the previous operational seman-

$$\forall a : \tau \in \textbf{Value}_{\Box} \ : \ \{(a : \tau, a : \tau),\ (a : \tau, \bot : \tau),\ (\bot : \tau, a : \tau)\} \subset \mathit{match}$$

$$\forall \mathbf{s}, \mathbf{t} \in \textbf{Value}^{n}_{\diamond} \ : \ \bigwedge_{i=1}^{n} (\mathbf{s}[i], \mathbf{t}[i]) \in \mathit{match} \Leftrightarrow (\mathbf{s}, \mathbf{t}) \in \mathit{match}$$

Figure 6.7: The *TSspec match* relation.

tics. Finally, we discuss the contributions of paraDOS for Linda to the body of work in formal models of tuple space computation.

6.4.1 The *TSspec* Model

We establish the soundness of paraDOS for Linda by giving an equivalence proof of our operational semantics with the operational semantics for Linda's tuple space given in [CJY94] and also in Jensen's Ph.D. thesis [Jen94]. Jensen presents his semantics, *TSspec*, in section 3.2 of his thesis, pp. 48–51. For completeness, we present *TSspec*'s *match* relation in Figure 6.7, domain specification in Figure 6.8, and operational semantics in Figure 6.9.

The *match* relation specifies that templates and tuples match if their respective values match. Two values match if they are of the same type, and exactly one value is formal; or if both values are actual and the identical.

In Figure 6.9, the case for local evaluation consists of two inference rules. The second inference rule is a recursive specification for the *TSspec* transition relation, and defines how a tuple space (a multiset of tuples) can be partitioned into two multisets of tuples. The resulting multisets can then be further partitioned by recursively applying the second inference rule. Jensen's description of the

$$
\begin{array}{llcl}
\text{Type Names:} & \mathbf{Type} & = & \{\mathbf{int},\ \mathbf{char},\ \ldots\} \\
\text{Formal Values:} & \mathbf{Value}_{\perp} & = & \{\perp : \tau \mid \tau \in \mathbf{Type}\} \\
\text{Passive Values:} & \mathbf{Value}_{\square} & = & \{a : \tau \mid \tau \in \mathbf{Type},\ a \in \mathcal{V}_{\perp}\} \cup \mathbf{Value}_{\perp} \\
\text{General Values:} & \mathbf{Value}_{\diamond} & = & \{\mathbf{p} : \tau \mid \tau \in \mathbf{Type},\ \mathbf{p} \in \mathbf{Proc}\} \cup \mathbf{Value}_{\square} \\
\text{Passive Tuples:} & \mathbf{Tuple}_{\square} & = & \mathbf{Value}_{\square}^{\star} \\
\text{General Tuples:} & \mathbf{Tuple}_{\diamond} & = & \mathbf{Value}_{\diamond}^{\star} \\
\text{Tuple Space:} & \mathbf{TS}_{\diamond} & = & MS[\mathbf{Tuple}_{\diamond}] \\
\text{Linda Operations:} & \mathbf{Op} & = & \{\mathtt{eval}(\mathbf{t}) \mid \mathbf{t} \in \mathbf{Tuple}_{\diamond}\} \cup \\
 & & & \{\mathtt{out}(\mathbf{t}), \mathtt{rd}(\mathbf{s}), \mathtt{in}(\mathbf{s}) \mid \mathbf{s}, \mathbf{t} \in \mathbf{Tuple}_{\square}\} \\
\text{Linda Processes:} & \mathbf{Proc} & = & \{op.e,\ e(\mathbf{t}),\ e \mid \\
 & & & \quad op \in \mathbf{Op},\ \mathbf{t} \in \mathbf{Tuple}_{\square},\ e \in \mathcal{C}_p\}
\end{array}
$$

Figure 6.8: The *TSspec* Domain Specification.

parallelism specified by *TSspec* includes more information than what is conveyed by the second inference rule [Jen94, CJY94, Jen00]. In particular, the *TSspec* operational semantics describes concurrency through an arbitrary interleaving of a set of atomic transitions. This set of transitions derives from the partitions of tuple space that result from the recursive applications of the second inference rule. Computation proceeds within each partition of tuple space via a single, independent transition. Each of these transitions corresponds to one of *TSspec*'s non-recursive transition rules. Collectively, these transitions represent the set of atomic transitions to be interleaved.

The *TSspec* approach, via the second inference rule, provides an elegant formal specification of parallelism for tuple space computation. However, the instantiation of paraDOS for Linda does not use this approach. By avoiding partitioning, our approach is capable of modeling a higher degree of parallelism than that reflected in *TSspec*'s second inference rule. For example, consider Linda processes A and B, residing within different tuples, both about to issue a rd() primitive, where their respective templates both match tuple t. *TSspec* can partition a multiset for the tuple copying rule. Such a multiset contains the matching tuple t, and either the tuple containing Linda process A or the tuple containing the Linda process B. Once such a partition exists, a second partition to copy tuple t is no longer possible – even though it is theoretically possible for both Linda processes A and B to copy tuple t in parallel. ParaDOS for Linda permits this level of parallelism.

This equivalence proof focuses on individual computational steps, not degrees of parallelism, between *TSspec* and paraDOS for Linda. The reason for this focus is because the two models represent parallelism at different levels of abstraction: *TSspec* via interleaving events (transitions) from the second inference rule, paraDOS for Linda via parallel events. When reasoning about a *TSspec* trace, one cannot distinguish, in general, whether some sequence of events in the trace occurred sequentially, or resulted from interleaving two or more simultaneous, partitioned transitions. Thus, we restrict our attention to the only remaining case for a transition within a tuple space partition: the first inference rule for individual local computations. The first rule describes how a Linda process makes local computational progress while residing within a field of some tuple in tuple space.

Several assumptions of *TSspec* influence the framework of this equivalence proof. First, *TSspec* specifies the coordination of Linda processes via tuple space. Second, *TSspec* does not specify exactly what a Linda program is: *TSspec* is not a terminal transition system. Finally, concurrency is described by an arbitrary interleaving of a set of atomic transitions performed by the active processes.

Operational Semantics:

Linda's Tuple Space:

$$\mathit{TSspec} = \langle\ \Gamma_{ts},\ \rightarrowtail_{ts}\ \rangle \text{ where } \begin{array}{rcl} \Gamma_{ts} & = & \mathbf{TS}_{\diamond} \\ \rightarrowtail_{ts} & \subseteq & \Gamma_{ts} \times \Gamma_{ts} \end{array}$$

Tuple Spaces Transitions:

Process Creation:

$$\forall \mathbf{t}' \in \mathbf{Tuple}_{\diamond}\ :\ \{\!|\ \mathbf{t}[\mathtt{eval}(\mathbf{t}').\mathbf{p} : \tau]\ |\!\}\ \rightarrowtail_{ts}\ \{\!|\ \mathbf{t}[\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}.$$

Tuple Creation:

$$\forall \mathbf{t}' \in \mathbf{Tuple}_{\square}\ :\ \{\!|\ \mathbf{t}[\mathtt{out}(\mathbf{t}').\mathbf{p} : \tau]\ |\!\}\ \rightarrowtail_{ts}\ \{\!|\ \mathbf{t}[\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}.$$

Tuple Copying:

$$\forall (\mathbf{s}, \mathbf{t}') \in \mathit{match}\ :\ \{\!|\ \mathbf{t}[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}\ \rightarrowtail_{ts}\ \{\!|\ \mathbf{t}[\mathbf{p}(\mathbf{t}') : \tau],\ \mathbf{t}'\ |\!\}$$

Tuple Removal:

$$\forall (\mathbf{s}, \mathbf{t}') \in \mathit{match}\ :\ \{\!|\ \mathbf{t}[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}\ \rightarrowtail_{ts}\ \{\!|\ \mathbf{t}[\mathbf{p}(\mathbf{t}') : \tau]\ |\!\}$$

Local Evaluation:

$$\frac{\mathbf{p}'\ \rightarrowtail_{p}\ \mathbf{p}''}{\{\!|\ \mathbf{t}[\mathbf{p}' : \tau]\ |\!\}\ \rightarrowtail_{ts}\ \{\!|\ \mathbf{t}[\mathbf{p}'' : \tau]\ |\!\}} \qquad \frac{\mathbf{ts}'\ \rightarrowtail_{ts}\ \mathbf{ts}''}{\mathbf{ts} \uplus \mathbf{ts}'\ \rightarrowtail_{ts}\ \mathbf{ts} \uplus \mathbf{ts}''}$$

Abbreviations:

Configurations:

$$\mathbf{p}, \mathbf{p}', \mathbf{p}'' \in \mathbf{Proc},\ \ \mathbf{s}, \mathbf{t}, \mathbf{t}' \in \mathbf{Tuple}_{\diamond},\ \ \mathbf{ts}, \mathbf{ts}', \mathbf{ts}'' \in \mathbf{TS}_{\diamond}.$$

Figure 6.9: Operational Semantics for *TSspec*.

Thus, in this equivalence proof, we are not concerned with initial or final configurations of Linda programs, but rather the states of individual Linda processes and tuple space, and the subsequent states of these processes and tuple space after a transition. Moreover, since *TSspec* represents parallelism via an arbitrary interleaving of sequential computations, we will consider a restricted version of paraDOS for Linda, capable of computational progress by a single Linda process in each transition. This should not bother us going from *TSspec* to paraDOS, since this restriction is indeed one of the possibilities for any given transition. In the other direction, the restriction of paraDOS to single-process transitions will be sufficient to show equivalence with *TSspec*.

This equivalence proof will not be in the form of induction on the number of transitions. The proof is nontraditional because *TSspec* is defined in terms of a process's individual transitions. First, we define equivalence relations between the two models' configurations, tuples, and tuple fields. Next, we demonstrate that the transitions possible in one semantics are possible in the other semantics (i.e. what one can do, the other can do, and vice versa). In all cases, we must show that the equivalence relation on configurations holds between the two models' states before and after their respective transitions.

6.4.2 Definitions and Assumptions

Before we can state (and prove) our theorem, we must define our equivalence relations and state our assumptions. We define three equivalence relations, among state configurations ($\underset{cfg}{\longmapsto}$), tuples ($\underset{tpl}{\longmapsto}$), and tuple fields ($\underset{fld}{\longmapsto}$). Our assumptions concern notation, fonts, and the existence of helper functions.

We use different fonts to distinguish tuples from each of the two computational models. In particular, $\mathtt{t}$ refers to a tuple from the Linda instantiation of paraDOS, and $\mathbf{t}$ refers to a tuple from *TSspec*. In both cases, we use standard subscript notation to project individual tuple fields. Further, $\mathbf{t}[1]$ refers to the first field of tuple $\mathbf{t}$ from *TSspec*, where fields are numbered from 1 to $\#\mathbf{t}$ (similarly for `t` and paraDOS instantiated for Linda).

Note that $\underset{tpl}{\longmapsto}$ applies to tuple templates as well as tuples. λ is a closure containing a Linda primitive. Where convenient, we indicate the type of Linda primitive a closure contains with a subscript, e.g. λ_{rd}. Helper function $\texttt{lindaprim}(\lambda)$ extracts the Linda primitive operation from λ. Helper function $\texttt{asynchSub}(\overline{\Lambda})$ returns the subset of closures in $\overline{\Lambda}$ containing asynchronous Linda primitive operations.

The operational semantics of the synchronous Linda primitives within *TSspec* are guarded by the predicate *match*, which returns true for pairs of tuples that match. paraDOS for Linda assumes the existence of predicate `match?` for the same purpose.

We now define the equivalence relations. One way to describe a Linda program is as a collection of tuples. This description captures not only the passive tuples, but the Linda processes, which reside within the active tuples in tuple space. Thus, configurations between $\mathcal{P}^{\mathbf{TS}}$ and *TSspec* are equivalent if all the tuples in $\mathcal{P}^{\mathbf{TS}}$ have counterparts in *TSspec*, and all tuples in *TSspec* have counterparts in $\mathcal{P}^{\mathbf{TS}}$. Two tuples are equivalent if their respective fields are equivalent. Two tuple fields are semantically equivalent if their respective contents are syntactically equivalent; syntactic equality implies semantic equality. The definitions of $\underset{tpl}{\longmapsto}$ and $\underset{fld}{\longmapsto}$ are straightforward.

The definition of $\underset{cfg}{\longmapsto}$ merits further explanation. There are two main expressions, the first evaluating equivalence from *TSspec* to $\mathcal{P}^{\mathbf{TS}}$, the second evaluating equivalence from $\mathcal{P}^{\mathbf{TS}}$ to *TSspec*. The first condition is met if, for all tuples in *TSspec*'s tuple space, there is an equivalent tuple in either $\mathcal{P}^{\mathbf{TS}}$'s set of active tuples or set of passive tuples, or that the matching tuple in $\mathcal{P}^{\mathbf{TS}}$ resides within a communication closure yet to be reduced. In the case that the matching $\mathcal{P}^{\mathbf{TS}}$ tuple resides within a closure, there are two possibilities. The first possibility is that the matched tuple $\mathtt{t}$ belongs to the blocked process residing in the k-th field of the m-th tuple in tuple space. The second possibility is that the tuple $\mathtt{t}$ is to be placed in tuple space by an $\mathtt{out}()$ or $\mathtt{eval}()$ operation.

The second main condition is the conjunction of three sub-conditions. First, for all tuples in $\mathcal{P}^{\mathbf{TS}}$'s sets of active and passive tuples, there must be an equivalent tuple in *TSspec*'s tuple space. Second, for all $\mathcal{P}^{\mathbf{TS}}$ closures λ that are in the *reactCl* domain, there must be a tuple in *TSspec*'s tuple space that is equivalent to the tuple $\mathtt{t}$ contained in λ. Third, for all closures λ in $\mathcal{P}^{\mathbf{TS}}$ that contain the nonblocking Linda primitive operation $\mathtt{out(t)}$ or $\mathtt{eval(t)}$, there must be a tuple in *TSspec*'s tuple space that is equivalent to the tuple $\mathtt{t}$ contained in λ.

Definition 20 ($\underset{cfg}{\longmapsto}$) Let $K_i^{TS} = \mathbf{ts}_i$, $K_j^{pL} = \langle \sigma_j, \overline{\Lambda}_j \rangle$, and $\sigma_j = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where $\mathbf{ts}_i \in \mathsf{TS}_\diamond$ and $\langle \sigma_j, \overline{\Lambda}_j \rangle \in SCSPair$. Then $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$ iff

$$\Bigg[\forall \mathbf{t} \in \mathbf{ts}_i .\ \Bigg(\Big(\exists \mathtt{t} \in \overline{\mathcal{A}} \bigcup \overline{\mathcal{T}} \mid \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}\Big) \bigvee$$
$$\Big(\exists \lambda \in \overline{\Lambda}_j \ .\ \exists \mathtt{t} \in \mathit{tuple} \ .\ \exists \mathtt{m}, \mathtt{k} \ \mid \ (\lambda = \text{``}\mathtt{react(m,k,t)}\text{''} \bigvee$$
$$\mathtt{lindaprim}(\lambda) \in \{\mathtt{out(t)}, \mathtt{eval(t)}\}) \bigwedge \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}\Big)\Bigg)\Bigg] \bigwedge$$

$$\Bigg[\Big(\forall \mathtt{t} \in \overline{\mathcal{A}} \bigcup \overline{\mathcal{T}} \ .\ \exists \mathbf{t} \in \mathbf{ts}_i \mid \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}\Big) \bigwedge$$
$$\Big(\forall \lambda \in \overline{\Lambda}_j \bigcap \mathit{reactCl} \ .\ \exists \mathbf{t} \in \mathbf{ts}_i \ .\ \exists \mathtt{t} \in \mathit{tuple} \ .\ \exists \mathtt{m}, \mathtt{k} \ \mid$$
$$\lambda = \text{``}\mathtt{react(m,k,t)}\text{''} \bigwedge \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}\Big) \bigwedge$$
$$\Big(\forall \lambda \in \mathtt{asynchSub}(\overline{\Lambda}_j) \ .\ \exists \mathtt{t} \in \mathit{tuple} \ .\ \exists \mathbf{t} \in \mathbf{ts}_i \mid$$
$$\mathtt{lindaprim}(\lambda) \in \{\mathtt{out(t)}, \mathtt{eval(t)}\} \bigwedge \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}\Big)\Bigg].$$

Definition 21 ($\underset{tpl}{\longmapsto}$) $\mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}$ iff

$$(\#\mathbf{t} = |\mathtt{t}|) \bigwedge (\forall 1 \le k \le \#\mathbf{t}.\ \mathbf{t}[k] \underset{fld}{\longmapsto} \mathtt{t}[k]).$$

Definition 22 ($\underset{fld}{\longmapsto}$) $\mathtt{t}[k] \underset{fld}{\longmapsto} \mathtt{t}[k]$ iff

$\mathbf{t}[k] = \mathtt{t}[k].\mathtt{contents}$ (syntactic equality implies semantic equality)

6.4.3 Theorem and Proof

Theorem 2 states that we can model the computations of all Linda processes equivalently in both *TSspec* and $\mathcal{P}^{\mathbf{TS}}$. Specifically, if Linda process $\mathbf{p}$ is contained in equivalent configurations from *TSspec* and $\mathcal{P}^{\mathbf{TS}}$ then, for all possible transitions in *TSspec* involving $\mathbf{p}$, there exists zero or more transitions to an equivalent state in $\mathcal{P}^{\mathbf{TS}}$. Similarly, for all possible transitions in $\mathcal{P}^{\mathbf{TS}}$, there exists zero or more transitions to an equivalent state in *TSspec*. The proof considers all possible cases of transitions from *TSspec* to $\mathcal{P}^{\mathbf{TS}}$, and from $\mathcal{P}^{\mathbf{TS}}$ to *TSspec*.

Theorem 2 *For all Linda processes,* $\mathbf{p}$*, let* K_i^{TS} *be a configuration of* K^{TS} *containing* $\mathbf{p}$*, and* K_j^{pL} *be a configuration of* K^{pL} *containing* $\mathbf{p}$*, s.t.* $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$*. Then*

1. $K_i^{TS} \rightarrowtail_{ts} K_{i+1}^{TS} \implies \exists n \geq 0$ *s.t.* $K_j^{pL} \rightarrowtail_{pL}^{n} K_{j+n}^{pL} \bigwedge K_{i+1}^{TS} \underset{cfg}{\longmapsto} K_{j+n}^{pL}$, *and*
2. $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL} \implies \exists n \geq 0$ *s.t.* $K_i^{TS} \rightarrowtail_{ts}^{n} K_{i+n}^{TS} \bigwedge K_{i+n}^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$.

Proof:

Part 1: Consider each case of tuple space transitions in *TSspec*. For each case, demonstrate the equivalent transition in paraDOS for Linda.

Process Creation: $\forall \mathbf{t}' \in \mathsf{Tuple}_{\diamond}$: $\{\!| \ \mathbf{t}[\mathtt{eval}(\mathbf{t}').\mathbf{p} : \tau] \ |\!\} \rightarrowtail_{ts} \{\!| \ \mathbf{t}[\mathbf{p} : \tau], \ \mathbf{t}' \ |\!\}$

1. Let $K_i^{TS} \supseteq \{\!| \ \mathbf{t}[\mathtt{eval}(\mathbf{t}').\mathbf{p} : \tau] \ |\!\}$ and $K_{i+1}^{TS} \supseteq \{\!| \ \mathbf{t}[\mathbf{p} : \tau], \ \mathbf{t}' \ |\!\}$, where $K_i^{TS} \rightarrowtail_{ts} K_{i+1}^{TS}$.
   Let $\mathbf{t}_1$ denote tuple $\mathbf{t}$ before, and $\mathbf{t}_2$ denote tuple $\mathbf{t}$ after, transition $\rightarrowtail_{ts}$.

2. Given $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$, by definition of $\underset{cfg}{\longmapsto}$, $\exists \mathtt{t} \in \sigma_j.\overline{\mathcal{A}} \mid \mathbf{t}_1 \underset{tpl}{\longmapsto} \mathtt{t}$,
   and by definition of $\underset{tpl}{\longmapsto}$, $\exists \ell \mid \mathbf{t}_1[\mathtt{eval}(\mathbf{t}').\mathbf{p} : \tau] \underset{fld}{\longmapsto} \mathtt{t}[\ell]$,
   and by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_1[\ell] = \mathtt{t}[\ell].\mathtt{contents}$.

3. By definition of $\rightarrowtail_{ts}$,

   the meaning of $\mathbf{t}_1[\mathtt{eval}(\mathbf{t}').\mathbf{p} : \tau]$ yields $\{\!|\ \mathbf{t}_2[\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}$.

   By definition of Lm-prim(),

   $\exists \mathtt{t}'' \in \mathit{tuple} \mid \mathtt{t}''[\ell].\mathtt{contents} = \mathbf{t}_2[\ell]\ \bigwedge$
   $\forall 1 \leq k \neq \ell \leq \#\mathbf{t}.\ \mathtt{t}''[k] = \mathtt{t}[k]$,

   and $\exists \lambda \in \mathit{asynchCl}\ \exists \mathtt{t}' \in \mathit{tuple} \mid$

   $\mathtt{lindaprim}(\lambda) = \mathtt{eval}(\mathtt{t}')\ \bigwedge\ \mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}'$.

4. By definition of $\underset{fld}{\longmapsto}$ and $\mathtt{t}''[\ell]$, $\mathbf{t}_2[\ell] \underset{fld}{\longmapsto} \mathtt{t}''[\ell]$, and

   By definition of $\underset{tpl}{\longmapsto}$ and $\mathtt{t}''$, $\mathbf{t}_2 \underset{tpl}{\longmapsto} \mathtt{t}''$.

5. By definition of $\mathtt{G}$ applied to σ_j and $\overline{\Lambda}_j$, and by definition of $\mathtt{t}$, $\mathtt{t}''$, and λ,

   $\exists \sigma_{j+1} \in \mathit{state} \mid \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

   $\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}) \bigcup \{\mathtt{t}''\}$,

   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,

   $\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,

   $\sigma_{j+1}.\sigma_{next}$ is initially undefined, and

   $\exists \overline{\Lambda}_{j+1} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+1} = \overline{\Lambda}_j \bigcup \{\lambda\}$.

6. By definition of $\rightarrowtail_{pL}$, and by steps 3 and 5, where $\sigma_j.\sigma_{next} = \sigma_{j+1}$ and $K_{j+1}^{pL} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,

   $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL}$ is one legal transition.

7. By definition of `reduce-send` applied to λ,

   $\exists \lambda' \in \mathit{asynchLPrim} \mid \lambda' = \mathtt{reduce\text{-}send}(\lambda)\ \bigwedge\ \mathtt{lindaprim}(\lambda') = \mathtt{eval}(\mathtt{t}')$.

8. By definition of `F-LambdaBar` applied to σ_{j+1} and $\overline{\Lambda}_{j+1}$, and by definition of λ and λ',

   $\exists \sigma_{j+2} \in \mathit{state} \mid \sigma_{j+2} = \sigma_{j+1}$,

   $\exists \overline{\Lambda}_{j+2} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+2} = (\overline{\Lambda}_{j+1} - \{\lambda\}) \bigcup \{\lambda'\}$.

9. By definition of $\rightarrowtail_{pL}$, and by steps 7 and 8, where $\sigma_{j+1}.\sigma_{next} = \sigma_{j+2}$ and $K_{j+2}^{pL} = \langle \sigma_{j+2}, \overline{\Lambda}_{j+2} \rangle$,

   $K_{j+1}^{pL} \rightarrowtail_{pL} K_{j+2}^{pL}$ is one legal transition.

10. By definition of $\texttt{reduce-eval}$ applied to λ', and by definition of $\texttt{t}'$,
    $\texttt{t}' = \texttt{reduce-eval}(\lambda')$.
11. By definition of $\texttt{F-LambdaBar}$ applied to σ_{j+2} and $\overline{\Lambda}_{j+2}$, and by definition of $\texttt{t}'$, and λ',
    $\exists \sigma_{j+3} \in \mathit{state} \mid \sigma_{j+3} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where
    $\sigma_{j+3}.\overline{\mathcal{A}} = \sigma_{j+2}.\overline{\mathcal{A}} \bigcup \{\texttt{t}'\}$,
    $\sigma_{j+3}.\overline{\mathcal{T}} = \sigma_{j+2}.\overline{\mathcal{T}}$,
    $\sigma_{j+3}.\overline{\mathcal{P}} = \{\langle \texttt{'Egenerating}, \texttt{t}' \rangle\}$,
    $\sigma_{j+3}.\sigma_{next}$ is initially undefined, and
    $\exists \overline{\Lambda}_{j+3} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+3} = \overline{\Lambda}_{j+2} - \{\lambda'\}$
12. By definition of $\rightarrowtail_{pL}$, and by steps 10 and 11, where $\sigma_{j+2}.\sigma_{next} = \sigma_{j+3}$ and $K^{pL}_{j+3} = \langle \sigma_{j+3}, \overline{\Lambda}_{j+3} \rangle$,
    $K^{pL}_{j+2} \rightarrowtail_{pL} K^{pL}_{j+3}$ is one legal transition.
13. By definition of $\underset{cfg}{\longmapsto}$ and σ_{j+3}, and by steps 3, 4, 5, and 11, where respectively, $\mathbf{t}' \underset{tpl}{\longmapsto} \texttt{t}'$, $\mathbf{t}_2 \underset{tpl}{\longmapsto} \texttt{t}''$, $\texttt{t}'' \in \sigma_{j+1}.\overline{\mathcal{A}}$, and $\texttt{t}' \in \sigma_{j+3}.\overline{\mathcal{A}}$,
    $K^{TS}_{i+1} \underset{cfg}{\longmapsto} K^{pL}_{j+3}$.
14. From the transitions in steps 6, 9, and 12, and by the configuration equivalence in step 13, we demonstrated the ability of paraDOS for Linda to perform in $n = 3$ transitions the *TSspec* computational step of **Process Creation**.

Tuple Creation: $\forall \mathbf{t}' \in \mathbf{Tuple}_{\Box} \; : \; \{\!|\; \mathbf{t}[\texttt{out}(\mathbf{t}').\mathbf{p} : \tau] \;|\!\} \rightarrowtail_{ts} \{\!|\; \mathbf{t}[\mathbf{p} : \tau],\; \mathbf{t}' \;|\!\}$

1. Let $K^{TS}_i \supseteq \{\!|\; \mathbf{t}[\texttt{out}(\mathbf{t}').\mathbf{p} : \tau] \;|\!\}$ and $K^{TS}_{i+1} \supseteq \{\!|\; \mathbf{t}[\mathbf{p} : \tau],\; \mathbf{t}' \;|\!\}$, where $K^{TS}_i \rightarrowtail_{ts} K^{TS}_{i+1}$.
   Let $\mathbf{t}_1$ denote tuple $\mathbf{t}$ before, and $\mathbf{t}_2$ denote tuple $\mathbf{t}$ after, transition $\rightarrowtail_{ts}$.
2. Given $K^{TS}_i \underset{cfg}{\longmapsto} K^{pL}_j$, by definition of $\underset{cfg}{\longmapsto}$, $\exists \texttt{t} \in \sigma_j.\overline{\mathcal{A}} \mid \mathbf{t}_1 \underset{tpl}{\longmapsto} \texttt{t}$,
   and by definition of $\underset{tpl}{\longmapsto}$, $\exists \ell \mid \mathbf{t}_1[\texttt{out}(\mathbf{t}').\mathbf{p} : \tau] \underset{fld}{\longmapsto} \texttt{t}[\ell]$,
   and by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_1[\ell] = \texttt{t}[\ell].\texttt{contents}$.

3. By definition of $\rightarrowtail_{ts}$,

   the meaning of $\mathbf{t}_1[\texttt{out}(\mathbf{t}').\mathbf{p} : \tau]$ yields $\{\!|\ \mathbf{t}_2[\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}$.

   By definition of Lm-prim(),

   $\exists \texttt{t}'' \in \mathit{tuple} \mid \texttt{t}''[\ell].\texttt{contents} = \mathbf{t}_2[\mathbf{p} : \tau]\ \bigwedge$
   $\forall 1 \leq k \neq \ell \leq \#\mathbf{t}.\ \texttt{t}''[k] = \texttt{t}[k]$,

   and $\exists \lambda \in \mathit{asynchCl}\ \exists \texttt{t}' \in \mathit{tuple} \mid$

   $\texttt{lindaprim}(\lambda) = \texttt{out}(\texttt{t}')\ \bigwedge\ \mathbf{t}' \underset{tpl}{\longmapsto} \texttt{t}'$.

4. By definition of $\underset{fld}{\longmapsto}$ and $\texttt{t}''[\ell]$, $\mathbf{t}_2[\ell] \underset{fld}{\longmapsto} \texttt{t}''[\ell]$, and

   By definition of $\underset{tpl}{\longmapsto}$ and $\texttt{t}''$, $\mathbf{t}_2 \underset{tpl}{\longmapsto} \texttt{t}''$.

5. By definition of $\texttt{G}$ applied to σ_j and $\overline{\Lambda}_j$, and by definition of $\texttt{t}$, $\texttt{t}''$, and λ,

   $\exists \sigma_{j+1} \in \mathit{state} \mid \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

   $\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\texttt{t}\}) \bigcup \{\texttt{t}''\}$,

   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,

   $\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,

   $\sigma_{j+1}.\sigma_{next}$ is initially undefined, and

   $\exists \overline{\Lambda}_{j+1} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+1} = \overline{\Lambda}_j \bigcup \{\lambda\}$.

6. By definition of $\rightarrowtail_{pL}$, and by steps 3 and 5, where $\sigma_j.\sigma_{next} = \sigma_{j+1}$ and $K^{pL}_{j+1} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,

   $K^{pL}_j \rightarrowtail_{pL} K^{pL}_{j+1}$ is one legal transition.

7. By definition of $\texttt{reduce-send}$ applied to λ,

   $\exists \lambda' \in \mathit{asynchLPrim} \mid$

   $\lambda' = \texttt{reduce-send}(\lambda)\ \bigwedge\ \texttt{lindaprim}(\lambda') = \texttt{out}(\texttt{t}')$.

8. By definition of $\texttt{F-LambdaBar}$ applied to σ_{j+1} and $\overline{\Lambda}_{j+1}$, and by definition of λ and λ',

   $\exists \sigma_{j+2} \in \mathit{state} \mid \sigma_{j+2} = \sigma_{j+1}$,

   $\exists \overline{\Lambda}_{j+2} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+2} = (\overline{\Lambda}_{j+1} - \{\lambda\}) \bigcup \{\lambda'\}$.

9. By definition of $\rightarrowtail_{pL}$, and by steps 7 and 8, where $\sigma_{j+1}.\sigma_{next} = \sigma_{j+2}$, and $K^{pL}_{j+2} = \langle \sigma_{j+2}, \overline{\Lambda}_{j+2} \rangle$.

   $K^{pL}_{j+1} \rightarrowtail_{pL} K^{pL}_{j+2}$ is one legal transition.

10. By definition of $\texttt{reduce-out}$ applied to λ', and by definition of $\texttt{t}'$,

    $\texttt{t}' = \texttt{reduce-out}(\lambda')$.

11. By definition of `F-LambdaBar` applied to σ_{j+2} and $\overline{\Lambda}_{j+2}$, and by definition of $\mathtt{t}'$, and λ',

    $\exists\sigma_{j+3} \in \mathit{state} \mid \sigma_{j+3} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

    $\sigma_{j+3}.\overline{\mathcal{A}} = \sigma_{j+2}.\overline{\mathcal{A}}$

    $\sigma_{j+3}.\overline{\mathcal{T}} = \sigma_{j+2}.\overline{\mathcal{T}} \bigcup \{\mathtt{t}'\}$,

    $\sigma_{j+3}.\overline{\mathcal{P}} = \{\langle \mathtt{'Ecreated}, \mathtt{t}' \rangle\}$,

    $\sigma_{j+3}.\sigma_{next}$ is initially undefined, and

    $\exists\overline{\Lambda}_{j+3} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+3} = \overline{\Lambda}_{j+2} - \{\lambda'\}$

12. By definition of $\rightarrowtail_{pL}$, and by steps 10 and 11, where $\sigma_{j+2}.\sigma_{next} = \sigma_{j+3}$ and $K^{pL}_{j+3} = \langle \sigma_{j+3}, \overline{\Lambda}_{j+3} \rangle$,

    $K^{pL}_{j+2} \rightarrowtail_{pL} K^{pL}_{j+3}$ is one legal transition.

13. By definition of $\underset{cfg}{\longmapsto}$ and σ_{j+3}, and by steps 3, 4, 5, and 11, where respectively, $\mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}'$, $\mathbf{t}_2 \underset{tpl}{\longmapsto} \mathtt{t}''$, $\mathtt{t}'' \in \sigma_{j+1}.\overline{\mathcal{A}}$, and $\mathtt{t}' \in \sigma_{j+3}.\overline{\mathcal{T}}$,

    $K^{TS}_{i+1} \underset{cfg}{\longmapsto} K^{pL}_{j+3}$.

14. From the transitions in steps 6, 9, and 12, and by the configuration equivalence in step 13, we demonstrated the ability of paraDOS for Linda to perform in $n = 3$ tranitions the *TSspec* computational step of **Tuple Creation**.

Tuple Copying: $\forall(\mathbf{s}, \mathbf{t}') \in \mathit{match}$:

$$\{\!|\ \mathbf{t}[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\} \rightarrowtail_{ts} \{\!|\ \mathbf{t}[\mathbf{p}(\mathbf{t}') : \tau],\ \mathbf{t}'\ |\!\}$$

1. Let $K^{TS}_i \supseteq \{\!|\ \mathbf{t}[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}$ and $K^{TS}_{i+1} \supseteq \{\!|\ \mathbf{t}[\mathbf{p}(\mathbf{t}') : \tau],\ \mathbf{t}'\ |\!\}$, where $K^{TS}_i \rightarrowtail_{ts} K^{TS}_{i+1}$.

   Let $\mathbf{t}_1$ denote tuple $\mathbf{t}$ before, and $\mathbf{t}_2$ denote tuple $\mathbf{t}$ after, transition $\rightarrowtail_{ts}$.

2. Given $K^{TS}_i \underset{cfg}{\longmapsto} K^{pL}_j$, by definition of $\underset{cfg}{\longmapsto}$,

   $\exists\mathtt{t} \in \sigma_j.\overline{\mathcal{A}}\ \exists\mathtt{t}' \in \sigma_j.\overline{\mathcal{T}} \mid \mathbf{t}_1 \underset{tpl}{\longmapsto} \mathtt{t} \bigwedge \mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}'$,

   and by definition of $\underset{tpl}{\longmapsto}$, $\exists\ell \mid \mathbf{t}_1[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau] \underset{fld}{\longmapsto} \mathtt{t}[\ell]$,

   and by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_1[\ell] = \mathtt{t}[\ell].\mathtt{contents}$,

    $\exists\mathtt{s} \in \mathit{tuple} \mid \mathbf{s} \underset{tpl}{\longmapsto} \mathtt{s}$, and

    $C[\ ]$ is the evaluation context of $t[\ell]$.`contents` with redex $\mathtt{rd}(\mathtt{s})$.

3. Given $(\mathbf{s}, \mathbf{t}') \in \mathit{match}$, and from step 2, $\mathbf{s} \underset{tpl}{\longmapsto} \mathtt{s}$ and $\mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}'$,

   (match? s t$'$) evaluates true.

4. By definition of $\longrightarrow\!\!\!\!\!\succ_{ts}$, the meaning of $\{\!|\ \mathbf{t}_1[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}$ yields $\{\!|\ \mathbf{t}_2[\mathbf{p}(\mathbf{t}') : \tau],\ \mathbf{t}'\ |\!\}$.

   By definition of Lm-prim(),

   $\exists \mathtt{t}'' \in \mathit{tuple} \mid \mathtt{t}''[\ell].\mathtt{type} =$ 'Pending' $\bigwedge$

   $\mathtt{t}''[\ell].\mathtt{contents} = \mathtt{t}[\ell].\mathtt{contents}\ \bigwedge$

   $\forall 1 \leq k \neq \ell \leq \#\mathbf{t}.\ \mathtt{t}''[k] = \mathtt{t}[k]$, and

   $\exists \lambda \in \mathit{synchCl} \mid \mathtt{lindaprim}(\lambda) = \mathtt{rd}(\mathtt{s})$.

5. By definition of G applied to σ_j and $\overline{\Lambda}_j$, and by definition of t, t$''$, and λ,

   $\exists \sigma_{j+1} \in \mathit{state} \mid \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

   $\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}) \bigcup \{\mathtt{t}''\}$,

   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,

   $\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,

   $\sigma_{j+1}.\sigma_{next}$ is intially undefined, and

   $\exists \overline{\Lambda}_{j+1} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+1} = \overline{\Lambda}_j \bigcup \{\lambda\}$.

6. By definition of $\longrightarrow\!\!\!\!\!\succ_{pL}$, and by steps 4 and 5, where $\sigma_j.\sigma_{next} = \sigma_{j+1}$, and $K^{pL}_{j+1} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,

   $K^{pL}_j \ \longrightarrow\!\!\!\!\!\succ_{pL}\ K^{pL}_{j+1}$ is one legal transition.

7. By definition of reduce-send applied to λ,

   $\exists \lambda' \in \mathit{sendCl} \mid \lambda' =$reduce-send$(\lambda)\ \bigwedge\ \mathtt{lindaprim}(\lambda') = \mathtt{rd}(\mathtt{t}')$.

8. By definition of F-LambdaBar applied to σ_{j+1} and $\overline{\Lambda}_{j+1}$, and by definition of λ and λ',

   $\exists \sigma_{j+2} \in \mathit{state} \mid \sigma_{j+2} = \sigma_{j+1}$.

   $\exists \overline{\Lambda}_{j+2} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+2} = (\overline{\Lambda}_{j+1} - \{\lambda\}) \bigcup \{\lambda'\}$.

9. By definition of $\longrightarrow\!\!\!\!\!\succ_{pL}$, and by steps 7 and 8, where $\sigma_{j+1}.\sigma_{next} = \sigma_{j+2}$, and $K^{pL}_{j+2} = \langle \sigma_{j+2}, \overline{\Lambda}_{j+2} \rangle$,

   $K^{pL}_{j+1} \ \longrightarrow\!\!\!\!\!\succ_{pL}\ K^{pL}_{j+2}$ is one legal transition.

10. By definition of reduce-rd,

    rd(s) matches t$'$, since from step 3, (match? s t$'$) evaluates true.

11. By definition of reduce-let,

    $\exists \lambda'' \in \mathit{matchCl} \mid$ binding t$'$ within λ' yields λ''.

12. By definition of F-LambdaBar applied to σ_{j+2} and $\overline{\Lambda}_{j+2}$, and by definition of $\mathtt{t}'$, λ', and λ'',

    $\exists\sigma_{j+3} \mid \sigma_{j+3} = \langle\overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next}\rangle$, where
    $\sigma_{j+3}.\overline{\mathcal{A}} = \sigma_{j+2}.\overline{\mathcal{A}}$,
    $\sigma_{j+3}.\overline{\mathcal{T}} = \sigma_{j+2}.\overline{\mathcal{T}}$,
    $\sigma_{j+3}.\overline{\mathcal{P}} = \{\langle\mathtt{'Ecopied}, \mathtt{t}'\rangle\}$, and
    $\sigma_{j+3}.\sigma_{next}$ is initially undefined.
    $\exists\overline{\Lambda}_{j+3} \mid \overline{\Lambda}_{j+3} = (\overline{\Lambda}_{j+2} - \{\lambda'\}) \bigcup \{\lambda''\}$.

13. By definition of $\rightarrowtail_{pL}$, and by steps 10, 11, and 12, where $\sigma_{j+2}.\sigma_{next} = \sigma_{j+3}$, and $K^{pL}_{j+3} = \langle\sigma_{j+3}, \overline{\Lambda}_{j+3}\rangle$,

    $K^{pL}_{j+2} \rightarrowtail_{pL} K^{pL}_{j+3}$ is one legal transition.

14. By definition of reduce-react and λ'',

    $\exists\mathtt{t}''' \in \mathit{tuple} \mid \mathtt{t}'''[\ell].\mathtt{type} =$ 'Active' $\bigwedge \mathtt{t}'''[\ell].\mathtt{contents} = C[\mathtt{t}'] \bigwedge$
    $\forall 1 \leq k \neq \ell \leq \#\mathbf{t}.\ \mathtt{t}'''[k] = \mathtt{t}''[k]$.

15. By definition of $\underset{fld}{\longmapsto}$ and $\mathtt{t}'''[\ell]$ in step 14, $\mathbf{t}_2[\ell] \underset{fld}{\longmapsto} \mathtt{t}'''[\ell]$, and

    By definition of $\underset{tpl}{\longmapsto}$ and $\mathtt{t}'''$, $\mathbf{t}_2 \underset{tpl}{\longmapsto} \mathtt{t}'''$.

16. By definition of F-LambdaBar applied to σ_{j+3} and $\overline{\Lambda}_{j+3}$, and by definition of $\mathtt{t}''$, $\mathtt{t}'''$, and λ'',

    $\exists\sigma_{j+4} \mid \sigma_{j+4} = \langle\overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next}\rangle$, where
    $\sigma_{j+4}.\overline{\mathcal{A}} = (\sigma_{j+3}.\overline{\mathcal{A}} - \{\mathtt{t}''\}) \bigcup \{\mathtt{t}'''\}$,
    $\sigma_{j+4}.\overline{\mathcal{T}} = \sigma_{j+3}.\overline{\mathcal{T}}$,
    $\sigma_{j+4}.\overline{\mathcal{P}} = \emptyset$,
    $\sigma_{j+4}.\sigma_{next}$ is initially undefined.
    $\exists\overline{\Lambda}_{j+4} \mid \overline{\Lambda}_{j+4} = \overline{\Lambda}_{j+3} - \{\lambda''\}$

17. By definition of $\rightarrowtail_{pL}$, and by steps 14 and 16, where $\sigma_{j+3}.\sigma_{next} = \sigma_{j+4}$ and $K^{pL}_{j+4} = \langle\sigma_{j+4}, \overline{\Lambda}_{j+4}\rangle$.

    $K^{pL}_{j+3} \rightarrowtail_{pL} K^{pL}_{j+4}$ is one legal transition.

18. By definition of $\underset{cfg}{\longmapsto}$ and σ_{j+4}, and by steps 2, 15, and 16, where respectively, $(\mathtt{t}' \in \sigma_j.\overline{\mathcal{T}} \bigwedge \mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}')$, $\mathbf{t}_2 \underset{tpl}{\longmapsto} \mathtt{t}'''$, and $\mathtt{t}' \in \sigma_{j+4}.\overline{\mathcal{A}}$,

    $K^{TS}_{i+1} \underset{cfg}{\longmapsto} K^{pL}_{j+4}$.

19. From the transitions in steps 6, 9, 13, and 17, and by the configuration equivalence in step 18, we demonstrated the ability of paraDOS for Linda to perform in $n = 4$ tranitions the *TSspec* computational step of **Tuple Copying**.

Tuple Removal: $\forall(\mathbf{s},\mathbf{t}') \in \mathit{match}\ :\ \{\!|\ \mathbf{t}[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\} \rightarrowtail_{ts} \{\!|\ \mathbf{t}[\mathbf{p}(\mathbf{t}') : \tau]\ |\!\}$

1. Let $K_i^{TS} \supseteq \{\!|\ \mathbf{t}[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}$ and $K_{i+1}^{TS} \supseteq \{\!|\ \mathbf{t}[\mathbf{p}(\mathbf{t}') : \tau]\ |\!\}$, where $K_i^{TS} \rightarrowtail_{ts} K_{i+1}^{TS}$.
   Let $\mathbf{t}_1$ denote tuple $\mathbf{t}$ before, and $\mathbf{t}_2$ denote tuple $\mathbf{t}$ after, transition $\rightarrowtail_{ts}$.
2. Given $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$, by definition of $\underset{cfg}{\longmapsto}$,
   $\exists \mathtt{t} \in \sigma_j.\overline{\mathcal{A}}\ \exists \mathtt{t}' \in \sigma_j.\overline{\mathcal{T}} \mid \mathbf{t}_1 \underset{tpl}{\longmapsto} \mathtt{t} \bigwedge \mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}'$,
   and by definition of $\underset{tpl}{\longmapsto}$, $\exists \ell \mid \mathbf{t}_1[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau] \underset{fld}{\longmapsto} \mathtt{t}[\ell]$,
   and by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_1[\ell] = \mathtt{t}[\ell].\mathtt{contents}$,
   $\exists \mathtt{s} \in \mathit{tuple} \mid \mathbf{s} \underset{tpl}{\longmapsto} \mathtt{s}$, and
   $C[\ ]$ is the evaluation context of $t[\ell].\mathtt{contents}$ with redex $\mathtt{in}(\mathtt{s})$.
3. Given $(\mathbf{s},\mathbf{t}') \in \mathit{match}$, and from step 2, $\mathbf{s} \underset{tpl}{\longmapsto} \mathtt{s}$ and $\mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}'$,
   $(\mathtt{match?\ s\ t}')$ evaluates true.
4. By definition of $\rightarrowtail_{ts}$, the meaning of $\{\!|\ \mathbf{t}_1[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}'\ |\!\}$ yields $\{\!|\ \mathbf{t}_2[\mathbf{p}(\mathbf{t}') : \tau]\ |\!\}$.
   By definition of Lm-prim(),
   $\exists \mathtt{t}'' \in \mathit{tuple} \mid \mathtt{t}''[\ell].\mathtt{type} =$ 'Pending' $\bigwedge$
   $\mathtt{t}''[\ell].\mathtt{contents} = \mathtt{t}[\ell].\mathtt{contents} \bigwedge$
   $\forall 1 \le k \ne \ell \le \#\mathtt{t}.\ \mathtt{t}''[k] = \mathtt{t}[k]$, and
   $\exists \lambda \in \mathit{synchCl} \mid \mathtt{lindaprim}(\lambda) = \mathtt{in}(\mathtt{s})$.
5. By definition of $\mathtt{G}$ applied to σ_j and $\overline{\Lambda}_j$, and by definition of $\mathtt{t}$, $\mathtt{t}''$, and λ,
   $\exists \sigma_{j+1} \in \mathit{state} \mid \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next}\rangle$ where
   $\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}) \bigcup \{\mathtt{t}''\}$,
   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
   $\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,
   $\sigma_{j+1}.\sigma_{next}$ is intially undefined, and
   $\exists \overline{\Lambda}_{j+1} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+1} = \overline{\Lambda}_j \bigcup \{\lambda\}$.
6. By definition of $\rightarrowtail_{pL}$, and by steps 4 and 5, where $\sigma_j.\sigma_{next} = \sigma_{j+1}$, and $K_{j+1}^{pL} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1}\rangle$,
   $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL}$ is one legal transition.
7. By definition of `reduce-send` applied to λ,
   $\exists \lambda' \in \mathit{sendCl} \mid \lambda' = \mathtt{reduce\text{-}send}(\lambda) \bigwedge \mathtt{lindaprim}(\lambda') = \mathtt{in}(\mathtt{t}')$.

8. By definition of `F-LambdaBar` applied to σ_{j+1} and $\overline{\Lambda}_{j+1}$, and by definition of λ and λ',

   $\exists \sigma_{j+2} \in \mathit{state} \mid \sigma_{j+2} = \sigma_{j+1}.$

   $\exists \overline{\Lambda}_{j+2} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+2} = (\overline{\Lambda}_{j+1} - \{\lambda\}) \bigcup \{\lambda'\}.$

9. By definition of $\rightarrowtail_{pL}$, and by steps 7 and 8, where $\sigma_{j+1}.\sigma_{next} = \sigma_{j+2}$, and $K^{pL}_{j+2} = \langle \sigma_{j+2}, \overline{\Lambda}_{j+2} \rangle$,

   $K^{pL}_{j+1} \rightarrowtail_{pL} K^{pL}_{j+2}$ is one legal transition.

10. By definition of `reduce-in`,

    `rd(s)` matches $\mathtt{t}'$, since from step 3, (`match? s` $\mathtt{t}'$) evaluates true.

11. By definition of `reduce-let`,

    $\exists \lambda'' \in \mathit{matchCl} \mid$ binding $\mathtt{t}'$ within λ' yields λ''.

12. By definition of `F-LambdaBar` applied to σ_{j+2} and $\overline{\Lambda}_{j+2}$, and by definition of $\mathtt{t}'$, λ', and λ'',

    $\exists \sigma_{j+3} \mid \sigma_{j+3} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where

    $\sigma_{j+3}.\overline{\mathcal{A}} = \sigma_{j+2}.\overline{\mathcal{A}}$,

    $\sigma_{j+3}.\overline{\mathcal{T}} = \sigma_{j+2}.\overline{\mathcal{T}} - \{\mathtt{t}'\}$,

    $\sigma_{j+3}.\overline{\mathcal{P}} = \{\langle \mathtt{'Econsumed}, \mathtt{t}' \rangle\}$, and

    $\sigma_{j+3}.\sigma_{next}$ is initially undefined.

    $\exists \overline{\Lambda}_{j+3} \mid \overline{\Lambda}_{j+3} = (\overline{\Lambda}_{j+2} - \{\lambda'\}) \bigcup \{\lambda''\}.$

13. By definition of $\rightarrowtail_{pL}$, and by steps 10, 11, and 12, where $\sigma_{j+2}.\sigma_{next} = \sigma_{j+3}$, and $K^{pL}_{j+3} = \langle \sigma_{j+3}, \overline{\Lambda}_{j+3} \rangle$,

    $K^{pL}_{j+2} \rightarrowtail_{pL} K^{pL}_{j+3}$ is one legal transition.

14. By definition of `reduce-react` and λ'',

    $\exists \mathtt{t}''' \in \mathit{tuple} \mid \mathtt{t}'''[\ell].\mathtt{type} =$ 'Active' $\bigwedge$ $\mathtt{t}'''[\ell].\mathtt{contents} = C[\mathtt{t}']$ $\bigwedge$
    $\forall 1 \leq k \neq \ell \leq \#\mathbf{t}.\ \mathtt{t}'''[k] = \mathtt{t}''[k]$.

15. By definition of $\underset{\mathit{fld}}{\longmapsto}$ and $\mathtt{t}'''[\ell]$ in step 14, $\mathtt{t}_2[\ell] \underset{\mathit{fld}}{\longmapsto} \mathtt{t}'''[\ell]$, and

    By definition of $\underset{\mathit{tpl}}{\longmapsto}$ and $\mathtt{t}'''$, $\mathtt{t}_2 \underset{\mathit{tpl}}{\longmapsto} \mathtt{t}'''$.

16. By definition of `F-LambdaBar` applied to σ_{j+3} and $\overline{\Lambda}_{j+3}$, and by definition of $\mathtt{t}''$, $\mathtt{t}'''$, and λ'',

    $\exists \sigma_{j+4} \mid \sigma_{j+4} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where

    $\sigma_{j+4}.\overline{\mathcal{A}} = (\sigma_{j+3}.\overline{\mathcal{A}} - \{\mathtt{t}''\}) \bigcup \{\mathtt{t}'''\}$,

    $\sigma_{j+4}.\overline{\mathcal{T}} = \sigma_{j+3}.\overline{\mathcal{T}}$,

    $\sigma_{j+4}.\overline{\mathcal{P}} = \emptyset$,

    $\sigma_{j+4}.\sigma_{next}$ is initially undefined.

    $\exists \overline{\Lambda}_{j+4} \mid \overline{\Lambda}_{j+4} = \overline{\Lambda}_{j+3} - \{\lambda''\}$

17. By definition of $\rightarrowtail_{pL}$, and by steps 14 and 16, where $\sigma_{j+3}.\sigma_{next} = \sigma_{j+4}$ and $K^{pL}_{j+4} = \langle \sigma_{j+4}, \overline{\Lambda}_{j+4} \rangle$.

    $K^{pL}_{j+3} \ \rightarrowtail_{pL} \ K^{pL}_{j+4}$ is one legal transition.

18. By definition of $\underset{cfg}{\longmapsto}$ and σ_{j+4}, and by steps 15, and 16, where respectively, $\mathbf{t}_2 \ \underset{tpl}{\longmapsto} \ \mathtt{t}'''$, and $\mathtt{t}' \in \sigma_{j+4}.\overline{\mathcal{A}}$,

    $K^{TS}_{i+1} \ \underset{cfg}{\longmapsto} \ K^{pL}_{j+4}$.

19. From the transitions in steps 6, 9, 13, and 17, and by the configuration equivalence in step 18, we demonstrated the ability of paraDOS for Linda to perform in $n = 4$ tranitions the *TSspec* computational step of **Tuple Removal**.

Local Evaluation:

$$\frac{\mathbf{p}' \ \rightarrowtail_{p} \ \mathbf{p}''}{\{\!|\ \mathbf{t}[\mathbf{p}' : \tau]\ |\!\} \ \rightarrowtail_{ts} \ \{\!|\ \mathbf{t}[\mathbf{p}'' : \tau]\ |\!\}} \qquad \frac{\mathbf{ts}' \ \rightarrowtail_{ts} \ \mathbf{ts}''}{\mathbf{ts} \uplus \mathbf{ts}' \ \rightarrowtail_{ts} \ \mathbf{ts} \uplus \mathbf{ts}''}$$

1. Let $K^{TS}_{i} \supseteq \{\!|\ \mathbf{t}[\mathbf{p}' : \tau]\ |\!\}$ and $K^{TS}_{i+1} \supseteq \{\!|\ \mathbf{t}[\mathbf{p}'' : \tau]\ |\!\}$, where $K^{TS}_{i} \ \rightarrowtail_{ts} \ K^{TS}_{i+1}$.
   Let $\mathbf{t}_1$ denote tuple $\mathbf{t}$ before, and $\mathbf{t}_2$ denote tuple $\mathbf{t}$ after, transition $\rightarrowtail_{ts}$.

2. Given $K^{TS}_{i} \ \underset{cfg}{\longmapsto} \ K^{pL}_{j}$, by definition of $\underset{cfg}{\longmapsto}$,

   $\exists \mathtt{t} \in \sigma_j.\overline{\mathcal{A}} \mid \mathbf{t}_1 \ \underset{tpl}{\longmapsto} \ \mathtt{t}$,

   and by definition of $\underset{tpl}{\longmapsto}$, $\exists \ell \mid \mathbf{t}_1[\mathbf{p}' : \tau] \ \underset{fld}{\longmapsto} \ \mathtt{t}[\ell]$,

   and by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_1[\ell] = \mathtt{t}[\ell].\mathtt{contents}$.

3. By definition of $\rightarrowtail_{ts}$, the meaning of $\{\!|\ \mathbf{t}_1[\mathbf{p}' : \tau]\ |\!\}$ yields $\{\!|\ \mathbf{t}_2[\mathbf{p}'' : \tau]\ |\!\}$.
   By definition of $\mathtt{Lm\text{-}comp}()$, $\exists \mathtt{t}' \in \mathit{tuple} \mid \mathtt{t}'[\ell].\mathtt{contents} = \mathbf{t}_2[\ell] \ \bigwedge$
   $\forall 1 \leq k \neq \ell \leq \#\mathbf{t}.\ \mathtt{t}'[k] = \mathtt{t}[k]$.

4. By definition of $\underset{fld}{\longmapsto}$ and $\mathtt{t}'[\ell]$, $\mathbf{t}_2[\ell] \ \underset{fld}{\longmapsto} \ \mathtt{t}'[\ell]$, and

   By definition of $\underset{tpl}{\longmapsto}$ and $\mathtt{t}'$, $\mathbf{t}_2 \ \underset{tpl}{\longmapsto} \ \mathtt{t}'$.

5. By definition of `G` applied to σ_j and $\overline{\Lambda}_j$, and by definition of `t` and `t'`,
   $\exists \sigma_{j+1} \in \mathit{state} \mid \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where
   $\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}) \bigcup \{\mathtt{t'}\}$,
   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
   $\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,
   $\sigma_{j+1}.\sigma_{next}$ is initially undefined, and
   $\exists \overline{\Lambda}_{j+1} \in \mathit{closureSet} \mid \overline{\Lambda}_{j+1} = \overline{\Lambda}_j$.
6. By definition of $\longrightarrow\!\!\succ_{pL}$, and by steps 3 and 5, where $\sigma_j.\sigma_{next} = \sigma_{j+1}$ and
   $K^{pL}_{j+1} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,
   $K^{pL}_j \;\longrightarrow\!\!\succ_{pL}\; K^{pL}_{j+1}$ is one legal transition.
7. By definition of $\underset{cfg}{\longmapsto}$ and σ_{j+1}, and by steps 4 and 5, where respectively,
   $\mathtt{t}_2 \underset{tpl}{\longmapsto} \mathtt{t'}$ and $\mathtt{t'} \in \sigma_{j+1}.\overline{\mathcal{A}}$,
   $K^{TS}_{i+1} \underset{cfg}{\longmapsto} K^{pL}_{j+3}$.
8. From the transition in step 6, and by the configuration equivalence in step 7, we demonstrated the ability of paraDOS for Linda to perform in $n = 1$ transition the *TSspec* computational step of **Local Evaluation**.

$\therefore$ Statement 1. of theorem is true.

Part 2: Consider each case of transitions in a restricted paraDOS for Linda, i.e. transition density parameter set to one. For each case, demonstrate the equivalent transition in *TSspec*.

λ **Creation:** Communication closure creation

1. Let $K^{pL}_j, K^{pL}_{j+1} \in \mathit{SCSPair} \;\mid\; K^{pL}_j = \langle \sigma_j, \overline{\Lambda}_j \rangle \;\bigwedge\; K^{pL}_{j+1} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$, where $K^{pL}_j \;\longrightarrow\!\!\succ_{pL}\; K^{pL}_{j+1}$.
2. By definition of communication closure creation and $\longrightarrow\!\!\succ_{pL}$,
   $\exists \lambda \in \mathit{asynchCl} \bigcup \mathit{synchCl} \;\mid\; \lambda \notin \overline{\Lambda}_j \;\bigwedge\; \lambda \in \overline{\Lambda}_{j+1}$, and
   $\exists \mathtt{t} \in \mathit{tuple}.\exists \mathtt{k} \;\mid\; \mathtt{t} \in \sigma_j.\overline{\mathcal{A}} \;\bigwedge$
   $C[\;]$ is the evaluation context of `t`[`k`].`contents` with redex `r` $\bigwedge$
   (`linda-prim? r`) is true.

3. Given $K_j^{pL} \underset{cfg}{\longmapsto} K_i^{TS}$, by definition of $\underset{cfg}{\longmapsto}$, $\mathtt{t}$, and K_i^{TS},

   $\exists \mathbf{t} \in \mathbf{Tuple}_\diamond \mid \mathtt{t} \underset{tpl}{\longmapsto} \mathbf{t} \bigwedge \mathbf{t} \in K_i^{TS}$.

4. By definition of $\rightarrowtail_{pL}$, depending on the domain of $\mathtt{r}$, `Lm-prim` applied to $\mathtt{t}$, $\mathtt{k}$, σ_j, and $\overline{\Lambda}_j$ results in one of the following cases:

   (a) Asynchronous ($\mathtt{r} \in \mathit{asynchLPrim}$)

      i. Let $\mathbf{t}_1$ denote $\mathbf{t}$ from K_i^{TS} (we define $\mathbf{t}_2$ later in step 4.a).

      ii. By definition of $\mathtt{r}$, we know $\lambda \in \mathit{asynchCl}$.

      iii. By definition of $\underset{tpl}{\longmapsto}$,
      and by steps 2 and 3, $\mathbf{t}_1[\mathtt{k}] \underset{fld}{\longmapsto} \mathtt{t}[\mathtt{k}]$; and
      by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_1[\mathtt{k}] = \mathtt{t}[\mathtt{k}].\mathtt{contents}$; and
      by definition of asynchronous Linda primitive and λ,
      $\exists \mathbf{t}' \in \mathbf{Tuple}_\diamond \bigcup \mathbf{Tuple}_\square . \exists \mathtt{t}' \in \mathit{tuple} \mid \mathbf{t}' \underset{tpl}{\longmapsto} \mathtt{t}' \bigwedge$

      $$\mathbf{t}_1[\mathtt{k}] = \begin{cases} \mathbf{t}_1[\mathtt{eval}(\mathbf{t}').\mathbf{p} : \tau] & \text{if } \mathtt{lindaprim}(\lambda) = \mathtt{eval}(\mathtt{t}') \\ \mathbf{t}_1[\mathtt{out}(\mathbf{t}').\mathbf{p} : \tau] & \text{if } \mathtt{lindaprim}(\lambda) = \mathtt{out}(\mathtt{t}') \end{cases}$$

      iv. Let $\mathtt{v}$ be the result of reducing $\mathtt{r}$ in $C[\,]$.

      v. By definition of `Lm-prim` and $\mathtt{v}$,
      $\exists \mathtt{t}'' \in \mathit{tuple} \mid \mathtt{t}''[\mathtt{k}].\mathtt{type} = \mathtt{t}[\mathtt{k}].\mathtt{type} \bigwedge$
      $\mathtt{t}''[\mathtt{k}].\mathtt{contents} = C[\mathtt{v}] \bigwedge$
      $\forall 1 \le \ell \ne \mathtt{k} \le \#\mathtt{t}.\ \mathtt{t}''[\ell] = \mathtt{t}[\ell]$.

      vi. By definition of $\rightarrowtail_{pL}$ and `Lm-prim`, $\sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where
      $\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}) \bigcup \{\mathtt{t}''\}$,
      $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
      $\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,
      $\sigma_{j+1}.\sigma_{next}$ is initially undefined.

      vii. Let $\mathbf{t}_2$ denote $\mathbf{t}$ from K_{i+1}^{TS}, then by definition of $\rightarrowtail_{ts}$, K_i^{TS}, $\mathbf{t}_1[\mathtt{k}]$, and $\mathbf{t}'$, one possible transition for K_i^{TS} to K_{i+1}^{TS} is
      $\{\!|\ \mathbf{t}_1[\mathtt{k}]\ |\!\} \rightarrowtail_{ts} \{\!|\ \mathbf{t}_2[\mathbf{p} : \tau], \mathbf{t}'\ |\!\}$.

      viii. By definition of $\rightarrowtail_{ts}$, the transition in step 4.a.vi, and the meaning of $\mathbf{t}_2[\mathtt{k}]$,
      $\mathbf{t}_2[\mathtt{k}] = \mathtt{t}''[\mathtt{k}].\mathtt{contents}$.

      ix. By definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}_2$, and $\mathtt{t}''$,
      $\forall 1 \le \ell \le \#\mathtt{t}''.\mathtt{t}''[\ell] \underset{fld}{\longmapsto} \mathbf{t}_2[\ell]$; and thus
      by definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}_2$, and $\mathtt{t}''$,
      $\mathtt{t}'' \underset{tpl}{\longmapsto} \mathbf{t}_2$.

x. By definition of $\underset{cfg}{\longmapsto}$, K^{pL}_{j+1}, K^{TS}_{i+1}, and by steps 2, 4.a.ii, 4.a.v, 4.a.vi, and 4.a.vii, where respectively, $\lambda \in \overline{\Lambda}_{j+1}$, $\mathtt{t}' \underset{tpl}{\longmapsto} \mathbf{t}'$,
$\mathtt{t}'' \in \sigma_{i+1}.\overline{\mathcal{A}}$,
$\mathbf{t}_2 \in K^{TS}_{i+1}$, and $\mathtt{t}'' \underset{tpl}{\longmapsto} \mathbf{t}_2$,

$$K^{pL}_{j+1} \underset{cfg}{\longmapsto} K^{TS}_{i+1}.$$

xi. From the transition in step 4.a.vi, and by the configuration equivalence in step 4.a.viii, we demonstrated the ability of *TSspec* to perform in $n = 1$ transition the paraDOS for Linda computational step of asynchronous communication closure creation.

(b) Synchronous ($\mathtt{r} \in synchLPrim$)

i. By definition of $\mathtt{r}$, we know $\lambda \in synchCl$.

ii. By definition of $\underset{tpl}{\longmapsto}$, and by steps 2 and 3, $\mathbf{t}[\mathtt{k}] \underset{fld}{\longmapsto} \mathtt{t}[\mathtt{k}]$; and by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}[\mathtt{k}] = \mathtt{t}[\mathtt{k}].\mathtt{contents}$; and
by definition of synchronous Linda primitive and λ,

$$\exists \mathbf{s} \in \mathbf{Tuple}_{\Box}.\exists \mathtt{s} \in tuple \;\mid\; \mathbf{s} \underset{tpl}{\longmapsto} \mathtt{s} \;\bigwedge$$

$$\mathbf{t}[\mathtt{k}] = \begin{cases} \mathbf{t}[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau] & \text{if } \mathtt{lindaprim}(\lambda) = \mathtt{rd}(\mathtt{s}) \\ \mathbf{t}[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau] & \text{if } \mathtt{lindaprim}(\lambda) = \mathtt{in}(\mathtt{s}) \end{cases}$$

iii. By definition of $\mathtt{Lm\text{-}prim}$,

$$\exists \mathtt{t}'' \in tuple \;\mid\; \mathtt{t}''[\mathtt{k}].\mathtt{type} = \text{'Pending} \bigwedge$$
$$\mathtt{t}''[\mathtt{k}].\mathtt{contents} = \mathtt{t}[\mathtt{k}].\mathtt{contents} \bigwedge$$
$$\forall 1 \leq \ell \neq \mathtt{k} \leq \#\mathtt{t}.\ \mathtt{t}''[\ell] = \mathtt{t}[\ell].$$

iv. By definition of $\longrightarrow_{pL}$ and $\mathtt{Lm\text{-}prim}$, $\sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

$\sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}) \bigcup \{\mathtt{t}''\}$,
$\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
$\sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,
$\sigma_{j+1}.\sigma_{next}$ is initially undefined.

v. By definition of $\longrightarrow_{ts}$, since no transition is a legitimate next step,

$$K^{TS}_i \longrightarrow^0_{ts} K^{TS}_i$$

vi. By definition of $\underset{fld}{\longmapsto}$, step 4.b.ii, and the transition in step 4.b.v,

$$\mathbf{t}[\mathtt{k}] = \mathtt{t}[\mathtt{k}].\mathtt{contents} = \mathtt{t}''[\mathtt{k}].\mathtt{contents}$$

vii. By definition of $\underset{fld}{\longmapsto}$, $\mathtt{t}$, $\mathtt{t}''$, and step 4.b.vi,

$\forall 1 \leq \ell \leq \#\mathtt{t}''.\mathtt{t}''[\ell] \underset{fld}{\longmapsto} \mathtt{t}[\ell]$; and thus

by definition of $\underset{tpl}{\longmapsto}$, $\mathtt{t}$, and $\mathtt{t}''$,

$\mathtt{t}'' \underset{tpl}{\longmapsto} \mathtt{t}$.

viii. By definition of $\underset{cfg}{\longmapsto}$, K^{pL}_{j+1}, K^{TS}_{i}, and by steps 2, 4.b.ii, 4.b.iv, 3, and 4.b.vii, where respectively, $\lambda \in \overline{\Lambda}_{j+1}$, $\mathtt{s} \underset{tpl}{\longmapsto} \mathtt{s}$, $\mathtt{t}'' \in \sigma_{i+1}.\overline{\mathcal{A}}$, $\mathtt{t} \in K^{TS}_{i}$, and $\mathtt{t}'' \underset{tpl}{\longmapsto} \mathtt{t}$.

$K^{pL}_{j+1} \underset{cfg}{\longmapsto} K^{TS}_{i}$.

ix. From the transition in step 4.b.v, and by the configuration equivalence in step 4.b.viii, we demonstrated the ability of *TSspec* to perform in $n = 0$ transitions the paraDOS for Linda computational step of synchronous communication closure creation.

5. From steps 4.a.xi and 4.b.ix, we demonstrated, for the case of communication closure creation, that

$$K^{pL}_{j} \rightarrowtail_{pL} K^{pL}_{j+1} \implies$$
$$\exists n \geq 0 \text{ s.t. } K^{TS}_{i} \rightarrowtail^{n}_{ts} K^{TS}_{i+n} \bigwedge K^{TS}_{i+n} \underset{cfg}{\longmapsto} K^{pL}_{j+1}.$$

λ **Reduction:** Consider closure reductions according to whether the closure contains asynchronous or synchronous Linda primitives.

Asynchronous Linda primitives: Closures containing asynchronous Linda primitives reduce in two steps.

λ **Reduction 1:**

1. Let $K^{pL}_{j}, K^{pL}_{j+1} \in \mathit{SCSPair} \mid$
   $K^{pL}_{j} = \langle \sigma_j, \overline{\Lambda}_j \rangle \bigwedge K^{pL}_{j+1} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,
   where $K^{pL}_{j} \rightarrowtail_{pL} K^{pL}_{j+1}$.
2. By definition of $\rightarrowtail_{pL}$, and `reduce-send`,
   $\sigma_{j+1} = \sigma_j$,
   $\exists \lambda \in \mathit{asynchCl} \mid \lambda \in \overline{\Lambda}_j \bigwedge \lambda \notin \overline{\Lambda}_{j+1}$, and
   $\exists \lambda' \in \mathit{asynchLPrim} \mid \lambda' \notin \overline{\Lambda}_j \bigwedge \lambda' \in \overline{\Lambda}_{j+1}$,
   where
   $\mathtt{lindaprim}(\lambda) = \mathtt{lindaprim}(\lambda')$.

3. Given $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$,

   $\exists \mathbf{t} \in \mathbf{Tuple}_{\Box} \bigcup \mathbf{Tuple}_{\diamond}\ .\ \exists \mathtt{t} \in \mathit{tuple}\ \mid$
   $\mathbf{t} \in K_i^{TS}\ \bigwedge\ \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}\ \bigwedge$
   $\mathtt{lindaprim}(\lambda) \in \{\mathtt{eval}(\mathtt{t}), \mathtt{out}(\mathtt{t})\}$

4. By definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}$, $\mathtt{t}$, and step 3, where
   $\mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}$, and

   by definition of λ, λ', and step 2, where
   $\lambda' \in \overline{\Lambda}_{j+1}\ \bigwedge\ \mathtt{lindaprim}(\lambda) = \mathtt{lindaprim}(\lambda')$,

   then by definition of $\underset{cfg}{\longmapsto}$, K_i^{TS}, and K_{j+1}^{pL},
   $K_i^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$.

5. By definition of $\rightarrowtail_{ts}$, since no transition is a legitimate next step,
   $K_i^{TS} \rightarrowtail_{ts}^{0} K_i^{TS}$

6. By the configuration equivalence in step 4, and from the transition in step 5, we demonstrated, for the case of λ **Reduction 1**, that
   $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL} \implies$
   $\exists n \geq 0 \text{ s.t. } K_i^{TS} \rightarrowtail_{ts}^{n} K_{i+n}^{TS}\ \bigwedge\ K_{i+n}^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$,

   where $n = 0$.

λ **Reduction 2:**

1. Let $K_j^{pL}, K_{j+1}^{pL} \in \mathit{SCSPair}\ \mid$
   $K_j^{pL} = \langle \sigma_j, \overline{\Lambda}_j \rangle\ \bigwedge\ K_{j+1}^{pL} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,
   where $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL}$.

2. By definition of $\rightarrowtail_{pL}$, `reduce-eval`, and `reduce-out`,
   $\exists \lambda \in \mathit{asynchLPrim}\ .\ \exists \mathtt{t} \in \mathit{tuple}\ \mid$
   $\lambda \in \overline{\Lambda}_j\ \bigwedge\ \lambda \notin \overline{\Lambda}_{j+1}\ \bigwedge$
   $\mathtt{lindaprim}(\lambda) \in \{\mathtt{eval}(\mathtt{t}), \mathtt{out}(\mathtt{t})\}$,

   and $\sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where
   $\sigma_{j+1}.\sigma_{next}$ is initially undefined, and
   if $\mathtt{lindaprim}(\lambda) = \mathtt{eval}(\mathtt{t})$
   $\sigma_{j+1}.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}} \bigcup \{\mathtt{t}\}$,
   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
   $\sigma_{j+1}.\overline{\mathcal{P}} = \{\langle \mathtt{'Egenerating}, \mathtt{t} \rangle\}$,
   else // $\mathtt{lindaprim}(\lambda) = \mathtt{out}(\mathtt{t})$
   $\sigma_{j+1}.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}}$,
   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}} \bigcup \{\mathtt{t}\}$,
   $\sigma_{j+1}.\overline{\mathcal{P}} = \{\langle \mathtt{'Ecreated}, \mathtt{t} \rangle\}$,

3. Given $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$,

   $\exists \mathbf{t} \in \mathbf{Tuple}_\diamond \bigcup \mathbf{Tuple}_\square \;\mid\; \mathbf{t} \in K_i^{TS} \;\bigwedge\; \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}$

4. By definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}$, $\mathtt{t}$, K_i^{TS}, and step 3, where

   $\mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t} \;\bigwedge\; \mathbf{t} \in K_i^{TS}$,

   and by definition of σ_{j+1} from step 2, where

   $$\mathtt{t} \in \begin{cases} \sigma_{j+1}.\overline{\mathcal{A}} & \text{if } \mathtt{lindaprim}(\lambda) = \mathtt{eval}(\mathtt{t}) \\ \sigma_{j+1}.\overline{\mathcal{T}} & \text{otherwise} \end{cases}$$

   then by definition of $\underset{cfg}{\longmapsto}$, K_i^{TS}, and K_{j+1}^{pL},

   $K_i^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$.

5. By definition of $\rightarrowtail_{ts}$, since no transition is a legitimate next step,

   $K_i^{TS} \rightarrowtail_{ts}^{0} K_i^{TS}$

6. By the configuration equivalence in step 4, and from the transition in step 5, we demonstrated, for the case of λ **Reduction 2**, that

   $$K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL} \implies \exists n \geq 0 \text{ s.t. } K_i^{TS} \rightarrowtail_{ts}^{n} K_{i+n}^{TS} \;\bigwedge\; K_{i+n}^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL},$$

   where $n = 0$.

Synchronous Linda primitives: Closures containing synchronous Linda primitives reduce in three steps.

λ **Reduction 1:** (similar to λ reduction 1 for asynchronous Linda primitives)

1. Let $K_j^{pL}, K_{j+1}^{pL} \in \mathit{SCSPair} \;\mid$

   $K_j^{pL} = \langle \sigma_j, \overline{\Lambda}_j \rangle \;\bigwedge\; K_{j+1}^{pL} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,

   where $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL}$.

2. By definition of $\rightarrowtail_{pL}$, and `reduce-send`,

   $\sigma_{j+1} = \sigma_j$,

   $\exists \lambda \in \mathit{synchCl} \;\mid\; \lambda \in \overline{\Lambda}_j \;\bigwedge\; \lambda \notin \overline{\Lambda}_{j+1}$, and

   $\exists \lambda' \in \mathit{sendCl} \;\mid\; \lambda' \notin \overline{\Lambda}_j \;\bigwedge\; \lambda' \in \overline{\Lambda}_{j+1}$,

   where

   $\mathtt{lindaprim}(\lambda) = \mathtt{lindaprim}(\lambda')$.

3. Given $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$,

   $\exists \mathbf{t} \in \mathbf{Tuple}_\square \,.\, \exists \mathtt{t} \in \mathit{tuple} \;\mid\; \mathbf{t} \in K_i^{TS} \;\bigwedge\; \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t} \;\bigwedge$

   $\mathtt{lindaprim}(\lambda) \in \{\mathtt{rd}(\mathtt{t}), \mathtt{in}(\mathtt{t})\}$

4. By definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}$, $\mathtt{t}$, and step 3, where $\mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}$, and by definition of λ, λ', and step 2, where
   $\lambda' \in \overline{\Lambda}_{j+1} \bigwedge \mathtt{lindaprim}(\lambda) = \mathtt{lindaprim}(\lambda')$,
   then by definition of $\underset{cfg}{\longmapsto}$, K_i^{TS}, and K_{j+1}^{pL},
   $K_i^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$.
5. By definition of $\rightarrowtail_{ts}$, since no transition is a legitimate next step,
   $K_i^{TS} \rightarrowtail_{ts}^{0} K_i^{TS}$
6. By the configuration equivalence in step 4, and from the transition in step 5, we demonstrated, for the case of the first synchronous communication closure reduction, that
   $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL} \Longrightarrow$
   $\exists n \geq 0 \text{ s.t. } K_i^{TS} \rightarrowtail_{ts}^{n} K_{i+n}^{TS} \bigwedge K_{i+n}^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$,
   where $n = 0$.

λ **Reduction 2:**

1. Let $K_j^{pL}, K_{j+1}^{pL} \in \mathit{SCSPair} \mid$
   $K_j^{pL} = \langle \sigma_j, \overline{\Lambda}_j \rangle \bigwedge K_{j+1}^{pL} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,
   where $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL}$.
2. By definition of $\rightarrowtail_{pL}$, `reduce-rd`, and `reduce-in`,
   $\exists \lambda \in \mathit{sendCl} \,.\, \exists \lambda' \in \mathit{reactCl} \,.\, \exists \mathtt{s}, \mathtt{t} \in \mathit{tuple} \mid$
   $\lambda \in \overline{\Lambda}_j \bigwedge \lambda \notin \overline{\Lambda}_{j+1} \bigwedge$
   $\mathtt{lindaprim}(\lambda) \in \{\mathtt{rd}(\mathtt{s}), \mathtt{in}(\mathtt{s})\} \bigwedge$
   $\lambda' \notin \overline{\Lambda}_j \bigwedge \lambda' \in \overline{\Lambda}_{j+1} \bigwedge \lambda' =$ "`react(m,k,t)`" $\bigwedge$
   (`match? s t`) is true $\bigwedge$ $\mathtt{t} \in \sigma_j.\overline{\mathcal{T}}$,
   and $\sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where
   $\sigma_{j+1}.\sigma_{next}$ is initially undefined, and
   if $\mathtt{lindaprim}(\lambda) = \mathtt{rd}(\mathtt{s})$
   $\sigma_{j+1}.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}}$,
   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
   $\sigma_{j+1}.\overline{\mathcal{P}} = \{\langle \mathtt{'Ecopied}, \mathtt{t} \rangle\}$,
   else // $\mathtt{lindaprim}(\lambda) = \mathtt{in}(\mathtt{s})$
   $\sigma_{j+1}.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}}$,
   $\sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}} - \{\mathtt{t}\}$,
   $\sigma_{j+1}.\overline{\mathcal{P}} = \{\langle \mathtt{'Econsumed}, \mathtt{t} \rangle\}$,
3. Given $K_i^{TS} \underset{cfg}{\longmapsto} K_j^{pL}$,
   $\exists \mathbf{s}, \mathbf{t} \in \mathbf{Tuple}_{\square} \mid \mathbf{t} \in K_i^{TS} \bigwedge \mathit{match}(\mathbf{s}, \mathbf{t})$ is true $\bigwedge$
   $\mathbf{s} \underset{tpl}{\longmapsto} \mathtt{s} \bigwedge \mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}$

4. By definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}$, $\mathtt{t}$, K_i^{TS}, and step 3, where

   $\mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t} \bigwedge \mathbf{t} \in K_i^{TS}$,

   and by definition of σ_{j+1} from step 2, where

   $\mathtt{t} \in \sigma_{j+1}.\overline{\mathcal{T}}$,

   then by definition of $\underset{cfg}{\longmapsto}$, K_i^{TS}, and K_{j+1}^{pL},

   $K_i^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$.

5. By definition of $\rightarrowtail_{ts}$, since no transition is a legitimate next step,

   $K_i^{TS} \rightarrowtail_{ts}^{0} K_i^{TS}$

6. By the configuration equivalence in step 4, and from the transition in step 5, we demonstrated, for the case of λ **Reduction 2**, that

   $$K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL} \implies$$
   $$\exists n \geq 0 \text{ s.t. } K_i^{TS} \rightarrowtail_{ts}^{n} K_{i+n}^{TS} \bigwedge K_{i+n}^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL},$$

   where $n = 0$.

λ **Reduction 3:**

1. Let $K^{pL}_j, K^{pL}_{j+1} \in \mathit{SCSPair} \mid$
   $K^{pL}_j = \langle \sigma_j, \overline{\Lambda}_j \rangle \;\bigwedge\; K^{pL}_{j+1} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$,
   where $K^{pL}_j \rightarrowtail_{pL} K^{pL}_{j+1}$.
2. By definition of $\rightarrowtail_{pL}$, `reduce-react`, and synchronous Linda primitives,

   $\exists \lambda \in \mathit{reactCl} \;.\; \exists \mathtt{t} \in \mathit{tuple} \;.\; \exists \mathtt{m}, \mathtt{k} \mid$
   $\quad \lambda \in \overline{\Lambda}_j \;\bigwedge\; \lambda \notin \overline{\Lambda}_{j+1} \;\bigwedge\; \lambda = \text{“}\mathtt{react(m,k,t)}\text{”} \;\bigwedge$
   $\quad \exists \mathtt{t_m}, \mathtt{s} \in \mathit{tuple} \mid$
   $\quad\quad \mathtt{t_m} \in \sigma_j.\overline{\mathcal{A}} \;\bigwedge\; \mathtt{t_m}[\mathtt{k}].\mathtt{type} = \text{'Pending} \;\bigwedge$
   $\quad\quad C[\;]$ is the evaluation context of $\mathtt{t_m}[\mathtt{k}].\mathtt{contents}$ with redex $\mathtt{r}$, where
   $\quad\quad\quad \mathtt{r} \in \{\mathtt{rd(s)}, \mathtt{in(s)}\} \;\bigwedge$
   $\quad\quad\quad \big((\mathtt{t} \in \sigma_j.\overline{\mathcal{T}} \;\bigwedge\; \mathtt{r} = \mathtt{rd(s)}) \;\bigvee$
   $\quad\quad\quad\quad (\mathtt{t} \notin \sigma_j.\overline{\mathcal{T}} \;\bigwedge\; \mathtt{r} = \mathtt{in(s)})\big) \;\bigwedge$
   $\quad\quad\quad \exists \mathtt{t'_m} \in \mathit{tuple} \mid$
   $\quad\quad\quad\quad \mathtt{t'_m}[\mathtt{k}].\mathtt{type} = \text{'Active} \;\bigwedge$
   $\quad\quad\quad\quad \mathtt{t'_m}[\mathtt{k}].\mathtt{contents} = C[\mathtt{t}] \;\bigwedge$
   $\quad\quad\quad\quad \forall 1 \leq \ell \neq \mathtt{k} \leq \#\mathtt{t}.\; \mathtt{t'_m}[\ell] = \mathtt{t_m}[\ell] \;\bigwedge$
   $\quad\quad\quad\quad \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$, where
   $\quad\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{A}} = (\sigma_j.\overline{\mathcal{A}} - \{\mathtt{t_m}\}) \bigcup \{\mathtt{t'_m}\}$,
   $\quad\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$,
   $\quad\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$,
   $\quad\quad\quad\quad\quad \sigma_{j+1}.\sigma_{next}$ is initially undefined.
3. Given $K^{pL}_j \underset{cfg}{\longmapsto} K^{TS}_i$, by definition of $\underset{cfg}{\longmapsto}$, $\mathtt{t_m}$, $\mathtt{t}$, and K^{TS}_i,

   $\exists \mathbf{t_m} \in \mathbf{Tuple}_{\diamond} \;.\; \exists \mathbf{t} \in \mathbf{Tuple}_{\square} \mid$
   $\quad \mathtt{t_m} \underset{tpl}{\longmapsto} \mathbf{t_m} \;\bigwedge\; \mathtt{t} \underset{tpl}{\longmapsto} \mathbf{t} \;\bigwedge\; \{\mathbf{t_m}, \mathbf{t}\} \subseteq K^{TS}_i$.
4. By definition of $\underset{tpl}{\longmapsto}$, $\mathtt{t_m}$, and $\mathbf{t_m}$ from step 3, and the definition of $\mathtt{k}$ from step 2,

   $\mathbf{t_m}[\mathtt{k}] \underset{fld}{\longmapsto} \mathtt{t_m}[\mathtt{k}]$; and

   by definition of $\underset{fld}{\longmapsto}$, and redex $\mathtt{r}$ and tuple $\mathtt{s}$ from step 2,

   $\mathbf{t_m}[\mathtt{k}] = \mathtt{t_m}[\mathtt{k}].\mathtt{contents}$; where
   $\exists \mathbf{s} \in \mathbf{Tuple}_{\square} \mid$
   $\quad \mathtt{s} \underset{tpl}{\longmapsto} \mathbf{s} \;\bigwedge \mathit{match}(\mathbf{s}, \mathbf{t})$ is true $\bigwedge$

   $$\mathbf{t_m}[\mathtt{k}] = \begin{cases} \mathbf{t_m}[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau] & \text{if } \mathtt{r} = \mathtt{rd(s)} \\ \mathbf{t_m}[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau] & \text{otherwise // } \mathtt{r} = \mathtt{in(s)} \end{cases}$$

5. By definition of $\rightarrowtail_{ts}$, and $\mathbf{t}_{\mathtt{m}}[\mathtt{k}]$ from step 4, one possible transition for $K_i^{TS} \rightarrowtail_{ts} K_{i+1}^{TS}$ is either
   $\{\!|\ \mathbf{t}_{\mathtt{m}}[\mathtt{rd}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}\ |\!\} \rightarrowtail_{ts} \{\!|\ \mathbf{t}'_{\mathtt{m}}[\mathbf{p}(\mathbf{t}) : \tau],\ \mathbf{t}\ |\!\}$, or
   $\{\!|\ \mathbf{t}_{\mathtt{m}}[\mathtt{in}(\mathbf{s}).\mathbf{p} : \tau],\ \mathbf{t}\ |\!\} \rightarrowtail_{ts} \{\!|\ \mathbf{t}'_{\mathtt{m}}[\mathbf{p}(\mathbf{t}) : \tau]\ |\!\}$.
6. By definition of $\mathtt{t}'_{\mathtt{m}}[\mathtt{k}].\mathtt{contents}$ and $\mathbf{t}'_{\mathtt{m}}[\mathtt{k}]$, from steps 2 and 5, where
   $\mathtt{t}'_{\mathtt{m}}[\mathtt{k}].\mathtt{contents} = C[\mathtt{t}]\ \bigwedge$
   $\mathbf{t}'_{\mathtt{m}}[\mathtt{k}] = \mathbf{p}(\mathbf{t})\ \bigwedge$
   $C[\mathtt{t}] = \mathbf{p}(\mathbf{t})$,
   then by definition of $\underset{fld}{\longmapsto}$, since $\mathtt{t}'_{\mathtt{m}}[\mathtt{k}].\mathtt{contents} = \mathbf{t}'_{\mathtt{m}}[\mathtt{k}]$,
   $\mathbf{t}'_{\mathtt{m}}[\mathtt{k}] \underset{fld}{\longmapsto} \mathtt{t}'_{\mathtt{m}}[\mathtt{k}]$
7. By definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}'_{\mathtt{m}}$, $\mathtt{t}'_{\mathtt{m}}$, and by steps 5 and 6,
   $\forall 1 \leq \ell \leq \#\mathtt{t}'_{\mathtt{m}}.\ \mathtt{t}'_{\mathtt{m}}[\ell] \underset{fld}{\longmapsto} \mathbf{t}'_{\mathtt{m}}[\ell]$; and thus
   by definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}'_{\mathtt{m}}$, $\mathtt{t}'_{\mathtt{m}}$,
   $\mathtt{t}'_{\mathtt{m}} \underset{tpl}{\longmapsto} \mathbf{t}'_{\mathtt{m}}$.
8. By definition of $\underset{cfg}{\longmapsto}$, K_{j+1}^{pL}, K_{i+1}^{TS}, and by steps 2, 5, and 7, where respectively, $\mathtt{t}'_{\mathtt{m}} \in \sigma_{j+1}.\overline{\mathcal{A}}$, $\mathbf{t}'_{\mathtt{m}} \in K_{i+1}^{TS}$, and $\mathtt{t}'_{\mathtt{m}} \underset{tpl}{\longmapsto} \mathbf{t}'_{\mathtt{m}}$; and by step 2, where $\mathtt{t} \in \sigma_{j+1}.\overline{\mathcal{T}}$ iff $\mathtt{r} = \mathtt{rd}(\mathtt{s})$, otherwise $\mathtt{t} \notin \sigma_{j+1}.\overline{\mathcal{T}}$, and by step 3, where $\mathbf{t} \underset{tpl}{\longmapsto} \mathtt{t}$,
   $K_{j+1}^{pL} \underset{cfg}{\longmapsto} K_{i+1}^{TS}$.
9. From the transition in step 5, and by the configuration equivalence in step 8, we demonstrated, for the case of λ **Reduction 3**, that
   $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL} \Longrightarrow$
   $\exists n \geq 0$ s.t. $K_i^{TS} \rightarrowtail_{ts}^{n} K_{i+n}^{TS}\ \bigwedge\ K_{i+n}^{TS} \underset{cfg}{\longmapsto} K_{j+1}^{pL}$,
   where $n = 1$.

Internal Computation: (involves no communication closures)

1. Let $K_j^{pL}, K_{j+1}^{pL} \in SCSPair \mid K_j^{pL} = \langle \sigma_j, \overline{\Lambda}_j \rangle\ \bigwedge\ K_{j+1}^{pL} = \langle \sigma_{j+1}, \overline{\Lambda}_{j+1} \rangle$, where $K_j^{pL} \rightarrowtail_{pL} K_{j+1}^{pL}$.

2. By definition of $\longrightarrow\!\!\succ_{pL}$, `Lm`, and `Lm-comp`,

   $\overline{\Lambda}_{j+1} = \overline{\Lambda}_j \ \bigwedge$

   $\exists \mathtt{t} \in \mathit{tuple}\ .\ \exists \mathtt{k}\ \mid$

   $\quad \mathtt{t} \in \sigma_j.\overline{\mathcal{A}}\ \bigwedge\ \mathtt{t}[\mathtt{k}].\mathtt{type} = \text{'Active} \bigwedge$

   $\quad \exists \mathtt{t}' \in \mathit{tuple}\ \mid$

   $\quad\quad \mathtt{t}'[\mathtt{k}].\mathtt{type} \in \{\ \text{'Active, 'Passive}\ \}\ \bigwedge$

   $\quad\quad \forall 1 \leq \ell \neq \mathtt{k} \leq \#\mathtt{t}.\ \mathtt{t}'[\ell] = \mathtt{t}[\ell]\ \bigwedge$

   $\quad\quad \sigma_{j+1} = \langle \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma_{next} \rangle$ where

   $\quad\quad\quad \sigma_{j+1}.\sigma_{next}$ is initially undefined, and

   $\quad\quad\quad$ if $\exists \ell\ \mid\ \mathtt{t}'[\ell].\mathtt{type} = \text{'Active}$

   $\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\})\ \bigcup\ \{\mathtt{t}'\}$

   $\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}$

   $\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{P}} = \emptyset$

   $\quad\quad\quad$ else

   $\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{A}} = \sigma_j.\overline{\mathcal{A}} - \{\mathtt{t}\}$

   $\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{T}} = \sigma_j.\overline{\mathcal{T}}\ \bigcup\ \{\mathtt{t}'\}$

   $\quad\quad\quad\quad \sigma_{j+1}.\overline{\mathcal{P}} = \{\langle\, \mathtt{'Egenerated,\ t'} \rangle\}$

3. Given $K^{pL}_j \underset{cfg}{\longmapsto} K^{TS}_i$, by definition of $\underset{cfg}{\longmapsto}$, $\mathtt{t}$, and K^{TS}_i,

   $\exists \mathbf{t} \in \mathbf{Tuple}_{\diamond}\ \mid\ \mathtt{t} \underset{tpl}{\longmapsto} \mathbf{t}\ \bigwedge\ \mathbf{t} \in K^{TS}_i$.

4. By definition of $\underset{tpl}{\longmapsto}$, and by steps 2 and 3, $\mathbf{t}[\mathtt{k}] \underset{fld}{\longmapsto} \mathtt{t}[\mathtt{k}]$; and
   by definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}[\mathtt{k}] = \mathtt{t}[\mathtt{k}].\mathtt{contents}$; and
   by definition of internal computation,

   $\mathbf{t}[\mathtt{k}] = \mathbf{t}[\mathbf{p}' : \tau]$

5. By definition of $\longrightarrow\!\!\succ_{ts}$, K^{TS}_i, $\mathbf{t}[\mathtt{k}]$, and $\mathbf{t}'$, one possible transition for K^{TS}_i to K^{TS}_{i+1} is

   $\{\!|\ \mathbf{t}[\mathbf{p}' : \tau]\ |\!\}\ \longrightarrow\!\!\succ_{ts}\ \{\!|\ \mathbf{t}'[\mathbf{p}'' : \tau]\ |\!\}$.

6. By definition of $\longrightarrow\!\!\succ_{ts}$, the transition in step 5, and the meaning of $\mathbf{t}'[\mathtt{k}]$,

   $\mathbf{t}'[\mathtt{k}] = \mathtt{t}'[\mathtt{k}].\mathtt{contents}$.

7. By definition of $\underset{fld}{\longmapsto}$, $\mathbf{t}'$, and $\mathtt{t}'$,

   $\forall 1 \leq \ell \leq \#\mathtt{t}'.\ \mathtt{t}'[\ell] \underset{fld}{\longmapsto} \mathbf{t}'[\ell]$; and thus

   by definition of $\underset{tpl}{\longmapsto}$, $\mathbf{t}'$, and $\mathtt{t}'$,

   $\mathtt{t}' \underset{tpl}{\longmapsto} \mathbf{t}'$.

8. By definition of $\underset{cfg}{\longmapsto}$, K^{pL}_{j+1}, K^{TS}_{i+1}, and by steps 2, 5, and 7, where respectively, $\mathtt{t}' \in \sigma_{j+1}.\overline{\mathcal{A}} \bigcup \sigma_{j+1}.\overline{\mathcal{T}}$, $\mathbf{t}' \in K^{TS}_{i+1}$, and $\mathtt{t}' \underset{tpl}{\longmapsto} \mathbf{t}'$,
   $K^{pL}_{j+1} \underset{cfg}{\longmapsto} K^{TS}_{i+1}$.
9. From the transition in step 5, and by the configuration equivalence in step 8, we demonstrated, for the case of internal computation, that
   $K^{pL}_{j} \rightarrowtail_{pL} K^{pL}_{j+1} \Longrightarrow$
   $\exists n \geq 0$ s.t. $K^{TS}_{i} \rightarrowtail^{n}_{ts} K^{TS}_{i+n} \bigwedge K^{TS}_{i+n} \underset{cfg}{\longmapsto} K^{pL}_{j+1}$,
   where $n = 1$.

$\therefore$ Statement 2. of theorem is true.

Since Statement 1 and Statement 2 of the theorem are both true for all respective cases, we conclude Theorem 2 is true.

6.4.4 Beyond the Equivalence

An operational semantics describes how computation proceeds, at a level of abstraction appropriate for reasoning about that computation. Our goals for developing paraDOS for Linda are different from those of Jensen, so it is not surprising that our models are at different levels of abstraction. In particular, Jensen points out that *TSspec* does not specify precisely what constitutes a Linda program, and thus it is not a terminal transition system. We would like to reason about notions such as computation begins, computation ends, and computational quiescence with paraDOS for Linda, and therefore the notion of what constitutes a Linda program is an integral part of our level of abstraction.

Jensen's view of the Linda concept, subtley different from that of Carriero and Gelernter [CG89], is that it extends a host language with Linda primitives, resulting in a model for parallelism through the notion of tuple space. The reference to *host* language intentionally does not preclude the existence of other native interprocess communication capability, enabling processes to interact in ways other than via Linda primitives. But we know from Gelernter [Gel85] that a Linda program is *a collection of ordered tuples*, and from Carriero and Gelernter [CG89], that *if two processes need to communicate, they don't exchange messages or share a variable*, they communicate via tuple space. Thus, while the goals of paraDOS for Linda differ from the goals of *TSspec*, the assumptions of our operational semantics, in particular our notion of what is a Linda program, are consistent with the original Linda concept. The semantics of paraDOS for Linda are at a level of abstraction not limited to reasoning about the transitions of individual Linda processes, but about entire Linda programs.

CHAPTER 7

Parameters of paraDOS

One of the most important aspects of paraDOS is that it is a general, parameterized model; but what may not be clear yet is the nature of paraDOS parameters. Almost every element in paraDOS is a parameter. This is not to imply that paraDOS can become any existing model of computation, simply by specifying the right combination of parameter values. We designed paraDOS to support view-centric reasoning about properties of concurrent computation. Details of views of computation can differ from one system to another, but the approach to reasoning about computation through the multiple perspectives of possibly imperfect observers is the essence of paraDOS. The parameters provide the structure and rules that define the properties for each paraDOS instantiation.

If one wishes to reason about a particular concurrent system using paraDOS, the first step is to determine the appropriate level of abstraction; that is, deciding what observable events are possible. For example, events possible for the Actors model include messages sent and delivered; for Linda, events include tuple creation, consumption, etc. Even though the set of observable events is a parameter, and can vary widely, we do restrict internal computational events from eligibility. Internal computation is too granular, and thus not an appropriate level of abstraction for the computational properties about which we wish to reason with paraDOS. Interprocess communication and coordination are the broad criteria from which observable events may be chosen. This is consistent with computability theory, where only input/output behavior is observable; communication is one instance of input/output behavior.

7.1 The Model System

With one small syntactic change, we tranform the definition of our model system $\overline{\mathcal{S}}$ from a tuple to a grammar, and with the grammar for $\overline{\mathcal{S}}$, we propose a general model for composition. Recall from Chapter 4 the original definition of $\overline{\mathcal{S}}$ is a 3-tuple $\langle\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\rangle$. Now suppose we define a parameterized grammar with nonterminals $\{\overline{\mathcal{S}}, \sigma, \overline{\Lambda}, \overline{\Upsilon}\}$, terminals $\{'\langle','\rangle'\}$, and the production $\overline{\mathcal{S}}\ \rightarrow\ \langle\ \overline{\mathcal{S}}^{*}\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle$, where nonterminals σ, $\overline{\Lambda}$, and $\overline{\Upsilon}$ are parameters that depend on the instance of paraDOS. This production generates strings that can be represented as composition trees. Tuples contained in generated strings are delimited by $'\langle'$ and $'\rangle'$. The recursive production permits zero or more tuples to be nested within any such tuple.

The specification of our model system $\overline{\mathcal{S}}$ is, in fact, a parameter. For instances of paraDOS without composition, the 3-tuple specification of $\overline{\mathcal{S}}$ suffices. For instances of paraDOS that support composition, we propose a grammar consisting of just one production; other grammars with more complex productions are possible. Our production, $\overline{\mathcal{S}}\ \rightarrow\ \langle\ \overline{\mathcal{S}}^{*}\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle$, generates strings that represent n-ary trees, but more and less restrictive representations of composition are possible. The degenerate case of our production reduces to the 3-tuple specification of $\overline{\mathcal{S}}$; the simplest form of composition is no composition. Composition is such an important parameter and broad topic that we devote an entire chapter — Chapter 8 — to this topic.

7.2 Configurations and Computation Space

Defining any operational semantics requires defining what a computational state looks like, and defining a transition relation. Defining paraDOS is no exception. We gave definitions for state σ for the Actors and Linda instances of paraDOS; other instances will require new definitions. The elements of σ should represent the individual computational processes, abstracting away details of their internal computations; their interprocess communications, at an appropriate level of abstraction, from which the set of parallel events can be derived; the set of parallel events $\overline{\mathcal{P}}$; and the recursive next state σ, to be lazily evaluated. Again, even though the computation space σ is a parameter, it is restricted to be in the form of a lazy, n-ary tree. At each level of tree σ, the recursive next state σ to be elaborated represents the child node, chosen by the transition relation, to which computation proceeds.

7.3 Communication Closures

ParaDOS parameterizes communication through its set of communication closures, $\overline{\Lambda}$. In general, closures within $\overline{\Lambda}$ represent instances of communication or coordination necessary for a concurrent system to accomplish its computation. Each time a closure is reduced, as part the transition relation's activity, it is one discrete step closer to completion. The nature of each reduction, and the total number of reductions, for a closure to reduce fully depends on the system we wish to model. For example, actors and Linda processes communicate and coordinate according to very different paradigms, yet the abstraction of a communication closure is capable of modeling both forms of communication. In fact, we focused on message passing models first, then turned to generative communication. We didn't consider shared memory models because of known scalability issues. However, communication closures can be instantiated to model shared memory reads and writes, if we desire to reason about such systems. This capability of paraDOS revealed itself only after we identified communication closures as a parameter.

One approach to model a shared memory system is to map the shared memory to a tuple space. Briefly, the state contains a set of processes (continuations), a set of communication closures (reads and writes), a set of passive tuples, a parallel event set, and a next state. A shared memory system, unlike a tuple space, has a common memory model. To accommodate this, our tuple contains two fields: address and contents. The address field of a tuple is the address of its corresponding block of memory. The contents field is an array of bytes, of size corresponding to the blocksize of the shared memory. All tuple matching is explicit on tuple address fields. The closure for a memory read corresponds to that of a Linda `rd`() primitive. The closure for a memory write corresponds to that of the sequential composition of two Linda primitives, `in`() and `out`(), where `in`() and `out`() remove and replace tuples with identical address fields. Issues relating to mutual exclusion and race conditions are well-represented with such an instantiation of paraDOS. This approach also reflects writes taking more time than reads to complete, due to the number of reductions required by the respective communication closures. There is an implicit transaction semantics on the low-level Linda `in`()/`out`() operations relative to the high-level shared memory “read” and “write” operations. Further ideas on modeling transactions are discussed in Section 10.2, Future Work.

7.4 Transition Relation

The transition relation is a parameter. Actually, the transition relation represents the composition of several parameters. It relies on the meaning function(s) of native language(s), definitions of σ and $\overline{\Lambda}$, system feature specifications, and scheduling policies. Together, the parameters of the transition relation describe behavior we wish to model in an instance of paraDOS.

For example, scheduling policies influence the choice of a next computational state within σ, by encapsulating nondeterministic behavior among processes and communication closures. Other parameters, such as transition density, bound levels of parallelism. Transition density could specify a threshold for the number of parallel events permitted in a single transition, or even specify choosing a transition with the maximum number of parallel events from all possible states.

For the remainder of this section, we focus on native features of distributed systems about which we wish to reason. The first interesting feature is *transactions*. For our purposes, a transaction is the composition of more than one event into a single, atomic event. The semantics of a transaction prescribe that a transaction occurs only when all the individual events that comprise it occur; there are no partial transactions. The implication for distributed systems that endeavor to implement transactions as a native feature is that some sort of rollback-commit or replication strategy is required. Transactions are either supported or not supported by a distributed system.

The next interesting native feature we consider is *messaging type*: either asynchronous or synchronous, or both. The mechanics of asynchronous and synchronous messages are different, but both are computationally equivalent since each can simulate the other. In the case of asynchronous messages, the sender of a message may immediately resume further processing, regardless of whether the receiver has received the message. In the case of synchronous messages, the sender of a message must wait until the receiver receives (or receives *and* processes, as is the case for Java RMI) the message before the sender may resume processing.

Another important characterization of a distributed system is whether it employs a messaging *intermediary*, and if so, what kind of role the intermediary plays: active or passive. In a distributed framework that does not use intermediaries, processes exchange handles with each other to facilitate direct communication. An active intermediary assumes the role of a process in its own right, is generally known to all other executing processes that have a need to communicate, and is often responsible for a sent message's routing and delivery. A passive intermediary usually takes the form of a common, shared memory space accessible by the system's executing processes, according to some agreed-upon protocol.

There are three types of message *destination* characteristics we wish to reason about in distributed systems: one-to-one, one-to-any, and one-to-many. We do not consider broadcast messages because they are not scalable in the broadest sense, but more importantly because they can be modeled computationally by one-to-any messages. In distributed systems that support one-to-one messages, the sender knows the handle of its message's intended recipient, and has the capability to invoke a send command passing some message specifically to that recipient. One-to-any messages can be viewed like a blackboard architecture, where the sender writes a message on the blackboard (some shared memory), and zero or more receivers may read the message from the board if interested. One of these receivers may even erase the board, thus preventing further receipt of that message. With one-to-many messages, the sender need not have any knowledge of the receivers. One-to-many messages are messages that are sent by a designated sender to a group of designated receivers. In the case of an active message intermediary, the sender need not know who the receivers are, but someone — the intermediary — must keep track of who the receivers are for a message to be considered one-to-many.

Some distributed systems (e.g., HLA's RTI) provide a native *publish-subscribe* service for interprocess communication. Under this approach, a process assumes the role of publisher or subscriber of an event; a process can assume different roles for different events. This is another form of anonymous messaging, since a publisher need not know its subscribers, nor a subscriber know who is the publisher.

Finally, we are interested in aspects of message *delivery*, including guaranteed and order-preserving. The guarantee of message delivery is self-explanatory. The order-preserving feature also has a straight-forward meaning, relative to each sender: the order messages are received is the same as the order in which they are sent.

7.5 Views

The set of views $\overline{\Upsilon}$ is not a parameter per se, but subject to other paraDOS parameters. Specifically, each view is a list of ROPEs, so the choice of observable events for an instance of paraDOS ultimately affects the base elements of views. The view function, however, is a parameter. We previously describe the view function as capable of generating all possible views of a computation, but it could be more restrictive. Reasons for restricting possible views depend on what we wish to model. For example, we may wish to reason about security or filtering, and restrict views of certain events. We may wish to reason about network reliability, and set some threshold or probability for communication failures. Despite simultaneity of occurrence in a parallel event set, we may wish to impose on the views of computation different orderings of events, for example, causal only, or total and causal. Similar to but separate from the transition function, the view function in general represents the composition of possibly many policies.

CHAPTER 8

Composition within paraDOS

This chapter addresses composition, an essential element of distributed computation. Section 8.2 discusses composition briefly, then extends paraDOS with general support for composition. Sections 8.3 and 8.4 describe composition within the Linda and Actors instances of paraDOS, respectively.

8.1 Evolution

There is no mention of communication closures in our presentation of the Actors instantiation of paraDOS in Chapter 5. Actor machines influence other actor machines' behaviors by sending tasks. The notion of incoming and outgoing messages exists, dependent upon whether the recipient of a task existed in the set of local actor mail queues. We introduce functions $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$ to route incoming and outgoing tasks to and from the environment, respectively. References to the environment are the only mention of the composition possible for the Actors instance of paraDOS. Our approach is legitimate, as further revealed by Theorem 1 and the accompanying equivalence proof.

The P^{act} approach to handling incoming and outgoing tasks through functions $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$ is decidedly focused on point to point communication. We abandoned this level of abstraction entirely in the next instance of paraDOS — for Linda. One reason for abandoning $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$ is the nature of a single, global tuple space; a passive container through which all distributed processes can communicate. The notion of incoming and outgoing tasks, or more generally, messages, no longer applies, since tuples have no intended recipients, but rather are placed into tuple space. Further, tuples are not delivered to recipients, but rather are matched to a template within tuple space. Finally, the nature of a global tuple space precludes any concept of external messaging. Thus, in the set theoretic semantics for $\mathcal{P}^{\mathbf{TS}}$ presented in Section 6.2, there is no analog to $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$.

Next, we sought to model composition in paraDOS, and could choose from two existing instances as a starting point, P^{act} or $\mathcal{P}^{\mathbf{TS}}$, or begin from a new instance. Another decision was to consider homogeneous composition first, before attempting to model heterogeneous composition. We decided to model tuple space composition, largely because commercial tuple space implementations were growing in popularity, and these implementations were based on multiple tuple spaces.

We made another important decision at this time to reexpress the $\mathcal{P}^{\mathbf{TS}}$ semantics in the functional language Scheme. This decision was not motivated by any limitations in the set theoretic semantics, but rather a desire to gain a stronger intuition into how $\mathcal{P}^{\mathbf{TS}}$ could be implemented. Also, operational semantics permits the choice of level of abstraction, which includes the expression of the semantics itself. Among the benefits of using Scheme was the language's support for closures.

The semantics of Linda primitives with explicit tuple space handles led to wrapping the primitive expressions in closures, along with their corresponding

$$
\begin{aligned}
\overline{\mathcal{S}}_1 &\Rightarrow \langle\, \overline{\mathcal{S}}_2\, \overline{\mathcal{S}}_3\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \overline{\mathcal{S}}_4\, \overline{\mathcal{S}}_5\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\ \overline{\mathcal{S}}_3\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \overline{\mathcal{S}}_4\, \overline{\mathcal{S}}_5\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\ \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_4\ \overline{\mathcal{S}}_5\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\ \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1 \\
&\Rightarrow \langle\, \langle\, \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_4\ \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_5\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_2\ \langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_3\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle_1
\end{aligned}
$$

Figure 8.1: Example derivation for $\overline{\mathcal{S}}$

handles. Each closure explicated the routing requirements for a Linda primitive based on the primitive's tuple space handle and the handle of the tuple space from which the primitive was issued. Since paraDOS is a model for parallel and distributed computation, we needed an abstraction to support the evaluation of multiple simultaneous Linda primitives, or in general, multiple simultaneous communications. This need evolved into the introduction of the set of message closures $\overline{\Lambda}$ in paraDOS.

8.2 A Composition Grammar

Consider the derivation in Figure 8.1. Each nonterminal $\overline{\mathcal{S}}$ is labeled with unique numeric subscripts corresponding to the tuples they derive. The order of derivation is according to the subscripts of the nonterminals. The final string of Figure 8.1 corresponds to the composition tree of Figure 8.2.

The grammar produces two kinds of nodes: leaf nodes and composition (interior) nodes. Leaf nodes look like the old definition of $\overline{\mathcal{S}}$, $\langle\, \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle$; composition nodes are instances of $\langle\, \overline{\mathcal{S}}^{+}\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle$. Technically, composition nodes contain their children nodes. By extension, the root node r of a composition tree contains the entire tree rooted by r. Thus, representation of r as a tree is really an expansion of root node r, whose origin is one possible string generated by our grammar.

Trees are a meaningful abstraction for reasoning about composition. Consider a node $\overline{\mathcal{S}}_i$ within a composition tree. Node $\overline{\mathcal{S}}_i$ is a tuple containing a computation space σ, a set of message closures $\overline{\Lambda}$, and a set of views $\overline{\Upsilon}$. The scope of $\overline{\Lambda}$ and $\overline{\Upsilon}$ is the subtree with root node $\overline{\mathcal{S}}_i$. Now consider a composition tree in its entirety. Since σ, $\overline{\Lambda}$, and $\overline{\Upsilon}$ are parameters, paraDOS can model composition of heterogeneous distributed systems. That is, different leaves of the composition tree may represent different PDS instances, as specified by their respective σ, $\overline{\Lambda}$, and $\overline{\Upsilon}$ parameter values.

One of the advantages of event-based reasoning is the ability – through parameterization – to define common events across heterogeneous systems. Within each leaf node, multiple simultaneous views of its respective local computation are possible, just as is possible in paraDOS without composition. Taking the leaf nodes as an aggregate, though, composition in paraDOS now supports a natural partitioning of parallel event traces, and their respective views. There is not necessarily a temporal relationship between corresponding elements of the computational traces of a composition tree's leaf nodes. Such temporal relationships must be reasoned about using a common composition node.

$$\overline{\mathcal{S}}_1 = \langle\ \overline{\mathcal{S}}_2\ \overline{\mathcal{S}}_3\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle_1$$

$$\overline{\mathcal{S}}_2 = \langle\ \overline{\mathcal{S}}_4\ \overline{\mathcal{S}}_5\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle_2 \qquad \overline{\mathcal{S}}_3 = \langle\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle_3$$

$$\overline{\mathcal{S}}_4 = \langle\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle_4 \qquad \overline{\mathcal{S}}_5 = \langle\ \sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle_5$$

Figure 8.2: Example composition tree from derivation of $\overline{\mathcal{S}}$

Finally, consider the composition nodes. A composition node, like a leaf node, represents the possibility for multiple simultaneous views of its own local computation. Further, since the scope of a composition node c represents that of the entire subtree rooted at c, a subset of events present in c's parallel event trace, and corresponding views, may represent some subset of the aggregate events found in the subtrees of c's children. The extent to which events from the subtrees of node c's children occur in c is itself a parameter. For example, one may wish to compose two or more systems according to certain security policies. Alternatively, or additionally, one may wish to compose systems in a way that allows for relevance filtering to promote greater scalability. In both of these examples, the ability to limit event occurrence in composition nodes through parameterization supports modeling composition at a desirable level of abstraction.

It is difficult to introduce composition without any mention of paraDOS parameters. Similarly, it is difficult to motivate certain parameters without prior mention of composition. We discussed parameters previously in Chapter 7. The remaining sections of the current chapter discuss aspects of composition specific to the tuple space and actors instantiations of paraDOS.

8.3 Tuple space composition

Section 6.3 presents the Scheme-based operational semantics of paraDOS instantiated for Linda and tuple space. We developed the Scheme-based semantics with tuple space composition in mind, and thus extended the Linda syntax to support a tuple space handle prefix for the Linda primitives. Prior to introducing tuple space handles, the Linda primitives assumed a global tuple space. With the introduction of tuple space handles, explicit tuple space references are possible. Two predefined tuple space handles exist, `self` and `parent`, which refer to a process's own tuple space and that of its parent. The `parent` handle is a direct reference to the structure of the paraDOS composition tree, where each node in the tree represents a distinct tuple space.

It is possible for processes to communicate either through common ancestor tuple spaces, or through explicitly referenced tuple spaces. Consider processes p and q that reside in separate tuple spaces within a paraDOS composition tree, such that p and q share a common ancestor tuple space. Process p is able to create a tuple containing its own tuple space handle, and place it directly into its parent tuple space. A process r within p's parent tuple space can then place a tuple containing p's tuple space handle into r's parent tuple space, and so on. Similarly, a process s residing in a common ancestor tuple space to both p and q, can pass down to s's children, and so on, s's tuple space handle. In this way, process q can obtain the handle of ancestor tuple space s, which it shares with process p, and match the tuple directly in s's tuple space that contains p's tuple space handle.

We redefine $\mathcal{P}^{\mathbf{TS}}$ with support for tuple space composition as follows:

$$
\begin{array}{lcl}
\overline{\mathcal{S}} & \rightarrow & \langle\ \overline{\mathcal{S}}^{*}\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\ \rangle \\
\sigma & \rightarrow & \langle\ \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma\ \rangle \\
\overline{\Lambda} & \rightarrow & \textit{as before} \\
\overline{\Upsilon} & \rightarrow & \textit{as before}
\end{array}
$$

Tuple space composition requires a change to `reduce-send`, the part of the transition relation that reduces message closures in $\overline{\Lambda}$. The new version of `reduce-send` differs from the noncompositional version mainly in how and where reduced closures are added to the $\overline{\Lambda}$ set. We assume the existence of helper function `incorp-closure` that, given a tuple space handle and a reduced closure, incorporates the reduced closure into the appropriate tuple space handle's $\overline{\Lambda}$ set. The transition function remains otherwise unchanged. The view function remains completely unchanged, due to paraDOS's separation of computation from the multiple possible views of computation. Figure 8.3 contains the revised `reduce-send` function.

Linda primitives with implicit tuple space references (*i.e.*, no handle prefix) are equivalent to primitives prefixed with `self`. Originally specified Linda programs consist of processes that interact through creating, copying, and removing tuples from a global, shared tuple space via the four Linda primitives. By interpreting these primitives as if they were prefixed with tuple space handle prefix `self`, $\mathcal{P}^{\mathbf{TS}}$ with composition preserves program meaning. This means we don't need to define two versions of `reduce-send`; the version defined in this section works for Linda with or without tuple space composition. We presented the non-compositional version of `reduce-send` with the original $\mathcal{P}^{\mathbf{TS}}$ semantics to reflect Gelernter's definition of a single, shared tuple space [Gel85].

The matter of tuple space composition is of practical importance. Commercial tuple space implementations encourage development of distributed applications that rely on the notion of one *or more* tuple space servers and tuple space handles. For example, the names of Sun's JavaSpaces [FHA99] and IBM's T Spaces [WML98] are both conspicuously plural. Furthermore, both implementations employ handles for tuple space reference, and provide discovery services to promote handle propogation among distributed processes. The instance of paraDOS for Linda with tuple space composition is a first step toward the ability to reason formally about applications developed with commercial tuple space implementations.

```
; reduce-send (with support for TS composition)
; Summary:  returns a new state-LBar pair.  If the closure
; expression to be sent is delayed, strip the delay() and "send"
; by adding to handle's LambdaBar set, using incorp-closure.
; Otherwise, closure is a forced "let" expression.  Farm off to
; reduce-let function.  If reduce-let fails, then reduce-send
; fails, and the original closure is added to returned
; state-LBar's set of closures (where state-LBar's state is
; unchanged).  If reduce-let was successful, the let expression
; bound a tuple into it's delayed subexpression (reactivate).
; reduce-send then "sends" the result to handle's LambdaBar set,
; in the process of returning a new state-LBar pair consisting of
; the subsequent new state and the original LBar.
(define reduce-send
   (lambda (closure state-LBar)
      (let ((send-arg (get-send-arg closure)))
               (handle (get-handle closure))
         (if (delayed?  send-arg)
            (let ((x (incorp-closure handle
                           (strip-delay send-arg))))
               state-LBar)
            ;else forced
            (let ((closure-state
               (reduce-let (strip-force send-arg)
                     state-LBar)))
                  (if (null?  closure-state)
                     ;reduce failed
                     (list (car state-LBar)
                        (union (cadr state-LBar)
                           (singleton closure)))
                     ;else it reduced!
                     (let ((x (incorp-closure handle
                           (car closure-state))))
                        (list (cadr closure-state)
                           (cadr state-LBar)))))))))
```

Figure 8.3: Revised `reduce-send` function to support tuple space composition

8.4 Actors composition discussion

Applying the message closure abstraction to P^{act}, we recognize from composition in $\mathcal{P}^{\mathbf{TS}}$ that routing handle-prefixed Linda primitives between distributed tuple spaces is indeed a form of point to point communication, even though the Linda primitives themselves do not constitute point to point communication between distributed processes. Similarly, in P^{act}, when actor machines create new tasks, they are in fact initiating point to point communication. Thus, the same abstraction we used for $\mathcal{P}^{\mathbf{TS}}$, the message closures set $\overline{\Lambda}$, is meaningful to use for P^{act}, and composition of actor systems. The environment to which $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$ refer, allowing for wrapping tasks in closures, is the closure set $\overline{\Lambda}$ for an actor system's corresponding node in the paraDOS for Actors composition tree. The metaphor for reducing individual message closures represents discrete stages of interprocess communication modeled by paraDOS. For Actors, these stages include the events $\mathbf{E_S}$ (task sent) and $\mathbf{E_D}$ (task delivered), and possibly other intermediate (non-event) reductions.

We redefine P^{act} with support for composition as follows:

$$
\begin{aligned}
&\overline{\mathcal{S}} \rightarrow \langle\, \overline{\mathcal{S}}^{*}\sigma,\ \overline{\Lambda},\ \overline{\Upsilon}\,\rangle\\
&\sigma \rightarrow \langle\, \overline{\mathcal{M}}, \overline{\mathcal{A}}, \overline{\mathcal{T}}, \overline{\mathcal{P}}, \sigma\,\rangle\\
&\overline{\Lambda} \rightarrow \textit{to be defined, dependent on discrete stages of task delivery}\\
&\overline{\Upsilon} \rightarrow \textit{as before for actors}
\end{aligned}
$$

CHAPTER 9

Reasoning

The greatest problem with communication is the illusion it has been accomplished — George Bernard Shaw

This chapter discusses how to use paraDOS to reason about the behaviors of distributed systems. Previously, we identified the need for appropriate levels of abstraction, and introduced concepts important to paraDOS, like observable events, and traces and views of computation. Earlier chapters discuss paraDOS from many different perspectives: paraDOS as a general model of computation, paraDOS instantiated for Actors, paraDOS instantiated for Linda (twice), paraDOS composition trees, and the parameters of paraDOS.

One of the goals of paraDOS is to be able to reason formally about distributed computation, that is, to characterize possible behaviors that result from different approaches to distributed computation. The nondeterminism of multiple communicating distributed processes leads to a potentially intractable combinatorial explosion of possible behaviors. The sources of nondeterminism in a distributed system, and the corresponding policies and protocols in effect, impact the process by which paraDOS constructs traces and views of computation. By considering the sources of nondeterminism in a distributed system, the policies and protocols that govern choice, and the possible traces and views that result, one can utilize paraDOS as a framework to reason about the behavior of instances of distributed computation.

This chapter includes a review of some of the background material already presented in Chapter 2, but with an emphasis on using models to reason about distributed systems. We have attempted to make this presentation self-contained, since reasoning is the primiary purpose of paraDOS. Section 9.1 discusses the early history of formal event-based reasoning, leading up to the study of computability theory. Section 9.2 continues our discussion of computation with the progression of computational models, from sequential to parallel and distributed, and issues important to concurrency. We discuss the CSP approach to representing concurrency in Section 9.3, then introduce properties of computation in Section 9.4. Having provided background for concurrency, CSP, and computational properties, Section 9.5 presents trace-based reasoning about properties of computation with CSP, and motivates the need for paraDOS. Section 9.6 continues the topic of trace-based reasoning with a focus on paraDOS, its abstractions, its extensions to classic CSP, and its focus on scheduling policies and properties of computation for which reasoning with CSP is less well-suited or capable. This chapter concludes in Section 9.7 with a compelling demonstration of the usefulness of paraDOS.

9.1 Early Event-based Reasoning

Theoretical computer science is the formal study of computational models, including, through abstraction, the development of new models and metamodels of computation. Abstraction is the intellectual mechanism for cognitive progression. The inherent need for humans to communicate is manifest throughout our history, from cave drawings and ancient writings to stories and art and music. Each step of our evolution represents an abstraction from one or more steps before.

The development of words and language arose from the need to represent and communicate concepts to one another. The ability to record words, first through drawings and symbols, then alphabets, is a written abstraction of language itself. Early writings, in one sense, reduce to sequences of events, or *traces*, ordered in time. These event traces help preserve the original meanings of stories, enabling humans to understand stories passed down from previous generations. In fact, event-based reasoning is used to understand the past, even that which occurred before recorded history.

Returning to the importance of abstraction, Euclid designed some of the first known numeric algorithms over 2300 years ago. Euclid used the abstraction of a recurrence relation to describe his algorithm to find the greatest common divisor of two integers. Euclid's algorithm is not only an example of a computational abstraction, but its use results in an event trace of sorts. Such a trace is formed from the history of evaluations, a trace of the intermediate, recursive expressions leading to the final solution. The expression corresponding to the initial invocation of the recurrence represents the first event of such an evaluation trace, followed by subsequent expressions (events), each corresponding to the subsequent recurrence invocations, until the base condition of the recurrence is satisfied, resulting in the greatest common divisor itself corresponding to the final event in the trace. An event trace that results from applying Euclid's algorithm represents a computational history. The events abstracted for this example consist of input-output behavior as follows: expressions corresponding to recurrence invocations represent input behavior, the expressions that result from such recurrence invocations represent output behavior, and the intermediate expressions of a recurrence evaluation represent both input and output behavior, since the output expression from the i^{th} recurrence invocation is also the input expression to the $(i+1)^{st}$ recurrence invocation. This abstraction of using input-output behavior to represent computational events emerges again within the study of computability theory.

In the 1930's, the research of logicians Church, Gödel, Kleene, Post, and Turing formed the basis for modern theoretical computer science. Computability theory permits us to distinguish formally between problems for which there are algorithms and those for which there are none [DSW94]. In other words, computability theory is the study of language properties, undecidability, and what problems can and cannot be solved algorithmically. Computability theory promotes its own versions of event-based reasoning. For example, the input-output behavior of Turing machines, with specific instances of output behavior including halting, acceptance, and rejection, is an event-based means of reasoning. Two characteristics of Turing-based computability theory are implicit: computation is sequential and reasoning about computation is based on observable events. Turing-based computability theory does not preclude reasoning about concurrency, but its level of abstraction most naturally focuses on sequential computation.

9.2 Beyond Sequential Computation

New computational paradigms give rise to new classes of models. Without parallel or distributed computation, there is no need to distinguish computation as *sequential*. Classifications of sequential and concurrent computation do not represent a partitioning of computation; rather, there exists a relationship between the two classifications such that concurrent computation subsumes sequential computation. Within the paradigm of event-based reasoning, we can define sequential computation as being restricted to proceeding at most one event at a time, and concurrent computation as permitting zero or more events at a time. Multiple concurrent events suggest multiple concurrent processes, and with concurrency comes the need for communication and coordination among those processes.

A thread of execution refers to the individual path of a computational process. Single-threaded (i.e., sequential) computation consists of an individual computational process. Multi-threaded (i.e., concurrent) computation consists of multiple computational processes. In this sense, sequential computation is the degenerate case of concurrency, where multi-threaded reduces to single-threaded computation.

The concepts of interprocess communication and coordination do not exist when reasoning about sequential computation. These concepts require new, meaningful abstractions to create useful parallel and distributed models of computation. One of these abstractions is that of communication coupling, a term that refers to levels of speed and reliability of communication among threads of execution. Tightly-coupled processes exhibit properties of fast, reliable, interprocess communication behavior. Loosely-couple processes exhibit properties of slower, less reliable, interprocess communication behavior. Parallel computation and distributed computation are special cases of concurrency, each representing opposite ends of a concurrency continuum with respect to their degrees of communication coupling. Parallel computation is composed of tightly-coupled processes; distributed computation is composed of loosely-coupled processes.

Interest in reasoning about concurrency ranges from the desire to take computational advantage of available computer network infrastructures, such as the Internet, to the need for modeling concurrent phenomena in the real world. When

Table 9.1: Examples requiring parallel events

Example Instance	Description
Digital media	Digital media requires the synchronization of video and sound.
Olympic race	Olympic race competitions require detecting false starts (athletes who anticipate the starter's gun), and the final outcome, including the possibility of ties.
Articulated animation	Articulated animation requires concurrent, coordinated movements of arms and legs.
Nuclear missile launcher	A nuclear launch system may require two keys to be turned simultaneously to initiate a launch sequence.
Player piano	A player piano must allow keys to be pressed simultaneously as well as in sequence, to support both chords and musical runs.
Push-button combination lock	A push-button combination may require two buttons be pushed simultaneously as part of its combination.
Troupe movement	Many venues involving coordinated troupe movement exist, including dance productions, military simulations, and gaming environments.
Baseball game	If the runner reaches first base before the ball, he's safe. If the throw to first beats the runner, he's out. In the case of a tie, the runner is safe.

reasoning about events, many real world systems or human endeavors require parallel events. For some examples, see Table 9.1.

9.3 Representing Concurrency

How do we represent concurrency in models of computation? Currently the dominant approach is one developed by C.A.R. Hoare [Hoa85] that treats concurrency as a group of communicating sequential processes. In CSP, an individual process is defined by one or more possible sequences of observable events. CSP represents concurrency via an interleaving of event traces from two or more sequential processes. An idealized observer of computation records the events that occur, one after another, as computation proceeds. It is possible for two or more events to occur simultaneously, in which case the observer records the events in some arbitrary order. Hoare's approach is to ignore simultaneity in this case, since the events must be recorded in some order, and any such order represents a correct partial ordering of computational history. CSP thus employs nondeterministic interleaving to represent the different possibilities introduced by concurrency.

9.4 Properties of Computation

The questions we ask when we reason about computation concern properties of computation. A property of a program is an attribute that is true of every possible history of that program and hence of all executions of the program [And00]. Many interesting program properties fall under the categories of safety, liveness, or some combination of both safety and liveness. A safety property of a program is one in which the program never enters a bad state; nothing bad happens during computation. A liveness property of a program is one in which the program eventually enters a good state; something good eventually happens. Table 9.2 contains some example properties, and their corresponding categories and descriptions.

Table 9.2: Example properties of computation

Property	Category	Description
partial correctness	safety	A program is partially correct if the final state is correct, assuming the program terminates.
termination	liveness	A program terminates if every loop and procedure call terminates; that is, the length of every history is finite.
total correctness	both	Total correctness is a property that combines partial correctness and termination. A program is totally correct if it always terminates with the correct answer.
mutual exclusion	safety	Mutual exclusion is an example of a safety property in a concurrent program. The "bad" state in this case would be one in which two or more processes are executing simultaneous actions within a shared resource's critical section.
finite postponement	liveness	Finite postponement, or eventual entry to a critical section, is an example of a liveness property in a concurrent program. The "good" state for each process is one in which it is executing within its critical section.

Questions arise when reasoning about concurrency that do not otherwise arise in sequential computation. Sequential computation has no notion of critical sections, since a process need not worry about competing for resources with other processes within a given environment. Since critical sections do not exist in sequential computation, there is no need for mutual exclusion, nor any concern for race conditions, deadlock, or infinite postponement. The two properties from Table 9.2 that pertain solely to concurrent systems are mutual exclusion and finite postponement.

The increasingly pervasive Internet, and subsequent demand for Internet applications, appliances, resources, and services, compels us to reason about properties of decentralized, loosely-coupled systems. In this context, loosely-coupled refers to more than communication, it refers more generally to the interoperability of open systems. We are in an age of open systems development. Distributed objects provide protocols, and middleware provides both frameworks and protocols, for heterogeneous n-tier and peer-to-peer application development.

The need to manage shared resources and maintain system integrity in the decentralized environment of Internet applications emphasizes the importance of formal reasoning to describe and verify such complex systems. Indeed, we are concerned with safety and liveness properties of distributed systems. Scheduling policies prescribe how access among competing processes to shared system resources proceeds, based on some criteria. To this end, we are interested in modeling scheduling policies of processes and their respective communications to determine their effect on system properties. Furthermore, given a set of properties that hold for a system, we wish to identify and model varying notions of fairness.

9.5 Reasoning with Traces

Event traces are one possible framework from which to reason about properties of computation. Since a trace of events represents a history of computation, and a property must be true for every possible history of a computational system, a property of a computational system must hold for all possible traces of that system. In Section 9.5.1 we discuss how to reason about computation with CSP traces, then in Section 9.5.2 we discuss limitations of CSP, and motivate the extensions to CSP that paraDOS provides.

9.5.1 Reasoning with CSP

Table 9.3 exhibits some notation for reasoning about properties of computation using CSP. For a complete presentation of this topic, see Hoare [Hoa85]. For the purpose of this discussion, it suffices to elaborate a few points from Table 9.3, and give some examples. Process P is nondeterministic, due to the possible existence of a refusals set (i.e., environments in which P can deadlock). Nondeterminism in this sense represents the ability of a process to exhibit a range of possible bahaviors, with no way to predict these behaviors based on the external environment alone. This form of nondeterminism encourages developing higher levels of abstraction for describing physical behavior. Returning to Table 9.3, predicate S represents a property of computation, which may or may not be true for process P. Instances of predicate S are expressions that may include tr and ref. The meaning of a relation denoted **sat** is that P satisfies S (P **sat** S) if S is true for all possible traces tr and refusals ref of P.

Some examples describing computational properties within CSP are in order. Consider two safety properties: deadlock-free and divergent-free. The property of a process being deadlock-free specifies that a process with alphabet A (an event alphabet) will never stop, thus $NONSTOP = (ref \neq A)$. If P **sat** $NONSTOP$, and if P has an environment that permits all events in A, P must choose to perform one of them. To prove a process does not diverge, we proceed as follows. The CSP definition of a divergent process is one that can do anything and refuse

Table 9.3: Some CSP notation for reasoning about traces

Notation	Meaning
P	A process.
tr	An arbitrary trace of process P.
P/tr	P after (engaging in events of trace) tr.
$traces(P)$	The set of all traces of a process, P.
X	A set of events which are offered initially by the environment of P. X is a *refusal* of P if it is possible for P to deadlock on its first step when placed in this environment.
ref	An arbitrary refusal set of process P.
$refusals(P)$	The set of all refusals of a process, P.
P **sat** $S(tr, ref)$	$\forall tr, ref.\ tr \in traces(P) \bigwedge ref \in refusals(P/tr) \Rightarrow S(tr, ref)$

anything [Hoa85]. Following this definition, if there exists a set that cannot be refused, then the process is not divergent. We define predicate $NONDIV = (ref \neq A)$. Notice $NONSTOP \equiv NONDIV$! This demontrates proving the property absence of divergence requires no more work than proving the absense of deadlock property.

9.5.2 Why paraDOS?

With all the benefits that CSP provides for reasoning about concurrency, including event abstraction and event traces, what motivated the development of paraDOS? For all its elegance, CSP has limitations. In general, the CSP model does not directly represent event simultaneity (i.e., event aggregation). Two exceptions are synchronized events common to two or more interleaved processes, or abstracting a new event to represent the simultaneous occurrence of two or more designated atomic events. CSP does not provide extensive support for imperfect observation; CSP supports event hiding, or concealment, but this approach is event specific and all-or-nothing, which amounts to filtering. Since CSP represents concurrency through an arbitrary interleaving of events, it provides no support for multiple simultaneous views of an instance of computation.

To overcome the limitations to CSP just mentioned, paraDOS extends CSP with the notion of parallel events. Parallel event traces don't require interleaving to represent concurrency. Also, paraDOS replaces CSP's idealized observer with the notion of multiple, possibly imperfect observers. Multiple observers inspire the existence of views of computation. Thus, paraDOS distinguishes a computation's history – its trace – from the multiple possible views of a computation.

ParaDOS differs from CSP in other important ways. CSP is an algebraic model; paraDOS is a parameterized, operational semantics. As an operational semantics, instances of paraDOS require definition of a transition relation to describe how computation proceeds from one state to the next. The notion of state in paraDOS, across instantiations, is essentially composed of processes and communication closures – a potentially unifying characterization of concurrency. Finally, paraDOS introduces, as one of its parameters, the notion of a composition grammar, which may be represented as a tree. The composition grammar is an elegant mechanism for specifying rules of composition across instances of paraDOS.

9.6 Reasoning with paraDOS

This section discusses reasoning about properties of computation with paraDOS. We begin with a review of paraDOS constructs in Section 9.6.1, then discuss features of our model that distinguish it from CSP in Section 9.6.2. Section 9.6.3 discusses the role of policies in the paraDOS transition relation, and gives some examples from Linda. Finally, Section 9.6.4 discusses paraDOS approaches to reasoning about system properties.

9.6.1 ParaDOS Basics

The primitive element for reasoning in paraDOS is the observable event, or just event. An event is a discrete instance of observable behavior at a desired level of abstraction. Briefly, we review the definitions of paraDOS structures built up from these events. A set of events is a parallel event. A list of events selected from a parallel event is a ROPE. A list of parallel events is a trace. A list of ROPEs is a view. Each element of a view of computation, a ROPE, corresponds positionally to a parallel event in that computation's trace. For a given trace, in general, multiple views are possible. The choice of observable events for an instance of paraDOS does not change the definition of parallel event, ROPE, trace, or view.

ParaDOS is an operational semantics whose computation space is a lazy tree from which it is possible to construct parallel event traces from respective instances of computation. For a given trace of computation, paraDOS is capable of generating all possible corresponding views of that computation. A view is a sequentialized partial ordering of an instance of concurrent computation. The structure of a view is that of a list of ROPEs, which is by definition a list of lists of sequential events. Thus, a single perfect view of computation in paraDOS is analogous to a CSP trace; the transformation of a paraDOS view to the form of a CSP trace is straightforward, and described by the Scheme function `flatten`. Given this correspondence of views to CSP traces, it is possible to reason about properties of computation in paraDOS using the same tools and techniques as those from CSP.

9.6.2 Beyond CSP

ParaDOS is not restricted to standard CSP abstractions for reasoning about computation, though we certainly can instantiate paraDOS to be capable of generating event traces like those of CSP, and restrict reasoning about traces to a single view. ParaDOS is capable of generating parallel-event traces and multiple views of a given parallel-event trace, abstractions that don't exist in standard CSP. Multiple views permit reasoning about multiple perspectives of a computation, such as those of users of distributed systems (e.g., discrete event simulations, virtual worlds). Multiple perspectives of a system's computational trace includes the possibility for imperfect observation by design.

The purpose of paraDOS is to provide an overall higher level of abstraction for reasoning about distributed computation, a model that more closely approximates the reality of concurrency. ParaDOS differs in two significant ways from CSP: its traces preserve the concurrency inherent in the history of computation, and its semantics are operational rather than algebraic. CSP imposes the restriction that an idealized observer record arbitrary, sequential partial orderings of simultaneously occurring events, and in so doing, does not preserve event simultaneity. These differences impact reasoning about properties of computation in important ways, as will be demonstrated in Section 9.7.

We introduce one last paraDOS notion for reasoning about properties of computation, the unsuccessful event, or *un-event*. There are two categories of events in paraDOS: successful and unsuccessful. By default, events refer to successful events. The definition of un-event that we are using is, "an attempted computation or communication activity, associated with an event, that fails to succeed." The ability to observe successful and unsuccessful events within the context of parallel events and views permits us to reason directly about nondeterminism and its consequences. Parallel events that include un-events allows us to reason not only about what happened, but also about what might have happened.

CSP has a notion similar to paraDOS un-events that it calls refusal sets. Recall from Table 9.3, that refusal sets represent environments in which a CSP process might immediately deadlock. The notion of refusal sets is from a passive perspective of event observation. Since paraDOS is an operational semantics, our model employs the active notion of event occurrence, where designated computational progress corresponds to the events abstracted. The purpose of refusal sets in CSP and un-events in paraDOS is the same, to support reasoning about properties of concurrent computation.

9.6.3 Policies

We now discuss the implications of parameterized policies as they concern reasoning about properties of concurrent computation. Policies dictate the selection of processes to make computational progress during a transition, and the selection of message closures to be reduced during a transition. Policies are also parameters within a paraDOS transition relation. These parameters specify the sequence in which chosen processes attempt to make computational progress, and the sequence in which selected communication closure reductions attempt to reduce. When we choose policies for the transition relation, we can reason about resulting system behavior, and use paraDOS to prove properties of distributed systems with those policies.

Policies may determine access to critical regions, or specifiy the resolution of race conditions. The outcome of such shared resource scenarios, and the policies that lead to that outcome, influence what views are possible, and meaningful. In determining process and message closure selection, one policy could be pure randomness, and another policy could prioritize according to a particular scheme. The choice of a selection policy impacts the nature of nondeterminism in a concurrent system.

For the Linda instance of paraDOS, transitions from one state of computation to the next consist of individual processes making internal computational progress, or communication closure reductions that lead to instances of tuple space interaction. During each transition, the set of possible next states depends on the current state and the policies of the transition relation.

Consider policies that effect the level of parallelism in a tuple space, including maximal parallelism, minimal parallelism, and levels somewhere in between. A policy of selecting only one Linda process per transition to make computational progress, or one communication closure per transition to reduce, results in singular transition density, or sequential computation. In contrast, a policy that requires selecting every eligible Linda process and every communication closure is part of a set of policies needed to model maximal parallelism. The ability to model all possible transitions in a distributed system requires a policy that selects a random subset of Linda processes and communication closures. Other properties of distributed systems we wish to reason about may limit or bound the level of parallelism possible, for example, based on the number of processors available. ParaDOS permits the specification of appropriate policies for all the levels of parallelism discussed herein.

An important set of policies in tuple space systems concerns different protocols for matching tuples. Tuple matching is a significant source of nondeterminism in Linda programs, and it comes in two varieties. First, two or more matching operations, at least one of which is an `in`(), compete for a single, matching tuple. The second kind of nondeterminism involves just one synchronous primitive, but its template matches two or more tuples. In both cases, the outcome of the subsequent tuple space interactions is nondeterministic, but tuple matching policies can influence system properties. For example, a policy that attempts to match operations with the most specific templates first, and saves matching the most general templates for last, is likely to match more tuples than if the sequence of attempted matches is reversed. Another example of maximizing tuple space interactions would prioritize `out`() operations before any `rd`() and `in`() operations, and then attempt to match the `rd`() operations before any `in`()'s.

9.6.4 Properties

Depending on the presence or absence of mutual exclusion in a distributed system, and the policies in effect, we can use paraDOS to reason about a variety of safety and liveness properties. The following is a brief discussion of how elements of paraDOS contribute to new and meaningful approaches to reasoning about such systems.

Important safety properties — that bad states are never reached — include whether or not a system is deadlock free, whether or not race conditions exist, and whether or not transition density remains within a desired threshold. Consider the problem of deadlock, and the canonical dining philosophers example. An instantiation of paraDOS very naturally represents a trace where all five philosophers pick up their left forks in one parallel event — including all 120 (5!) possible views (ROPEs) of that event. In the next transition, paraDOS demonstrates very elegantly the un-events of five (or fewer) philosophers attempting to pick up their right forks. Reasoning about the trace of this history, or any of the views, a condition exists where after a certain transition, only un-events are possible. ParaDOS's decoupling of distributed processes' internal computations from their communication behavior, using the abstraction of communication closures, helps us reason that the dining philosophers are deadlocked.

Liveness properties — that good states are eventually reached — are also important. Some examples of particular interest when using paraDOS to reason about these properties include true concurrency of desired events, eventual entry into a critical section, guarantee of message delivery, and eventual honoring of requests for service. Liveness properties are especially affected by system policies, such as those discussed in the previous section. Instances of paraDOS, with their parallel events and ROPEs, readily handle properties of true concurrency, such as those examples in Table 9.1. The un-events of paraDOS also facilitate reasoning with traces about properties of message delivery and eventual entry as follows. Guarantee of message delivery is the property that, for all traces where a delivery un-event occurs, a corresponding (successful) delivery event eventually occurs. Similar descriptions exist for entry into critical sections, requests for service, etc. Of course, beyond these formulations, traces in paraDOS are subject to the same restrictions as in CSP. In cases where infinite observation is possible, or required, undecidability results similar to those from the Halting problem apply.

Properties that are both safety and liveness, such as levels of parallelism, including maximal and minimal, are particularly well suited for paraDOS. The magnitude of parallel events in traces of computation can be transformed to our notion of transition density, a measurable quantity. Once we have done this, we can reason about possible traces, and ask whether, for each transition, all communication closures are chosen to be reduced, and whether this ensures that these closures all reduce successfully (i.e., no inappropriate un-events). The existence of un-events in a trace does not necessarily preclude the possibility of maximal parallelism, since un-events can be due to system resource unavailability. The absence of un-events from a trace is not sufficient to conclude the property of maximal parallelism, either. As just discussed, all communication closures must be chosen for possible reduction, and all eligible processes must be chosen to make internal computational progress. The latter condition requires that we abstract non-communication behavior as observable events.

9.7 Demonstration of Reasoning with ParaDOS

To demonstrate the utility of reasoning with parallel events and views, we present a case study of two primitive operations from an early definition of Linda. In addition to the four primitives `rd`(), `in`(), `out`(), and `eval`(), the Linda definition once included predicate versions of `rd`() and `in`(). Unlike the `rd`() and `in`() primitives, predicate operations `rdp`() and `inp`() were nonblocking primitives. The goal was to provide tuple matching capabilities without the possibility of blocking. The Linda predicate operations seemed like a useful idea, but their meaning proved to be semantically ambiguous, and they were subsequently removed from the formal Linda definition.

First, we demonstrate the ambiguity of the Linda predicate operations when restricted to reasoning with an interleaved sequential event trace semantics like that provided by CSP. The ambiguity is subtle and, in general, not well described in the literature. Next, we demonstrate how reasoning about the same computation with an appropriate instance of paraDOS disambiguates the meaning of the Linda predicate operations. The instance of paraDOS for Linda presented earlier in this dissertation did not include the predicate operations. We discuss attributes required by a new instance for this purpose. Finally, we discuss the importance of this model for reasoning about these extended tuple space computations.

9.7.1 Ambiguity

Predicate operations `rdp()` and `inp()` attempt to match tuples for copy or removal from tuple space. A successful operation returns the value one (1) and the matched tuple in the form of a template. A failure, rather than blocking, returns the value zero (0) with no changes to the template. When a match is successful, no ambiguity exists. It is not clear, however, what it means when a predicate operation returns a zero.

The ambiguity of the Linda predicate operations is a consequence of modeling concurrency through an arbitrary interleaving of tuple space interactions. Jensen noted that when a predicate operation returns zero, "only if every existing process is captured in an interaction point does the operation make sense." [Jen94].

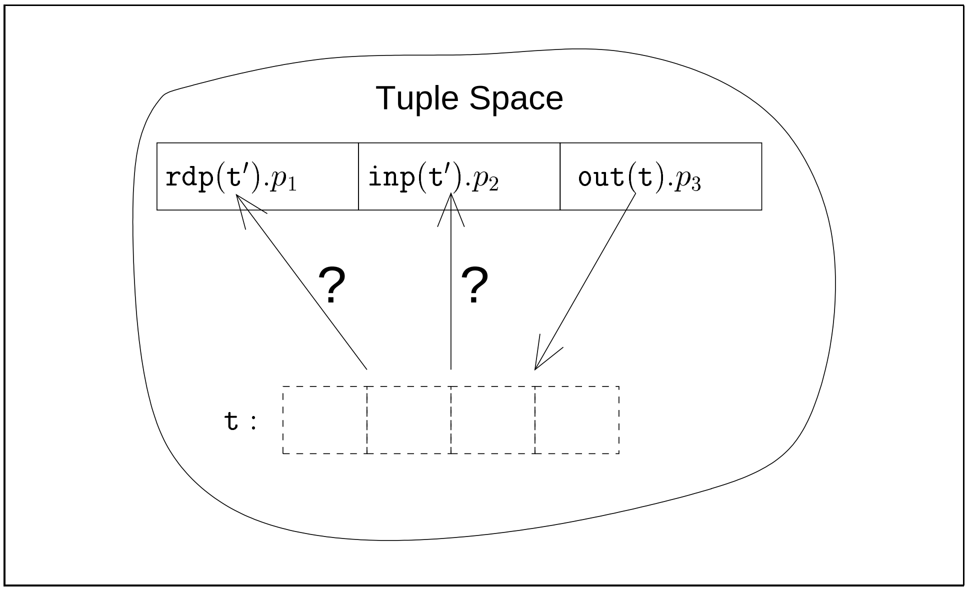

Figure 9.1: Case Study for Linda predicate ambiguity: an interaction point in tuple space involving three processes.

Suppose three Linda processes, p_1, p_2, and p_3, are executing concurrently in tuple space. Further suppose that each of these processes simultaneously issues a Linda primitive as depicted in Figure 9.1.

Assume no tuples in tuple space exist that match template $\mathtt{t}'$, except for the tuple $\mathtt{t}$ being placed in tuple space by process p_3. Together, processes p_1, p_2, and p_3 constitute an interaction point, as referred to by Jensen. There are several examples of ambiguity, but discussing one possibility will suffice. First consider that events are instantaneous, even though time is continuous. The outcome of the predicate operations is nondeterministic; either or both of the $\mathtt{rdp}(\mathtt{t}')$ and $\mathtt{inp}(\mathtt{t}')$ primitives may succeed or fail as they occur instantaneously with the $\mathtt{out}(\mathtt{t})$ primitive.

For this case study, let the observable events be the Linda primitive operations themselves. For example, $\mathtt{out}(\mathtt{t})$ is itself an event, representing a tuple placed in tuple space. The predicate operations require additional decoration to convey success or failure. Let bar notation denote failure for a predicate operation. For example, `inp(t')` represents the event of a successful predicate, returning value 1, in addition to the tuple successfully matched and removed from tuple space; $\overline{\texttt{rdp(t}'\texttt{)}}$ represents the event of a failed predicate, returning value 0.

The events of this interaction point occur in parallel, and an idealized observer keeping a trace of these events must record them in some arbitrary order. Assuming perfect observation, there are six possible correct partial orderings. Reasoning about the computation from any one of these traces, what can we say about the state of the system after a predicate operation fails? The unfortunate answer is "nothing." More specifically, upon failure of a predicate operation, does a tuple exist in tuple space that matches the predicate operation's template? The answer is, it may or it may not.

This case study involves two distinct levels of nondeterminism, one dependent upon the other. Since what happens is nondeterministic, then the representation of what happened is nondeterministic. The first level concerns computational history; the second level concerns the arbitrary interleaving of events. Once we fix the outcome of the first level of nondeterminism, that is, determine the events that actually occurred, we may proceed to choose one possible interleaving of those events for the idealized observer to record in the event trace. The choice of interleaving is the second level of nondeterminism.

Suppose in the interaction point of our case study, process p_1 and p_2's predicate operations fail. In this case, the six possible partial orderings an idealized observer can record are the following:

1. $\overline{\texttt{rdp(t')}} \longrightarrow \overline{\texttt{inp(t')}} \longrightarrow \texttt{out(t)}$
2. $\overline{\texttt{rdp(t')}} \longrightarrow \texttt{out(t)} \longrightarrow \overline{\texttt{inp(t')}}$
3. $\overline{\texttt{inp(t')}} \longrightarrow \overline{\texttt{rdp(t')}} \longrightarrow \texttt{out(t)}$
4. $\overline{\texttt{inp(t')}} \longrightarrow \texttt{out(t)} \longrightarrow \overline{\texttt{rdp(t')}}$
5. $\texttt{out(t)} \longrightarrow \overline{\texttt{rdp(t')}} \longrightarrow \overline{\texttt{inp(t')}}$
6. $\texttt{out(t)} \longrightarrow \overline{\texttt{inp(t')}} \longrightarrow \overline{\texttt{rdp(t')}}$

The idealized observer may choose to record any one of the six possible interleavings in the trace. All but the first and the third interleavings make no sense when reasoning about the trace of computation. Depending on the context of the trace, the first and third interleavings could also lead to ambiguous meanings of failed predicate operations. In cases 2, 4, 5, and 6, an `out(t)` operation occurs just before one or both predicate operations, yet the events corresponding to the outcome of those predicates indicate failure. It is natural to ask the question: "This predicate just failed, but is there a tuple in tuple space that matches the predicate's template?" According to these interleavings, a matching tuple `t` existed in tuple space; the predicates shouldn't have failed according to the definition of a failed predicate operation. The meaning of a failed predicate operation breaks down in the presence of concurrency expressed as an arbitrary interleaving of atomic events. This breakdown in meaning is due to the restriction of representing the history of a computation as a partial ordering of atomic events. Reasoning about computation with a sequential event trace leads to ambiguity for failed Linda predicate operations `rdp(t')` and `inp(t')`.

9.7.2 Clarity

Recording a parallel event sequentially does not preserve information regarding event simultaneity. With no semantic information about event simultaneity, the meaning of a failed predicate operation is ambiguous. The transformation from a parallel event to a partial ordering of that parallel event is one-way. Given an interleaved trace – that is, a partial ordering of events, some of which may have occurred simultaneously – we cannot in general recover the concurrent events from which that interleaved trace was generated.

A fundamental principle underlies the problem of representing the concurrency of multiple processes by interleaving their respective traces of computation: entropy. In this context, entropy is a measure of the lack of order in a system; or alternatively, a measure of disorder in a system. The system, for our purposes, refers to models of computation. There is an inverse relationship between the level of order represented by a model's computation, and its level of entropy. When a model's computation has the property of being in a state of order, it has low entropy. Conversely, when a model's computation has the property of being in a state of maximum disorder, it has high entropy. We state the *loss of entropy* property for interleaved traces.

> **Property:** (*Loss of Entropy*) Given a concurrent computation c, let trace tr be an arbitrary interleaving of atomic events from c, and let e_1 and e_2 be two events within tr, such that e_1 precedes e_2. A loss of entropy due to tr precludes identifying whether e_1 and e_2 occurred sequentially or concurrently in c.

By interleaving concurrent events to form a sequential event trace, a model (e.g., CSP) loses concurrency information about its computation. Interleaving results in a partial ordering of the events of a concurrent computation, an over-specification of the order in which events actually occurred. Concurrent models of computation that proceed in this fashion accept an inherent loss of entropy. A loss of entropy is not always a bad thing; CSP has certainly demonstrated its utility for reasoning about concurrency for a long time. But loss of entropy does limit reasoning about certain computational properties, and leads to problems such as the ambiguity of the Linda predicate operations in our case study.

The relationship between the trace of a computation and the multiple views of that computation's history reflects the approach of paraDOS to maintain multiple possible losses of entropy (i.e., views) from a single high level of entropy (i.e., parallel event trace). Furthermore, paraDOS views differ from CSP trace interleavings in two important ways. First, paraDOS distinguishes a computation's history from its views, and directly supports reasoning about multiple views of the same computation. Second, addressing the issue from the *loss of entropy* property, a view is a list of ROPEs, not a list of interleaved atomic events. The observer corresponding to a view of computation understands implicitly that an event within a ROPE occurred concurrently with the other events of that ROPE, after any events in a preceding ROPE, and before any events in a successive ROPE.

The parallel events feature of paraDOS makes it possible to reason about predicate tuple copy and removal operations found in commercial tuple space systems. A parallel event is capable of capturing the corresponding events of every process involved in an interaction point in tuple space. This capability disambiguates the meaning of a failed predicate operation, which makes it possible to reintroduce predicate operations to the Linda definition without recreating the semantic conflicts that led to their removal.

The additional structure within a view of computation, compared to that of an interleaved trace, permits an unambiguous answer to the question raised earlier in this section: "This predicate just failed, but is there a tuple in tuple space that matches the predicate's template?" By considering all the events within the ROPE of the failed predicate operation, we can answer yes or no, without ambiguity or apparent contradiction. In our case study from Figure 9.1, given both predicate operations nondeterministically failed within a ROPE containing the `out(t)` and no other events, we know that tuple `t` exists in tuple space. The transition to the next state doesn't occur between each event, it occurs from one parallel event to the next. For this purpose, order of events within a ROPE doesn't matter; it is the scope of concurrency that is important.

9.7.3 Importance

Our case study of the Linda predicate operations is important for several reasons. First, we demonstrated the power and utility of view-centric reasoning. Second, we provided a framework that disambiguates the meaning of the Linda predicate operations `rdp()` and `inp()`, making a case for their reintroduction into the Linda definition. Third, despite the removal of predicate operations from the formal Linda definition, several tuple space implementations, including Sun's JavaSpaces and IBM's T Spaces, provide predicate tuple matching primitives. ParaDOS improves the ability to reason formally about systems developed with commercial tuple space implementations by providing a framework capable of modeling the Linda predicate operations.

CHAPTER 10

Conclusions

We have presented a new parameterized model of parallel and distributed computation, paraDOS, and instantiations of two very different approaches to concurrency, the Actors model and the Linda communication language for tuple space. Our goals were to motivate the importance of views in reasoning about parallel and distributed computation, reveal useful abstractions for representing concurrency, and demonstrate the utility of operational semantics as an effective framework to model this computation.

We conclude this dissertation, beginning with a list of the primary contributions this research has made to the discipline of computer science. Section 10.1 reviews the state of reasoning about properties of concurrent systems — before paraDOS — as a basis for expounding upon each of our research contributions, and Section 10.2 discusses future work.

10.1 Contributions

Section 10.1.1 contains concise statements of our research contributions. The remaining sections discuss each of our contributions in more detail.

10.1.1 Concise Contributions

This research led to six major contributions:

1. Identification of the loss of entropy property for interleaved traces.
2. Introduction of two entropy-preserving abstractions — ordered and unordered parallel events — for representing event simultaneity within Hoare's CSP model.
3. Differentiation of a computation's history from its views, and direct support for reasoning about multiple, simultaneous views of a computation.
4. Creation of a general, composable model of computation — parameterized and capable of individual instantiation — for reasoning about properties of parallel and distributed systems.
5. Abstraction of a concurrent system's state, whose general definition includes process continuations, communication closures, parallel events, and a next state.
6. Utilization of view-centric reasoning to disambiguate the meaning, upon failure, of Gelernter's Linda predicate operations `rdp()` and `inp()`, in tuple space.

10.1.2 Loss of Entropy Property

Building on Hoare's seminal research that resulted in CSP, paraDOS has a proven model of concurrency from which to proceed. CSP provides the metaphor of an idealized observer recording the observable events of a concurrent computation, where concurrency is realized by communicating sequential processes. An event trace of an individual sequential process represents the history of that process's computation. Thus, in CSP, a history of a concurrent system is not a collection of individual event traces, but is rather a single trace that results from a sequential interleaving of those individual event traces.

In the case where the events from two or more processes occur simultaneously, CSP's observer interleaves those events in some sequential order. There is no incorrect order, since in a sequential event trace, the events must be recorded in some order. But once such an interleaving occurs, some potentially important information about the computation is lost, since the event trace represents a partial ordering, or overspecification, of the sequence of events in a computation's history.

Something apparently contradictory occurs when simultaneous events are interleaved in a trace. By specifying more information about event order, interleaving causes a loss of information concerning event simultaneity. This is a case where "more is less." The challenge is to identify a property for this phenomenon that does not confuse the issue further. Entropy is the measure of disorder in a system. A system with high entropy has a high level of disorder; low entropy corresponds to low disorder, or in the extreme, order.

An interleaved trace represents information from a system whose events may have occurred at a high level of disorder, but by interleaving simultaneous events, the CSP observer effects a loss of entropy for reasoning about the system. The characterization of loss of entropy is counter to what occurs in nature, where systems tend, over time, to increase their levels of entropy. For many computations, a loss of entropy is inconsequential; but for some computations, and more specifically for some reasonings about properties of computations, we need to model concurrency using an abstraction that preserves entropy.

10.1.3 Parallel Events and ROPEs

The challenge faced in this research is to preserve the usefulness of event traces as provided by CSP's process algebra, while extending the notion of event traces in a way that preserves entropy. To meet this challenge, paraDOS introduces new event abstractions, parallel events and ROPEs. A parallel event is an event aggregate, representing events of a computation observed to occur instantaneously in parallel. Parallel events serve as the primitives that form event traces in paraDOS. ROPEs are another event aggregate, denoting randomly ordered parallel events. A ROPE corresponds to some parallel event, and specifies a (possibly incomplete) partial ordering. In general, a parallel event has many possible corresponding ROPEs. ROPEs serve as the primitives that form views of computation in paraDOS.

Parallel events and ROPEs reveal the nondeterminism that results from a concurrent computation. By preserving entropy, parallel events convey levels of concurrency and provide intuition into other possible outcomes of nondeterminism. ROPEs provide intuition into the many possible perspectives (views) of a parallel event.

10.1.4 One History, Multiple Views

The paraDOS abstractions of parallel events and ROPEs permit us to distinguish a computation's history from possible views of that computation. We extend the notion of a CSP trace with parallel event primitives. In paraDOS, a trace is a sequence of parallel events — a parallel event trace. A parallel event trace corresponds to a computation's history.

ParaDOS introduces the notion of views. A view of computation refers to any (possibly incomplete) partial ordering of events from a computation's history. A view is constructed from a parallel event trace, built up from a sequence of ROPEs. Given a view of computation, each ROPE in the view corresponds positionally to its respective parallel event from the computation's trace. In general, for a given history of computation, multiple corresponding views of that computation's history are possible.

10.1.5 General Model for Reasoning

Many approaches to concurrent computation exist, and many models have been developed to reason about properties of such computation. When we wish to reason about different approaches to concurrency, it is useful to proceed from a common framework, rather than utilize separate computational models, with different abstractions. Our model is general enough to reason about many diverse approaches to concurrent computation, two of which are considered here, Actors and Linda. ParaDOS is an operational semantics, most of whose elements are parameterized; it establishes a framework of observable events, traces, views, and compositions. Based on CSP, paraDOS supports reasoning about properties of concurrent systems, such as deadlock and divergence. With its extensions, paraDOS provides abstractions for reasoning directly about properties related to event simultaneity and multiple views of computation.

10.1.6 Concurrent State Abstractions

One of the benefits of building a general model for reasoning about concurrency is the development of abstractions for representing the state of a concurrent system, independant of a system's approach to concurrency. Concurrent systems consist of a collection of processes capable of some form of interprocess communication. ParaDOS represents processes by their continuations, and instances of interprocess communication by bound expressions we call communication closures. Communication closures prove to be a unifying abstraction capable of representing a variety of communication paradigms. We abstract observable events from the communication behavior of a concurrent system. Generally, events arise from reductions of communication closures by the paraDOS transition relation. The collection of all such events that result from one state's closure reductions comprise the next state's parallel events. The transition relation chooses the next state to which computation proceeds.

10.1.7 Example of View-centric Reasoning

View-centric reasoning proves to be useful for describing the behavior and capabilities of concurrent systems. ParaDOS's parallel events and views provide a framework for reasoning about the predicate tuple space operations `rdp()` and `inp()` that Gelernter removed from Linda. Previous attempts to formally define these operations resulted in ambiguous meanings for some cases in which these predicates fail to match a tuple in tuple space. In Section 9.7.2 we demonstrate the problem and use view-centric reasoning to disambiguate the meaning of these failed predicate operations.

10.2 Future Work

There are several areas of future work that we plan to pursue. First, since commercial tuple space implementations support transaction semantics, we need to consider how paraDOS can be used to reason ablout such systems. Section 10.2.1 presents some initial thoughts on modeling transactions within the paraDOS framework, and Section 10.2.2 discusses two commercial tuple space implementations that are candidates for paraDOS instantiation. Finally, Section 10.2.3 presents other potential future work.

10.2.1 Transactions

For some systems we modeled (see Section 7.3), transactions were implicit, but this is not always the case. The approach to composition within paraDOS provides some clues toward an abstraction for transactions in distributed systems. The composition we presented in Chapter 8 may suggest an a priori (static) approach — indeed, this may have been true during the time we developed composition as a paraDOS parameter — but this need not be the case. If paraDOS utilizes composition to model transactions, we must accommodate the need to compose a system with existing system(s) dynamically, that is, at run time.

Transactions are initiated and then either committed or rolled-back at run time. One way to view a transaction is as a subprocess with the special quality that it only modifies its environment if it commits (success); in the case of roll-back, the environment reflects the state that would have existed had the transaction never been attempted. This all-or-nothing quality of transactions also suggests a natural filtering of observable events within transactions. It is also possible that views could play a role in modeling transactions within paraDOS (i.e., we can limit the observers of a transaction to be only those participating in the transaction).

10.2.2 Models of Commercial Systems

Relative to $\mathcal{P}^{\mathbf{TS}}$, we are investigating two major commercial tuple space implementations, JavaSpaces [FHA99] from Sun Microsystems and T Spaces [WML98] from IBM. Both JavaSpaces and T Spaces evolved from Gelernter's original work in Linda, but they evolved differently. We are in the process of using $\mathcal{P}^{\mathbf{TS}}$ to analyze both these implementations, and reason about their respective computational properties. This analysis could lead to the identification of new parameters for paraDOS. The paraDOS model might eventually be used as a tool to compare commercial tuple space implementations for the purpose of selecting the most appropriate system for different application needs.

For example, JavaSpaces and T Spaces both provide predicate, or asynchronous versions of Linda's *rd* and *in* operations, even though Gelernter removed both primitives from Linda due to semantic ambiguity (see Jensen [Jen91] for further discussion on asynchronous matching operations). The ambiguity is subtle, and elusive to understand, but view-centric reasoning demonstrates and disambiguates this problem. Similar issues with event notification primitives and other Linda extensions could possibly be exposed and formally understood.

Until recently, lack of efficient tuple space implementations limited reasoning about tuple spaces to academic pursuits. The ubiquity of the Internet and Java programming language, and the endorsement of companies like Sun Microsystems and IBM, have propelled Linda's popularity much closer to the forefront of distributed computing. Continued development of our paraDOS for Linda instantiation, toward a paraDOS for JavaSpaces or paraDOS for T Spaces instantiation, could provide an important framework for proving soundness properties about space based distributed protocols and systems.

10.2.3 Other Future Work

Other important areas of potential future work include modeling composition of heterogeneous systems, including the challenges associated with gateways and middleware in n-tier Internet applications. A first step toward modeling heterogeneous composition in paraDOS would probably consider how to represent a gateway between Actors and Linda programs.

We discussed one approach to modeling filtering with transactions and touched on the possible role views could play in this area. Regardless of how we model filtering, paraDOS's ability to filter events is important with respect to modeling security and scalability within distributed systems. In the case of security, we wish to intentionally filter certain events from certain observers. In the case of scalability, systems we model may have thresholds for maximum number of events before degrading performance, or in the case of relevance filtering, different observers may require the ability to observe different kinds or different numbers of events.

We just mentioned system performance as a scalability issue. In the background chapter of this dissertation (Chapter 2), we discussed the different purposes for models — prediction, description, or reasoning. While our research in developing paraDOS has focused on a model for reasoning about properties of concurrency, the process algebra paraDOS inherits from CSP may provide a bridge to performance modeling. This avenue of research became apparent to us during a presentation at the 2000 Future of Information Processing Symposium, in which Harrison [Har00] discussed current research investigating the use of stochastic process algebras (SPAs) to model systems composed of concurrently active cooperating components. What we believe makes paraDOS relevant to this is its view-centric approach, which can model probabilistic events. It is this connection which we intend to pursue to investigate the use of paraDOS as a tool to study both behavioral and performance properties.

APPENDIX A

Scheme Implementation of SECD Machine

This appendix contains my Scheme implementation of the SECD machine.

```
; *
; * transform: Transition function for the SECD machine
; *
(define transform
   (lambda (s e c d)
      (cond

         ; Case 7: if C = []
         ; -- Must be first. If c is null, can't check
         ;    anything else!
         ((null? c)
            ; If d is also empty stack, then finished,
            ; return top of stack s
            (if (null? d) (car s)
            ; else continue by popping saved environment
            ;    from dump stack d, and pushing result
            ;    currently on s on top of restored s
            (transform (cons (car s) (caar d)) ; -the new s
               (cadar d) (caddar d) (cadddar d))))

         ; Case 0: if head(C) is a func
         ((func? (car c))
            (transform (cons (car c) s) e (cdr c) d))

         ; Case 1: if head(C) is a constant
         ((const? (car c))
            (transform (cons (cadar c) s) e (cdr c) d))

         ; Case 2: if head(C) is a variable
         ((ident? (car c))
            (transform (cons (lookup e (cadar c)) s)
                       e (cdr c) d))

         ; Case 3: if head(C) is an application (Rator Rand)
         ((app? (car c))
            (transform s e
               (cons (cadar c) (cons (caddar c)
                       (cons (cons 'eval '()) (cdr c)))) d))
```

```
   ; Case 4: if head(C) is a lambda abstraction
   ;         lambda(V).B
   ((lambda-ab? (car c))
      (transform
         (cons (cons 'closure
            (cons (cadar c) (cons (caddar c)
                                   (cons e '()))))) s)
         e (cdr c) d))

   ; Case 5: if head(C) = @ and head(tail(S)) is a
   ;         predef. function f
   ((and (eval? (car c)) (func? (cadr s)))
      (transform (cons (eval (cadadr s) (car s))
                       (cddr s))
         e (cdr c) d))

   ; Case 6: if head(C) = @ and
   ;            head(tail(S)) = closure(V,B,E1)
   ((and (eval? (car c)) (closure? (cadr s)))
      (transform
         (mk-empty)
           ; which is the new s - empty
         (cons (cons (cadadr s)
                     (cons (car s) (mk-empty)))
               (cadddadr s))
           ; which is the new e, with V-->B added to
           ; E1 from closure
         (cons (caddadr s) (mk-empty))
           ; which is the new c - initialized to B
           ;                      from closure
         (cons
            (cons (cddr s)
                  (cons e
                        (cons (cdr c)
                              (cons d
                                    (mk-empty))))) d)))
           ; which is the new d - this saves the
           ; current cfg after eval

; close the cond, lambda, and define...
)))
```

```
; *
; * add1: A "predefined function" which does what it says...
; *
(define add1
   (lambda (n) (+ n 1)))

; *
; * app?: Boolean test which returns true when e an "application"
; *       of form ('app e e)
; *
(define app?
   (lambda (e)
      (eq? (car e) 'app)))

; *
; * closure?: Boolean test which returns true when e is a
; *           "closure" of form ('closure v b e)
; *
(define closure?
   (lambda (e)
      (eq? (car e) 'closure)))

; *
; * const?: Boolean test which returns true when e is a "const"
; * of form ('const n)
; *
(define const?
   (lambda (e)
      (eq? (car e) 'const)))

; *
; * func?: Boolean test which returns true when e is a
; *        "predefined function" of form ('func n), where n
; *        is the name of the function
; *
(define func?
   (lambda (e)
       (eq? (car e) 'func)))
```

```
; *
; * ident?: Boolean test which returns true when e is an
; *         "identifier" of form ('ident x)
; *
(define ident?
   (lambda (e)
       (eq? (car e) 'ident)))

; *
; * lambda-ab?: Boolean test which returns true when e is a
; *             "lambda abstraction" of form ('lambda x e)
; *
(define lambda-ab?
   (lambda (e)
      (eq? (car e) 'lambda)))

; *
; * eval?: Boolean test which returns true when e is a the
; *        "eval" symbol (@) of form ('eval)
; *
(define eval?
   (lambda (e)
      (eq? (car e) 'eval)))

; *
; * eval: Function which associates the symbol f with a
; *       predefined function of the same name, then returns
; *       the result of applying a to f.
; *
(define eval
   (lambda (f a)
      (if (eq? f 'add1)
         (add1 a)
         (display (cons f (cons a '())))))
         ;'error)))

; *
; * mk-empty: Returns an empty list
; *
(define mk-empty
   (lambda () '()))
```

```
; *
; * lookup: Checks environment env for variable name, and if
; *         found returns its bound value
; *
(define lookup
   (lambda (env name)
      (cond
         ((null? env) 'error)
         ((eq? name (caar env)) (cadar env))
         (#t (lookup (cdr env) name)))))

; *
; * update-env: Adds the binding of name to val to the
; *             environment env
; *
(define update-env
   (lambda (env name val)
      (cons (list name val) env)))

; *
; * caddadr: Apparently I reached a limit?
; *
(define caddadr
   (lambda (s)
      (car (cdr (cdr (car (cdr s)))))))

; *
; * cadddadr: Apparently I reached a limit?
; *
(define cadddadr
   (lambda (s)
      (car (cdr (cdr (cdr (car (cdr s))))))))

; *
; * cadddar: Apparently I reached a limit?
; *
(define cadddar
   (lambda (s)
      (car (cdr (cdr (cdr (car s)))))))
```

The following are three test functions for the SECD transform function.

```
(define test1
   (lambda ()
      (transform '() '()
         '((app (func add1) (const 6))) '())))

(define test2
   (lambda ()
      (transform '() '()
         '((app (lambda x (app (func add1) (ident x)))
                              (const 6))) '())))
(define test3
   (lambda ()
      (transform '() '()
         '((app (lambda x (ident x)) (const 42))) '())))
```

Thus, in the Scheme Read-Eval-Print loop, the following transactions occur:

```
1 ]=> (test1)

;Value: 7

1 ]=> (test2)

;Value: 7

1 ]=> (test3)

;Value: 42
```

List of References

[AC93] Gul Agha and Christian J. Callsen. "ActorSpace: an open distributed programming paradigm." *ACM SIGPLAN Notices*, **28**(7):23–32, July 1993.

[AC94] Gul Agha and Christian J. Callsen. "Open Heterogeneous Computing in ActorSpace." *Journal of Parallel and Distributed Computing*, pp. 289–300, 1994.

[Agh86] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1986.

[AMS97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. "A Foundation for Actor Computation." *Journal of Functional Programming*, **7**(1):1–72, 1997.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[Cal94] Christian J. Callsen. *Open Distributed Heterogeneous Computing.* PhD thesis, Aalborg University, Denmark, 1994.

[CG89] Nicholas Carriero and David Gelernter. "Linda in Context." *Communications of the ACM*, **32**(4), April 1989.

[CJY94] Paolo Ciancarini, Keld K. Jensen, and Daniel Yankelevich. "On the Operational Semantics of a Coordination Language." In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pp. 77–106. Springer Verlag, 1994.

[CT90] K. Mani Chandy and Stephen Taylor. "A primer for program composition notation." Technical Report CRPC-TR90056, California Institute of Technology, Pasadena, CA, jun 1990.

[CT92] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, MA, 1992.

[DD90a] Philippe Darondeau and Pierpaolo Degano. "Causal Trees: Interleaving + Causality." In Irene Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pp. 239–255. Springer Verlag, 1990.

[DD90b] Philippe Darondeau and Pierpaolo Degano. "Event structures, Causal trees, and Refinements." In Branislav Rovan, editor, *Mathematical Foundations of Computer Science 1990*, volume 452 of *LNCS*, pp. 239–245. Springer Verlag, 1990.

[DDM88] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. "Partial Orderings Descriptions and Observations of Nondeterministic Concurrent Processes." In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pp. 438–466. Springer Verlag, 1988.

[Dij71] E. W. Dijkstra. "Hierarchical Ordering of Sequential Processes." *Acta Informatica*, **1**:115–138, 1971.

[DSW94] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science.* AP, AP:adr, second edition, 1994.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice.* The Jini Technology Series. Addison Wesley, 1999.

[FOT92] Ian Foster, Robert Olson, and Steven Tuecke. "Productive Parallel Programming: The PCN Approach." *Scientific Programming*, **1**:51–66, 1992.

[Gel85] David Gelernter. "Generative Communication in Linda." *ACM Transactions on Programming Languages and Systems*, **7**(1), January 1985.

[Har00] Peter Harrison. "Performance Modeling." Guest speaker, The Future of Information Processing Symposium, October 2000. Prof. Harrison discussed the role of stochastic process algebras in current performance modeling research.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics.* Wiley, New York, 1990.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice-Hall International, UK, Ltd., UK, 1985.

[Hoa94] C.A.R. Hoare. *Unified Theories of Programming.* Unpublished Monograph, July 1994.

[How93] "The Free On-line Dictionary of Computing." URL http://www.foldoc.org/, 1993. Editor Denis Howe.

[Jen91] Keld K. Jensen. "Decoupling of Computation and Coordination in Linda." In D. Heidrich and J.C. Grossetie, editors, *Computing with T.Node Parallel Architectures*, pp. 43–62. unknown, 1991.

[Jen94] Keld K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, Aalborg University, November 1994. http://www.cs.auc.dk/-research/FS/teaching/PhD/mts.abstract.html.

[Jen00] Keld K. Jensen, February 2000. Personal communication with Marc L. Smith.

[Kah87] Giles Kahn. "Natural Semantics." In F. Bandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pp. 22–39, Berlin, 1987. Springer Verlag.

[Lan64] Peter Landin. "The Mechanical Evaluation of Expressions." *The Computer Journal*, **6**(4):308–320, January 1964.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd, Campus 400, Maylands Avenue, Hemel Hempstead, Hertfordshire, HP2 7EZ, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[MLB76] Michael Marcotty, Henry Ledgard, and Gregor Bochmann. "A Sampler of Formal Definitions." *Computing Surveys*, **8**(2):191–276, 1976.

[MT97] Ian A. Mason and Carolyn L. Talcott. "A Semantics Preserving Actor Translation." *Lecture Notes in Computer Science*, **1256**:369–378, 1997.

[Nar89] James E. Narem. "An Informal Operational Semantics of C-Linda V2.3.5." Technical report, Yale University, December 1989. YALEU/DCS/TR-839.

[Oxf97] Frank R. Abate, editor. *The Oxford Desk Dictionary and Thesaurus.* Oxford University Press, Inc., american edition, 1997.

[Plo81] Gordon Plotkin. "A Structural Approach to Operational Semantics." Technical report, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall Europe, 1998.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Worldwide Series in Computer Science. John Wiley & Sons, Ltd., 2000.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley, Reading, Massachusetts, 1995.

[SM99] Laura Semini and Carlo Montangero. "A Refinement Calculus for Tuple Spaces." *Science of Computer Programming*, **34**(2):79–140, June 1999.

[SPH98] Marc L. Smith, Rebecca J. Parsons, and Charles E. Hughes. "Operational Semantics for Actors: Toward a Parameterized Model for Reasoning about Parallel and Distributed Computation." Technical Report CS-TR-99-05, School of Computer Science, University of Central Florida, 1998.

[WML98] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. "T Spaces." *IBM Systems Journal*, **37**(3):454–474, 1998.